MODULAR LOCAL SEARCH:
A FRAMEWORK FOR SELF-ADAPTIVE METAHEURISTICS

A THESIS PRESENTED IN PARTIAL FULFILMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN DECISION SCIENCE AT
MASSEY UNIVERSITY

David Colin Woods

2010

Abstract

This research develops Modular Local Search (MLS), a framework such that trajectory-based metaheuristics can be expressed as subsets of "modules" from a common library, with a common structure. The standardized modules and structure allow the easy formulation of common metaheuristic paradigms, as well as the easy creation of relatively complex hybrids by simply listing the modules that should be included. A new markup language called Modular Local Search Markup Language (MLSML) is developed so that new metaheuristics can be implemented *declaratively*, rather than *programmatically*.

Some advanced ideas are introduced and explored, whereby metaheuristics are able to modify themselves during their execution, by varying parameters and swapping modules into and out of activation. This ability introduces the potential for semi-intelligent algorithms that are capable of a type of learning. Several demonstration methods are developed and these show promise on a small test set of problem instances.

A new combinatorial optimization problem is developed to serve as the testing ground for the new heuristic ideas. The Arc Subset Routing Problem (ASRP) involves routing a vehicle on a graph, choosing a subset of the arcs, such that the reward collected by traversing these arcs is maximised subject to a constraint on the total distance travelled. This problem is first formulated and explored as a traditional Operations Research investigation; construction heuristics are developed, as well as some improvement routines for local search, and computational tournaments are performed to compare the methods.

Some attention is given to developing methods to predict which of two heuristics is most suited to a given problem instance, based on an analysis of the characteristics of that problem. Initial results demonstrate the potential of such an approach.

The MLS framework offers a powerful and flexible structure both for the easy and consistent implementation of existing metaheuristics, and also as a platform for the development of new, advanced metaheuristic ideas. Early results are encouraging, and a number of directions for future research are discussed, including some complex real-world problems for which the self-adaptive capabilities of MLS would be especially useful.

Acknowledgements

I would like to thank all the many people who have supported and encouraged me over the years. It has been a long journey, including several complete changes in direction, and the effort to complete this work has been made easier by their understanding.

First and foremost I need to express my gratitude to my supervisors, Mark Bebbington and John Giffin. Mark, who took over as chief supervisor after John moved to Canterbury, has served as a constant reality check and I appreciate his resisting the urge, overwhelming at times I'm sure, to wash his hands of me as the pressures of developing a career and a family meant that my progress was, at times, intermittent. Extra special thanks are due to John, who served as my mentor and friend during my undergraduate years. Many hours were spent in his office discussing the world and Operations Research, and many ideas were discussed, including the germs of what later became this research, although via a circuitous route. Since then he has provided much timely advice, and encouragement to PhD thinking, which has served, barely, as a restraint on my own tendency to bite off more than I can chew. His critical eye has also prevented many potentially embarassing typos and misspellings, although of course I take responsibility for any mistakes in the final thesis, mindful of the words of Randy Milholland, who said that "*typos are very important to all written form. It gives the reader something to look for so they aren't distracted by the total lack of content in your writing*". I should also acknowledge a former PhD student of John's, Mark Johnston, the formatting and layout of whose thesis I shamelessly copied.

Continuing to work on this research over these many years would not have been possible without the support of my boss, Graeme Gee. As well as providing me the opportunity to develop my career in analytics consulting, which has allowed me to gain a hands-on appreciation for techniques that work in the real world, and the complexities of "real" optimization problems, he has been unfailingly supportive. This support has extended to dedicated periods of time where I could focus on my PhD research, financially sponsoring my study, including fees and any text books I decided I must have, and a steady stream of computing resources. In the last stages of writing up this thesis he even drove me to a meeting at Massey because I hadn't been getting enough sleep to drive safely. His unwavering support has made the completion of this work possible.

Last, but certainly not least, I need to thank my family and friends. My friends, who have offered many opportunities to escape for a time from thinking about anything related to Operations Research, in fact probably too many opportunities. My family, who have always had faith that I would finish; one might say blind faith, but that is what families are for. Especial thanks to my Nana, who has been patiently looking forward to my finishing for more years than I care to count, and to my father, who never fails to nag me about it. My biggest appreciation is reserved for my partner Vin; she has been heroically patient and supportive, especially over the last few months of the write-up while I have been absent in order to devote myself to it. She undertook, mostly without complaint, essentially to act as a single mother to our daughter Tui while I industriously finished this thesis. Thanks are also due to Tui; the burning desire to get back to see her motivated more late nights and early mornings than are really healthy – she finally provided my inspiration to finish, regardless of how "finished" I feel.

Table of Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

Introduction

1.1 Local search and metaheuristics

A major part of Operations Research is concerned with methods of solving combinatorial problems; this is known as *combinatorial optimization*. A combinatorial problem is one which has a finite, but often very large, number of solutions, and for which it is possible to evaluate an objective function. To solve a combinatorial problem it is desirable to use an *algorithm*, a method which is guaranteed to find the best solution, the optimum (or rather *an* optimum, if there are multiple best solutions with the same objective function value). The most basic algorithm is called Explicit Enumeration, and it involves simply evaluating all of the possible solutions, and then choosing the best one.

Algorithms are fine for small problems, and for problems with structures that they are suited for, however there are a large class of problems where the effort required to solve them with exact methods grows exponentially (or worse) with the size of the problem. For these problems it is believed that there can be no algorithm that *guarantees* an optimum in a reasonable amount of time. In these cases *heuristics* are used. A heuristic is a method for finding a solution, like an algorithm, except that this solution may not be the optimum. The trade off is that heuristics usually run much faster than algorithms; their aim is to find a reasonable solution in a reasonable amount of time. Note that we can also use the word "algorithm" in the sense of a set of step-by-step instructions, in which case we distinguish between exact algorithms and approximation, or heuristic, algorithms.

Early heuristic research concentrated on developing methods that were particular to a single problem. An example is the Nearest Neighbour heuristic for the Travelling Salesman Problem (TSP). Given a set

of cities, and knowing the distances between the cities, the Travelling Salesman Problem is to find a tour through every city, and back to the starting point, so that the total distance travelled is minimized; i.e., what order should the cities be visited in? The Nearest Neighbour heuristic is to start at any arbitrary city and select the nearest unvisited city to visit next, repeating this until all cities have been visited, and then returning to the starting point. Nearest Neighbour is an example of a *construction heuristic*, a technique that assembles a solution to the problem iteratively; at each step a partial solution is maintained, and a complete solution is found only at the completion of the heuristic.

The next development was the concept of *local search* methods. These heuristics were often used as improvement routines that were applied after a construction heuristic had found an initial solution. In a local search heuristic a full solution to the problem is transformed into another solution by a perturbation procedure called a *move*. For the TSP an example of a move is swapping the positions of two cities in the ordered sequence that defines the solution. The set of solutions that can be reached by the application of one move defines the *neighbourhood* of a solution. The set of all solutions that can be reached eventually by the repeated application of a set of moves is the *search space* for a problem instance, and local search heuristics are so named because they "search" through this space for good solutions.

The most basic local search heuristic is simply a *random walk*. This method chooses a neighbour at random at each iteration, and good solutions are found by chance. A more effective strategy is to use a hill-climbing approach, where at each iteration the search chooses a solution that has a better objective function value than the current solution until there are no improving solutions within one move, in which case a *local optimum* has been reached. For a given problem instance, and depending on the moves available, which define the *topology* of the search space, there may be many local optima, many of which may be severely sub-optimal.

Much research has been devoted to developing heuristics that have mechanisms to avoid or escape local optima; these methods may be broadly termed *metaheuristics*. Although metaheuristics can, and often do, include problem-specific components, there are several "families" of techniques that have been developed over time that are rather more problem-agnostic; they are general techniques that are designed to efficiently search a solution space. They operate through "moves" on "solutions" to find "neighbours"; the actual mechanism of a move is not relevant to the metaheuristic routine's logic, although the quality of the move structures influences the ability of the metaheuristic to find good neighbours, so the development of appropriate moves is still important.

An example of an early metaheuristic is Simulated Annealing, introduced in 1983 by Kirkpatrick et al. [160]. Starting from an initial solution the metaheuristic evaluates neighbours one at a time. If a neighbour improves the current solution value then it is accepted and becomes the new current solution. If it is non-improving then it is accepted with a certain probability, $p = e^{-\delta/T}$, where δ is the difference between the objective values of the neighbour and the current solution, and T is a parameter called the *temperature* that decreases slowly as the search progresses. This has the effect that non-improving solutions are accepted near the beginning of the search process but become progressively less likely, intensifying the search as it (presumably) finds a good region of the search space.

It is worth noting that none of these definitions are absolute. For example there are metaheuristics that are not based on a local search approach, including many that are hybrids with more traditional algorithmic approaches such as Branch-and-Bound. Also, the problems that metaheuristics can solve are not necessarily combinatorial. It is even not necessary that the solution space be finite; so long as there is some defined neighbourhood structure, and a method for evaluating solutions, their techniques may be applied.

1.2 Trajectory-based metaheuristics

The field of this research is in what may broadly be described as *metaheuristics*. However, there are many different types of metaheuristic, and we restrict our attention to a particular sub-genre. Metaheuristics can be classified in many different ways; for example: deterministic or stochasic, sequential or parallel, memoryless or with memory, naturally terminating or not. Perhaps the most fundamental distinction is between *trajectory*-based methods and *population*-based methods.

Population-based metaheuristics maintain a pool of solutions simultaneously, and each iteration represents a "generation" of solutions that are modified and, especially, combined with each other to form the next generation. In this way the population tends to "evolve" towards better solutions. Perhaps the most popular are Genetic Algorithms, which mimic the process of evolution by artificial selection, which is an incredibly powerful process in the real world. Many aspects of biological processes have been incorporated into their algorithm analogues, such as crossover combinations and mutations, with significant success, and population-based metaheuristics are a thriving field of study.

Trajectory-based metaheuristics instead maintain a single solution at a time, and a conceptual trajectory is formed by the path of the search through the solution space. Trajectory-based metaheuristics generally have mechanisms to intensify the search in a promising area, or diversify the search away from a non-promising, or no-longer-promising, area (a local optimum is no longer promising once it has been found). These metaheuristics too have a rich history of development, and modern metaheuristics are growing ever more sophisticated. What were initially a number of distinct "families" of techniques are beginning to blur and merge as hybrids become more prevalent.

In this research we restrict our attention to trajectory-based metaheuristics, and develop a taxonomy and framework which accomodates the trend towards hybridization and sophisticated multi-phase techniques. Our attention is further focussed on sequential, rather than parallel, methods, although these are briefly discussed. For convenience we use the terms "heuristic", "metaheuristic", and "local search" interchangeably to refer to sequentially processed trajectory-based techniques, except where it is necessary to make a distinction.

1.3 Introducing Modular Local Search

The main challenge of this research is to formulate a taxonomy of (trajectory-based) metaheuristics, and develop a framework whereby these can be expressed in a common "language". We introduce **Modular Local Search** (MLS). MLS is a conceptual metaheuristic template consisting of a number of components that are capable of expressing most of the main trajectory-based metaheuristics. We

identify, and explicitly formalize many concepts that are common in metaheuristic literature, but are often implied, assumed, or have inconsistent definitions. We also incorporate several novel concepts.

MLS is designed to be *modular*, such that each MLS component fulfils an architectural role, and there are many potential ways that each role can be implemented, corresponding to specific functions (modules) derived from various metaheuristics. As an example, one of the MLS components is the **admissibility condition**. Consider the neighbourhood of the current solution, defined as all the solutions that can be reached with the application of one move. For most heuristics only a subset of the neighbours are actually eligible to be chosen as the next solution in the trajectory; in ascent-based heuristics only *improving* solutions are eligible, in Tabu Search (see Section 5.4.1) only solutions that have not previously been made *tabu* are allowed, and in Simulated Annealing (see Section 5.3.1) solutions are accepted *probabilistically*. Each of these conditions is a "module" that can act in the admissibility condition role. MLS defines a number of other components that, acting together, define any desired metaheuristic.

The fundamental structure around which MLS is based is the **search iteration process**, which in one iteration starts with a given solution and chooses a neighbour solution according to the **search scheme**. The search scheme defines the search *topology* (which solutions are in the neighbourhood, which are admissible, and how attractive they are in relation to each other) and the search *logic* (how many neighbours are examined, and in what order, before the search chooses the next solution in the trajectory).

Most metaheuristics act through some mechansim to alter the search scheme. MLS introduces a new concept to faciliate and control this process: **triggers and responses**. The trigger-response model is, to the best of the author's knowledge, a novel approach to metaheuristic control; it acts as the "intelligence" of the metaheuristic. After each iteration of the *search iteration process* the **triggers** are checked. These are functions that check the state of the search for predefined conditions or events. If a trigger is "tripped" then the **responses** associated with that trigger are performed. The responses perform tasks to modify the heuristic somehow, usually changing the search scheme.

Every heuristic has a termination criterion, which defines when the heuristic should stop. Commonly used termination criteria are when a local optimum is reached, or when a certain number of iterations have elapsed. In MLS the criterion would be the trigger, and the response would be *termination*. However, the true power of the trigger-response model is that it is *reflexive*; the responses are not only able to modify the search scheme, they are able to modify any other part of the MLS heuristic, *including the triggers and responses themselves*. Due to its modular nature, when the appropriate trigger is tripped the heuristic is able to completely transform itself. For example transforming from Tabu Search to Simulated Annealing would simply require a set of responses to deactivate the appropriate Tabu Search modules and activate the corresponding Simulated Annealing modules.

The other primary feature of MLS is the formalized memory structures. All the parameters required for the operation of the search scheme (such as the number of neighbours to examine), as well as any parameters that are required for specific modules (such as the tabu tenure) are stored as **memory parameter** items that can be read by modules that need them and modified by responses. In addition

other memory structures, for example *lists* such as the *tabu list* or a list of *elite solutions*, can also be defined and modified. MLS has a number of specialised components that can be used to maintain these memory structures, for example immediately after the search iteration process but before the triggers are checked is an **update-memory** phase, where any memory update modules can be specified. The final aspect of memory is a large number of automatically updated **counters** that keep track of the number of iterations, the number of times a trigger has been tripped, etc.

1.4 Research overview

The original research topic of this PhD was subset-selection arc routing problems, following on from an honours project of the author on the Snowplough Routing Problem. Preliminary reviews of the literature led to an observation that most research in combinatorial optimization tended to perform one of the following activities: development of a new heuristic or variation, and application of this to a single, or limited number of, problems; development of a new problem or variation, and application of a few basic heuristics to it; or the application of existing heuristics to existing problems, with some evaluation of which is better in this case. Each of these activities is an important contribution to the body of knowledge, but it seemed that there was no systematic way to compile this knowledge, to create a standardized method for determining which heuristics were suited for which problems, and under what conditions. Each piece of research seemed to contribute a number of isolated data points, but because of different ways of structuring the heuristics and approaches to experimentation there was no way to effectively combine these into a unified model. Hooker [144,145] made a similar complaint and urged a more "scientific" approach to testing heuristics, although did not propose a specific framework for doing so. James [150] echoed these ideas and sketched some suggestions for how this could be implemented; these ideas were formative in the early development of MLS.

The need for a structured framework to support these ideas was the motivation for Modular Local Search. While programming metaheuristics for the ASRP it became apparent that the different techniques actually have most of their operation in common, and differ from each other in specific operations. This lead to identifying the aspects that these heuristics have in common, and those they do not, and development of the idea of modules that can "slot in" to specific roles; a heuristic could then simply be represented as a list of modules. The reasoning was that this standardized structure would provide a basis for meaningful comparisons between heuristics, and the structure would impose a type of *heuristic space*, where small tweaks in parameters or modules would represent heuristics "close" in heuristic space, and large differences in modules would represent heuristics "far apart" in heuristic space. A similar structure imposed on problems would then allow an empirical modelling of the relationshiip between *problem space* and *heuristic space*.

This type of investigation was briefly explored in the research; in the main experimental investigation of Chapter 6 one of the analysis goals is to determine whether it is possible to predict which of two heuristics will perform best on a certain problem instance, based solely on an analysis of the characteristics of that problem instance. Although this was not a major research imperative, the results of the analysis validate the concept that this is possible, and developing a systematic methodology for this may be a fruitful direction for future research.

Another side investigation in the research concerns the generation of "interesting" sets of problem instances. The motivation for this investigation again derived originally from the concept of modelling the relationship between problem characteristics and heuristic performance. Two approaches are introduced. The first attempts to select a "maximally diverse" subset of problem instances from a larger set that is randomly generated. The second method actually applies a version of MLS to *design* problem instances with desired properties, which in this case is a large performance differential between two heuristics.

The main direction of research eventually focused on the potential of the MLS framework to not only express existing metaheuristics in a standardized and comparable way, but to enable the efficient creation of new metaheuristics, utilizing the implicit hybridization that follows from modularization and the self-adaptive capabilites that result from the trigger-response model. MLS was developed in the Java programming language as an object-oriented framework that is problem-independent to as large a degree as possible.

The goal of the experimentation in this research is to demonstrate and validate the applicability of the MLS framework to the field of metaheuristic design, rather than to attempt to find a single new metaheuristic that performs better than existing methods. This is done in three phases:

1. A number of standard metaheuristics are modelled as MLS, and extensive experimentation is performed.

2. Some basic hybrids of the standard metaheuristics are developed to demonstrate the ease with which this can be performed by mixing and matching modules with no new programming required.

3. Several advanced metaheuristics are developed using the same modules used previously. These are briefly applied to some test problems as proof-of-concept demonstrations of the flexibility of MLS. The results suggest that these approaches have significant potential to facilitate advanced metaheuristic design.

Another goal of the experimentation was to demonstrate the flexibilty and generality of the MLS framework. A new markup language was developed so that new, quite sophisticated, metaheuristics can be expressed *declaratively*, by specifying the combination of modules and parameters, rather than programmatically. The modules of MLS are re-usable, and each additional metaheuristic that is modelled as MLS contributes to a "toolbox" of modules that can be mixed and matched to create new combinations.

Through the course of the research, MLS was applied to three distinct problem domains:

- The Arc Subset Routing Problem is the main problem domain;

- MLS is used to construct problem instances with desired properties, where a "solution" is a problem instance for the ASRP and "moves" involve changing the structure of the problem instance by adding and removing arcs from the graph;

- MLS is used to design other MLS heuristics. This is one of the advanced applications, and is made possible by the combination of the modular nature of MLS, and the object-oriented implementation that allows a "solution" to be anything desired.

1.5 Research questions and goals

There are two major research questions that motivate and guide the various investigations throughout this thesis.

1. Is it possible to predict the relative performance of heuristics on a problem instance based on analysis of the problem instance prior to running the heuristics?
2. Can we develop a modular metaheuristic system, that encapsulates most trajectory-based local search methods, that allows easy hybridization of metaheuristics, and is not simply a programming convenience but also supports the creation of *new* types of metaheuristics?

These two research questions are inter-related, in that a modular structure for metaheuristic components allows more systematic experimentation and analysis relating heuristic performance to problem instance characteristics.

Each of the investigations in the thesis supports or informs one of these research questions.

1.6 Other frameworks

As may be expected, various frameworks have been developed by other researchers. These may be broadly classified as one of two types: *conceptual* frameworks, and *programmatic* frameworks.

1.6.1 Conceptual frameworks

Conceptual frameworks attempt to unify the various metaheuristic paradigms within a single conceptual model. James [150] explores a model that abstracts several of the functions of a metaheuristic, chiefly those aspects that influence the search topology: the neighbourhood scheme and the fitness function. He briefly notes that these two points are where many metaheuristics operate. This paper inspired much of the original conception of MLS, which perhaps may be seen as an extension of these ideas. Vaessens et al. [241] propose another high-level framework, defining functions such as GENERATENEIGHBOURS() and REDUCENEIGHBOURS() to control the neighbourhood exploration. They have more of a focus on the intersection with population-based methods, with functions to generate a population from a single solution. They seem to have a similar motivation to MLS; they hint at the ability to create hybrids by using procedures that originated with more than one source metaheuristic.

Tabu Search itself may be thought of as a conceptual framework. Although the basic heuristic itself acts through a clearly defined mechanism, the *tabu list*, it has grown to encompass many other ideas, especially associated with Adaptive Memory Programming. The seminal book by Glover and Laguna [120] contains many ideas on how to express and extend metaheuristics, also incorporating the ideas of Scatter Search (which may also be considered a framework in its own right).

Many other authors have also created "generalizations" of one or more local search approaches (for example, [46]), and many hybrid heuristics are motivated by this approach. Hoos and Stützle [146] develop the idea of a *Generalized Local Search Machine*; a theoretical state-based framework that encompasses most metaheuristics. This model is an interesting unification of metaheuristic concepts, but seems more useful as a method of conceptualising these ideas than it is helpful in implementing or extending them.

1.6.2 Programmatic frameworks

The other type of framework that has been attempted multiple times is a programmatic framework. These are typically collections of classes in some programming language, and are usually extensible.

Whereas one of the goals of MLS is to provide for *new* types of metaheuristics, the frameworks that the author discovered in the literature were not means of extending metaheuristic research, they were primarily methods of saving time when implementing heuristics; many of the functions which would need to be written each time are standardized. However, they all seemed to be only the beginnings of a really useful framework; the authors present a basic framework that identifies several operations that heuristics have in common, such as generating the neighbourhood, and then develop specialised functions to implement Tabu Search, Simulated Annealing, and perhaps one or two more metaheuristics.

Di Gaspero and Schaerf [70] present **EasyLocal++**, which they describe as an object-oriented framework for local search algorithms. This heuristic provides a basic structure for implementing metaheuristics, although each metaheuristic must still be programmed separately; the framework seems to offer the structure around the use of the heuristic on problems: output classes, problem data, etc. There is little abstraction of the metaheuristic concepts.

Michel and Van Hentenryck [189,190] develop **Localizer**, a modelling language for local search. This framework has some similarities with MLS, in that it defines some standard operations that are common to many metaheuristics, and a particular heuristic can be expressed by specifying which functions fulfil which roles, with standardized structures for aspects like the Acceptance Criterion.

Andreatta et al. [6] describe the **LocalSearch** framework, an object-oriented package that attempts to abstract some of the components of a local search routine, but again tends to package metaheuristics into distinct packages; it features a TabuSearch class and SimualtedAnnealing class as subclasses of LocalSearch, rather than the MLS approach of modularizing the different components of these.

Perhaps the most promising programmatic framework is the **HotFrame** framework of Fink and Voß [96]. This framework provides a robust object-oriented architecture that breaks all the components required to implement local search techniques into objects and classes: problems, solutions, neighbours, moves and move attributes. Components of metaheuristics such as *tabu attributes* are also treated as objects. One of the key advantages of HotFrame is that is defines the architecture that allows the metaheuristics to interact with problem data in a standardized way, and handles the interaction between the objects of different classes. Similar to the previously described frameworks, HotFrame appears primarily to be an aid to speed up implementation, and make doing so more consistent, with

much reusable code, rather than a method of extending the metaheuristics concepts themselves. Each metaheuristic is still programmed separately as a collection of specific packages, rather than generic swappable modules.

Fink et al. [97] briefly summarize several other frameworks: Templar, NeighbourSearcher, iOpt, and ILOG.

1.6.3 The place of MLS among other frameworks

Modular Local Search offers some unique points of difference with other frameworks. It may be considered both a conceptual framework, *and* a programmatic framework.

Like the other conceptual frameworks it abstracts the components that metaheuristics have in common, however it goes significantly further along this path than other frameworks. For example, there are four distinct places in which the scope of the neighbourhood examination can be controlled, each with nuanced differences: the move-list size, the examinations maximum, the candidate list size, and the optional neighbourhood reduction process. Like programmatic frameworks, MLS provides an architecture such that most code can be reused between heuristics, and developing new metaheuristics only requires the specific logic for the parts that are new. Again though, MLS goes further than existing frameworks; there is no real concept of a metaheuristic such as Tabu Search within MLS, instead Tabu Search is simply a collection of modules, most of which may be in common with Simulated Annealing. MLS blurs the lines between metaheuristics, and encourages the creation of hybrids, for example a hybrid that combines the main features of Tabu Search and Simulated Annealing is created in Section 6.5.1.1. This hybrid analyses the admissibility of neighbours based on their tabu status *and* the Metropolis probability criterion.

The key difference of MLS from other frameworks, and what is its main contribution, is its potential for advanced applications such as multi-phase heuristics, self-adaptation, and learning. This is made possible because of the combination of explicit components that fulfil specific roles in all local search metaheuristics, modules that can easily be swapped in and out of use, and the trigger-response model.

The trigger-response model is a new contribution that underlies the ability of an MLS heuristic to completely change its structure between iterations. For example, consider a metaheuristic that performs a Steepest Ascent search until it reachs a local optimum, then changes to Tabu Search with a different set of available moves for a certain number of iterations, then finally finishs with a Simulated Annealing phase to intensify the search. This multi-phase metaheuristic would be possible to specify with MLS with absolutely no programming required, in addition to that already done to create the modules for the individual heuristics. Of course, some logical design would still be necessary to declare this order of phases, but it is conceivable that even this logical design can be performed by one heuristic *designing* the other. We introduce this type of idea in Chapter 8.

Similar to other programmatic frameworks, MLS provides an extensible set of classes. For new metaheuristics, only the modules containing the new logic need to be programmed, for example different admissibility conditions or move-types. However, MLS also allows new, sophisticated, metaheuristics to be created *declaratively*, rather than programmatically, by the use of a new markup

language designed to express MLS configurations: MLSML (Modular Local Search Markup Language).

In the words of George Box, "*all models are wrong, but some are useful*". Similarly, there is no "right" framework for local search techniques, each will be designed to support specific activities, and will have strengths and weaknesses. MLS is designed to allow changes to be made to the search process during its execution, and for each component to have a clearly defined role, so that the heuristic itself can swap modules into place. While other frameworks appear to be primarily structures to support and implement existing metaheuristics, MLS is designed to faciliate the creation of *new* metaheuristic paradigms, especially those involving changes to the structure during the search.

1.7 Thesis structure

The thesis is divided into three parts.

Part I – The Arc Subset Routing Problem. This part contains an investigation of a new arc routing problem. The Arc Subset Routing Problem (ASRP) is to find a walk on a weighted graph, such that the sum of the weights of traversed arcs is maximized and the total distance travelled does not exceed a given cost budget. Part I is structured like a traditional Operations Research investigation: Chapter 2 is a review of relevant arc routing and subset routing literature; and Chapter 3 is a preliminary investigation into the ASRP, as a combinatorial optimization problem. The preliminary investigation formulates the problem and develops a number of construction heuristics, and experimentally evaluates these in computational tournaments, along with some improvement procedures in the form of basic versions of Steepest Ascent and Tabu Search. The place of the ASRP in the remainder of the research is as the test problem on which the MLS heuristics are evaluated, and this investigation provides a solid foundation.

Part II – MLS Foundations. This part introduces and defines the Modular Local Search framework. Chapter 4 explains the architecture of MLS, defining all the components and discussing potential applications of these. This is followed by some illustrative examples of how popular metaheuristics would be modelled as MLS, without formally defining the logic for the modules, which occurs later. This chapter finishes with a discussion of the strengths and limitations of MLS. Chapter 5 is a literature review for trajectory-based metaheuristics. The main metaheuristic paradigms are discussed extensively, since these provide the building blocks of MLS. Thoughts are given as to how these metaheuristics could be modelled as MLS techniques.

Part III – Experimentation and Analysis. This part explores the uses of MLS, using the ASRP as a test problem. In Chapter 6 we define a number of MLS heuristics, based on the standard metaheuristic paradigms of Steepest Ascent, Simulated Annealing, Tabu Search, and Variable Neighbourhood Search. We explicitly formulate all the modules required for these heuristics, and these become the foundation of the MLS "toolbox". Extensive computational tournaments provide a large dataset of results and these are modelled against a set of problem characteristics. Several hybrids are developed to demonstrate the hybridization capability of MLS. Chapter 7 contains a side investigation into methods to develop interesting sets of problem instances. A number of heuristics are developed and tested on a large number of problem instances. A brief

study of using MLS to *design* problem instances that exhibit desired properties shows promise. Chapter 8 contains a brief discussion of ways to use MLS to develop advanced metaheuristics. Two examples are demonstrated: using MLS to design other MLS heuristics, as a "meta" strategy, and using the memory structures of MLS to adaptively modify the structure of the heuristic, with elements of *learning*. Chapter 9 concludes with a discussion of directions for future research.

A glossary of terms that have specific definitions in this thesis concerning MLS is given in Appendix A. Appendix B describes the programmatic structure used to implement MLS. Appendix C describes the MLSML markup language used to specify new MLS heuristics. Appendix D is an exploration of some possible extensions that could be made to the ASRP. A bibliography of referenced works concludes.

Part I
The Arc Subset Routing Problem

Overview of Part I
The Arc Subset Routing Problem

Part I introduces the Arc Subset Routing Problem (ASRP), which is to find a walk on a weighted graph, such that the sum of the weights of traversed arcs is maximized and the total distance travelled does not exceed a given cost budget. This part is structured as a traditional Operations Research investigation.

Chapter 2 is a review of relevant arc routing and subset routing literature. Chapter 3 is a preliminary investigation into the ASRP, as a combinatorial optimization problem.

A number of construction heuristics are developed, and experimentally evaluated in computational tournaments, along with some improvement procedures in the form of basic versions of Steepest Ascent and Tabu Search.

Methods for generating and characterising problem instances are introduced.

CHAPTER 2

Arc Routing Literature Review

This chapter surveys the literature on arc routing and on subset selection problems. We introduce vehicle routing problems, explaining the differences between node routing problems, arc routing problems, and general routing problems. We review the field of Arc Routing, summarizing the main problems and their solution techniques, and then continue by cataloging the literature on Subset Routing Problems; most of these have been explored in a node routing context. This chapter concludes with a description of the few Arc Subset Routing Problems in the literature.

2.1 Vehicle routing problems in general

One large subclass of combinatorial problems is that of *vehicle routing problems*. In general, routing problems require the design of a route, from a depot location to several other locations, and then back to the depot. There are endless ways to vary the basic problem, and many of these have been the subject of study; some variations define fields in their own right. The 'medium' for these problems is some sort of network, consisting of a set of nodes, and a set of arcs between these nodes. The broadest distinction is between *node routing problems* and *arc routing problems*. A solution to a node routing problem is a sequence of customers, starting and finishing with the depot. A solution to an arc routing problem is again a sequence of nodes, starting and finishing with the depot, but the focus is on the arcs that are traversed along the way rather than the nodes themselves.

The most basic node routing problem is the **Travelling Salesman Problem** (TSP). In this problem the aim is to visit all the vertices on a graph, while travelling the minimum distance. A solution consists of a sequence of nodes, starting and ending with the depot.

The most basic arc routing problem is the **Chinese Postman Problem** (CPP). In this problem the objective is to traverse all the edges on the graph, while travelling the minimum distance. Since an edge is the link between two nodes, a solution again consists of a sequence of nodes, starting and ending with the depot.

Node routing problems treat the vertices in the graph as the 'customers' to be serviced; whereas arc routing problems treat the edges as the customers to be serviced. The general routing problem is a combination of these, where both edges and vertices are different types of customers to be serviced.

2.2 Arc routing problems

Let us first define some terminology. Following Hertz and Mittaz [141], **Arc Routing Problems** (ARPs) are defined on a connected graph $G = (V, E \cup A)$, where V is the vertex set, E is the edge set, and A is the arc set. For the purposes of this discussion we regard $E \cup A$ to be the links of the graph; an edge is an undirected link, and an arc is a directed link. G is called *undirected* if A is empty, *directed* if E is empty, and *mixed* otherwise. There is a cost matrix $C = (c_{ij})$ associated with $E \cup A$. A *tour* in G is represented by a vector of the form $(v_1, v_2, \ldots, v_n)$ where (v_i, v_{i+1}) belongs to $E \cup A$ for $i = 1, \ldots, n-1$ and $v_n = v_1$. A *covering tour* for G is a tour which traverses all the links in $E \cup A$ at least once. A connected graph, G, is *unicursal* or *Eulerian* if there exists a covering tour such that each link is traversed exactly once. The following, from Ford and Fulkerson [102], is a list of necessary and sufficient conditions for a connected graph to be Eulerian:

- If G is **undirected**, then every vertex must have even degree, where the degree of a vertex is the number of links incident on that vertex.
- If G is **directed**, then the number of links entering and leaving a vertex must be equal.
- If G is **mixed**, then every vertex must be incident upon an even number of directed and undirected links (arcs and edges); moreover, for each subset of vertices S, the difference between the number of directed links out of S and the number of directed links into S must be less than or equal to the number of the number of undirected links incident on S.

A covering tour on an Eulerian graph is called an *Euler tour*. A simple algorithm for finding an Euler tour from an Eulerian graph is given in Algorithm 2.1, as described in Evans and Minieka [87]:

Algorithm 2.1 algorithm CONSTRUCT EULER TOUR FROM EULERIAN GRAPH

Step 1. Begin at any vertex s and construct a cycle C. This can be done by traversing any link (s, x) incident on vertex s and marking this link "used". Next, traverse any unused link incident on vertex x. Repeat this process of traversing unused edges until returning to vertex s. (This process must return to vertex s since every vertex has even degree and every visit to a vertex leaves an even number of unused links incident on that vertex. Hence, every time a vertex is entered, there is an unused link for departing from that vertex.)

Step 2. If C contains all the edges of G, stop. If not, then the subgraph G' in which all links of C are removed must be Eulerian since each vertex of C must have an even number of incident links. Since G is connected there must be at least one vertex v in common with C.

Step 3. Starting at v, construct a cycle in G', say C'.

Step 4. Splice together the cycles C and C', calling the combined cycle C. Return to Step 2.

end

2.2.1 The Chinese Postman Problem

The foundational ARP is the **Chinese Postman Problem** (CPP), so called since it was proposed by the Chinese mathematician M. Guan [131]; following Guan's original application of a postman designing his postal route, ARPs are known as *postman problems*. The CPP consists of finding a minimum-cost covering tour on a connected graph, G. The problem was shown by Edmonds and Johnson [78] to be solvable in polynomial time for both the case where all links are undirected (edges) and the case where all the links are directed (arcs). When G is Eulerian, the Euler tour passing through each link exactly once is the optimum. When G is not Eulerian, the problem becomes that of augmenting G, with copies of existing links, such that it becomes Eulerian; when this augmentation is done with minimum cost the optimum is the associated Euler tour. The augmentation is performed by adding edges to match odd degree vertices, i.e. by finding a minimum-cost perfect matching (see Edmonds and Johnson [78] and Christfides [47]).

The CPP on a directed graph can also be solved in polynomial time by solving a minimum cost flow problem where the flow on each arc has to be at least 1, see Orloff [202]. However, there is a further condition on G; in addition to being connected, G must be *strongly connected* (there must be a path between every pair of nodes). A least-cost Eulerian graph can be constructed by solving a transportation problem; details are provided by Edmonds and Johnson [78] and Orloff [202], and Beltrami and Bodin give an example [21]. The method given above (Algorithm 2.1) for finding an

Euler tour can then be used to find the optimal tour, with the addition that the arcs selected when leaving a vertex must be directed out of that vertex.

For a mixed graph, G, we define some additional terminology, from Eiselt et al. [84]. A graph is *even* if the total number of links (both arcs and edges) incident to each of its vertices is even; it is *symmetric* if for each vertex the number of incoming arcs is equal to the number of outgoing arcs; a graph is *balanced* if the balanced set condition is satisfied. The balanced set condition (as defined in Nobert and Picard [198]) says that a graph is balanced if, given any subset S of vertices, the difference between the number of arcs directed from S to $V \setminus S$ and the number of arcs directed from $V \setminus S$ to S is no greater than the number of (undirected) edges joining S and $V \setminus S$. The conditions for the unicursality of a mixed graph are therefore that the graph be both even and balanced. Note that if G is even and symmetric, then it is also balanced, and hence Eulerian.

If G is Eulerian then the mixed CPP may be solved by using the above method for finding an Euler tour. If G is even, but not balanced, then Edmonds and Johnson [78] show that the mixed CPP may still be solved in polynomial time, for example by using the procedure described in Coberán et al. [61].

If G is not even, then the mixed CPP is *NP*-hard, as shown by Papadimitriou [203]. The aim is to find a minimum cost augmentation of G by replicating a sufficient number of the edges and arcs of G so that the resulting graph is Eulerian. This is not always possible because not all graphs can be made Eulerian (Figure 2.1), so in that case, the mixed CPP is infeasible.

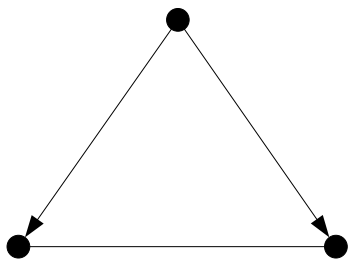


Figure 2.1: Example of a graph that cannot be made Eulerian

Edmonds and Johnson [78] suggest two heuristics, MIXED1 and MIXED2, which have been improved by Frederickson [103], Christofides et al. [48], Raghavachari and Veerasamy [216] and Pearn and Chou [205]. Cutting plane algorithms have been developed by Grötschel and Win [130], Nobert and Picard [198], and by Corberán et al. [62]. Pearn and Liu [208] also present several strong algorithms. Polyhedral results directly relating to the mixed CPP were reported by Eglese and Letchford [81]. Corberán et al. [61] have developed a GRASP heuristic for this problem.

Clossey et al. [54] study a version of the CPP with turn penalties. Prior methods solved this problem using a transformation into an equivalent node routing problem, but the authors present a direct method.

A variant known as the **Windy Postman Problem** (WPP) occurs when the cost matrix is asymmetric. The undirected CPP is a special case of the WPP, with $c_{ij} = c_{ji}$. Moreover, since an arc $a = (i, j)$ with

cost c_a can be transformed to an edge with costs $c_{ij} = c_a$ and $c_{ji} = \infty$, the directed and mixed problems can also be viewed as special cases of the WPP [250].

Brucker [34] and Guan [132] have shown that the WPP is *NP*-hard, but it is solvable in polynomial time if G is Eulerian [250]. The polyhedral structure of the WPP has been studied by Win [249,250] and by Grötschel and Win [130], who devised a cutting plane algorithm.

Benavent et al. [23] present several heuristics and a Scatter Search metaheuristic for the WPP, and give extensive computational results.

2.2.2 The Rural Postman Problem

The **Rural Postman Problem** (RPP) is a generalization of the CPP where a given subset $R \subseteq (E \cup A)$ of edges and arcs are said to be *required*. A *covering tour* for R is a tour that traverses all the edges and arcs of R at least once. Both the undirected and directed versions of the RPP are *NP*-hard (see Lenstra and Rinnooy Kan [169] and Garfinkel and Webb [106]), except for the special case where $R = E \cup A$, which is the CPP.

Eiselt et al. [85] describe applications for the RPP and its variants as including street sweeping [29,30,83], snow ploughing (often as a hierarchical RPP) [3,58,82,137,168,238], garbage collection [4,21,28,51,52,107,183,230,239,253], mail delivery [172,173,224], and meter reading [229,252].

Mathematical programming formulations of the RPP have been proposed in Christofides et al. [49], Corberán and Sanchis [63], and Ghiani and Laporte [113]. Early computational results were achieved by Ghiani and Laporte [113] (branch-and-cut), Corberán et al. [60] (cutting planes for the general routing polyhedron), Frederickson [103], Pearn and Wu [206], and Hertz et al. [139] (heuristics). A Memetic Algorithm was proposed by Rodrigues and Ferreira [221]. A fundamentally different formulation was introduced by Garfinkel and Webb [106], then modified and tested computationally by Fernández et al. [94]. Letchford [171], and Corberán and Sanchis [64], introduce new inequalities for the General Routing Problem, which are also valid for the RPP.

The directed RPP was examined in detail by Christofides et al. [50]. Benavent and Soler [24] introduce a generalization of the directed RPP with turn penalties, and transform it into an asymmetric TSP, which allows its solution with existing methods. An analogue of the hierarchical postman version of the CPP was studied by Dror and Langevin [74], who solved the *clustered* version of the directed RPP by transforming it into a generalized travelling salesman problem. Cabral et al. [41] also studied a transformation of the Hierarchical CPP into the RPP. Laporte [163] studied solving several classes of arc routing problems, including the Mixed Rural Postman Problem, by transforming them into travelling salesman problems.

Letchford [170] investigates the RPP with *deadline classes*, which are similar to time-windows, and gives a cutting plane algorithm. Ghiani et al. [115] consider the Periodic RPP, where each required arc/edge of a graph must be visited a given number of times over an m-day planning period in such a way that service days are equally spaced.

2.2.3 Capacitated Arc Routing Problems

In real-world applications there are often restrictions on the time or resources available to accomplish the touring. The theory has incorporated some of these restrictions in the form of *capacitated* problems. The most common capacitation is for the delivery vehicle to have a capacity, either on the distance it can travel before returning to the depot, or on the amount of demand it can supply before returning to the depot. In these cases it is necessary either for the vehicle to make multiple trips, or for multiple vehicles to work together. If there are no time constraints, then multiple trips and multiple vehicles are equivalent. The capacitated node-routing equivalent is the Vehicle Routing Problem.

The **Capacitated Chinese Postman Problem** (CCPP) was introduced by Christofides [47]. The CCPP may be stated as follows: given a connected graph, $G = (V, E)$ in which each edge (v_i, v_j) has an associated cost c_{ij} and positive demand d_{ij}, and a set of vehicles having fixed capacity W, find a minimum-cost set of tours, each starting and finishing at the depot, such that the total demand on each cycle does not exceed the vehicle capacity W.

The CCPP introduces the distinction between *service* and *traversal*; the example below (Figure 2.2), taken from Golden et al. [123], illustrates this point. Vertex 1 is the depot, $W = 4$, and edges are labelled by ordered pairs (c_{ij}, d_{ij}). One feasible solution is the set of tours

$$\underline{1\ 2\ 4}\ 3\ 1$$
$$\underline{1\ 4\ 3\ 1}$$

(underlines indicate the servicing of edge demands) with a total distance of 15 units. Note that this is not necessarily the optimum solution, but illustrates that edges can be either serviced or traversed.

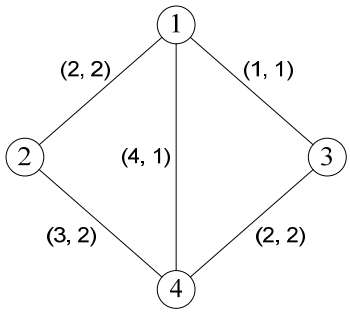


Figure 2.2: Example illustrating the difference between *service* and *traversal*

Golden and Wong [127] have shown that the CCPP is *NP*-hard. They also proved that the easier 0.5-approximate CCPP is *NP*-hard; the 0.5-approximate CCPP is to find a CCPP solution whose cost is less than 1.5 times the optimal solution.

Christofides [47] gave a construct-and-strike heuristic and calculated a lower bound. This lower bound was corrected by Golden and Wong [127], who also presented a formulation and suggested an augment-merge heuristic. Golden et al. [123] present an augment-merge heuristic and a path-scanning heuristic,

and give computational results for all three heuristics. Assad et al. [10] present a node-scanning lower bound.

The **Capacitated Arc Routing Problem** (CARP) is a generalization of the CCPP and the RPP. The only difference with the CCPP is that instead of being positive, the demands are non-negative, so that some of them can be zero. The problem becomes the capacitated analogue of the RPP, where only those edges with positive demand *require* service.

Because the CCPP is a special case of the CARP, most of the work done for the CCPP is applicable also to the CARP; the CARP and the 0.5-approximate CARP are also *NP*-hard.

We paraphrase Amberg et al. [5] to note that capacitated arc routing problems consist of two interdependent subproblems: the *assignment problem* which forms subsets or clusters of required edges served by the same vehicle and the *sequencing* or *routing problem* which determines the sequence of serving the edges.

There are several so-called "parallel" heuristics which simultaneously consider these subproblems: the path-scanning algorithm of [123] and the construct-and-strike heuristic of Christofides [47], both modified by Pearn [207]; the parallel-insert method of Chapleau et al. [45]; the augment-insert algorithm of Pearn [204]; and the augment-and-merge procedure of Golden and Wong [127], which uses the savings criterion of Clarke and Wright [53].

Sequential algorithms either choose the assignment before computing the routes within the formed subsets (cluster-first-route-second, CFRS) or construct a giant route that is broken into small routes (route-first-cluster-second, RFCS). CFRS methods are more suitable for node routing problems, but the greedy criterion of Win [249], or a generalized assignment algorithm, as in Benavent et al. [22], can be used to partition the edges, then a route is found for each cluster by a simple modification of a CPP algorithm (see Eiselt et al. [85]). The RFCS algorithm of Stern and Dror [229] designs a giant route as for the CPP and cuts it into small paths. The heuristic of Liebman and Male [183] splits the giant route into cycles and uses savings criteria for the recombination.

Hertz et al. [139] present a set of construction and post-optimization procedures for the RPP. These are borrowed and added to by Hertz et al. [140] for use on the CARP, and then combined into a new Tabu Search algorithm, CARPET; the procedures are SHORTEN, DROP, ADD, PASTE, CUT, SWITCH, and POSTOPT. Hertz and Mittaz [142] use these procedures in a variable neighbourhood descent algorithm. Amberg et al. [5] uses a Tabu Search implementation on the CARP with multiple depots.

Many metaheuristics have been applied to capacitated arc routing problems. Beullens et al. [25] develop a Guided Local Search heuristic for the CARP with some new moves derived from a node routing context. Greistorfer [128] presents a Tabu Scatter Search heuristic for the CCPP.

Ghiani and Improta [114] use a transformation *into* the CARP to solve the Generalized Vehicle Routing Problem, which at the time constituted the only known exact method of solution.

Mourão and Almeida [197] solve a refuse collection problem in Lisbon by modelling as a CARP with side constraints, and present lower bounds and a three-phase heuristic.

2.2.4 Advanced arc routing problems

A number of more complicated variations on arc routing problems have been studied, often arising as a result of a real-world problem.

Li and Fu [174] describe a case study of a school bus routing problem, formulated as a multi-objective problem to minimize the total number of buses required, the total travel time spent by pupils, and the total bus travel time.

Rosa et al. [223] study an extension they call the Arc Routing and Scheduling Problem with Transhipment, which has applications in garbage collection, where the demand is collected and taken to a transfer station by one set of vehicles, and then transferred to a dump site by another set of vehicles. They give a lower bound based on a relaxation, and computational results with a Tabu Search heuristic.

Fleury et al. [101] consider a *stochastic* CARP, where demands are random, rather than known deterministically, which is commonly the case in real-world applications. The goal for this problem is to create routes that are insensitive to changes in demand. They develop a Genetic Algorithm approach.

2.3 Subset routing problems

Subset routing problems (SRPs) can be modelled as reward collection problems; the objective is to collect the maximum reward, within some constraints on the route. This varies from simple routing problems where all customers require service and thus only their location is important. When it is not possible for all customers to be serviced, due to the constraints, a subset of them is serviced. Subset routing is a sub-class of *vehicle routing problems* (VRPs; see [164] for a description VRPs and their heuristics); a VRP solution consists of a number of routes, each of which satisfies the demand of a subset of the customer set. A SRP route can be thought of as a single VRP route that satisfies the maximum demand.

We divide subset routing problems into *Node Subset Routing Problems* (NSRPs), *Arc Subset Routing Problems* (ASRPs), and *Mixed Subset Routing Problems* (MSRPs). SRPs require finding a route on a graph, which visits a subset of vertices, and traverses a subset of edges. In a NSRP the reward is distributed on the vertices of the graph, in an ASRP it is distributed on the edges, and in a MSRP, the reward is distributed on both the vertices and the edges of the graph.

2.3.1 Node Subset Routing Problems

Most of the SRPs in the literature can be classified as NSRPs. Feillet et al. [91] describe a class of problems that they call **Traveling Salesman Problems with Profits**, define several generic problems of this class, and survey the literature on these types of problems. In these problems reward is collected for customers serviced, and cost is incurred during the travel, and the goal is to maximize the **profit** (reward collected minus cost incurred). They note that TSPs with profits may be viewed as bi-objective problems — to maximize profit and to minimize cost — and hence a valid solution technique could be to find the Pareto frontier (the set of feasible solutions such that neither objective can be improved without deteriorating the other). However researchers generally address their single-criterion versions,

which are either combining the objectives linearly, or constraining one of the objectives. The three generic problems they define are (1) to maximize a linear profit function, (2) to maximize profit while incurrable cost is constrained, and (3) to minimize cost while meeting a minimum profit threshold. The various problems described in the following sections generally fall into these categories (although there are always variations and exceptions).

2.3.1.1 Unconstrained NSRP

Unconstrained NSRPs relax the requirement that every vertex be visited at least once. Instead of being *forced* to service a subset of the vertices, they make it *desirable* to do so.

In the **Travelling Salesman Subset-tour Problem** (TSSP), studied by Mittenthal and Noon [193], a salesman collects a **reward**, r_j, in every city v_j that he visits, and incurs a fixed **penalty**, π_k, for every city v_k that he doesn't visit. There is a fixed cost c_{ij} to travel between cities v_i and v_j. The objective is to maximize the **total net benefit**; i.e., the sum of the rewards collected minus the sum of the penalties incurred. The salesman must determine both the subset of cities to visit, and also the order in which they are visited; these decisions are interdependent.

Beasley and Nascimento [20] consider a generalization of the TSSP where some reward may be gained by allocating customers not directly visited by the salesman to a nearby customer who is visited. They formulate the **Single Vehicle Routing Allocation Problem** (SVRAP), which serves as a framework for various other problems, such as the **Covering Tour Problem** (CTP) of Gendreau et al. [108], the **Shortest Covering Path Problem** (SCPP) of Current et al. [65], and the **Covering Salesman Problem** (CSP), **Median Tour Problem** (MTP), and **Maximal Covering Tour Problem** (MCTP), of Current and Schilling [66,67]. In the SVRAP, three types of selection occur: on-route customers, who contribute a fixed reward; off-route (allocated) customers, who contribute a reward dependent upon the on-route customers they are allocated to; and isolated customers, who contribute a fixed penalty cost. Customers may have a predetermined type.

Keller and Goodchild [159] and Keller [158] propose the **Multiobjective Vending Problem** (MVP), where the objective is to simultaneously minimize the total cost incurred and maximize the total reward collected. They attempt to find a non-inferior solution set, which is the set of all solutions such that no other solution has *both* a greater reward and a lower cost.

In another unconstrained NSRP, the objective is to minimize the sum of all the travel costs and non-inclusion penalties, without considering reward. This variation was studied by Bienstock et al. [26], Williamson [248], and Goemans and Williamson [121]. Volgenant and Jonker [244] had earlier considered the same problem, naming it the **Generalized Travelling Salesman Problem** (GTSP), and also a special case called the **Shortest Path Problem with Specified Nodes** (SPPSN) in which a specified set of vertices must be visited exactly once, remaining vertices at most once, and large penalties are incurred for not visiting the specified vertices. Note that there are many problems called the Generalized Travelling Salesman Problem in the literature, each is a generalization of the standard TSP.

Malandraki and Daskin [181] introduce the **Maximum Benefit Travelling Salesman Problem** (MBTSP), in which the objective is to maximize the net benefit (reward collected minus cost incurred). They have the additional feature of being able to collect additional reward on second and subsequent visits to a vertex. This reward diminishes with each visit.

2.3.1.2 Reward constrained NSRP

A **reward constrained NSRP** is an NSRP constrained by a single constraint on the sum of reward values collected during the subtour.

The **Prize Collecting Travelling Salesman Problem** (PCTSP) features a salesman who wishes to minimize the sum of his travel costs and penalty (isolation) costs during a subtour that starts and finishes at a depot, whilst visiting enough cities to collect a prescribed minimum amount of reward. This problem was studied by Awerbuch et al. [11], Balas [15,16], Dell'Amico et al. [69], and Fischetti and Toth [122].

Hamacher and Moll [133] consider a special case of the PCTSP in which all the rewards are equal and there is no pre-specified depot vertex. They name this version the **Travelling Salesman Selection Problem**, which we will label (Hamacher-TSSP). Heuristics are based on approximations for the ***k*-Minimal Spanning Tree Problem**, to find the vertex cluster containing the shortest subtour satisfying the requirement to visit the given number of cities.

2.3.1.3 Cost constrained NSRP

A **cost constrained NSRP** is a NSRP constrained by a single constraint on the cost or length of the subtour.

Gensch [112] studies a problem in which the salesman wishes to maximize the net profit (reward collected minus cost incurred) whilst not exceeding a prescribed cost budget. Gensch names this the **Travelling Salesman's Subtour Problem**, which we label (Gensh-TSSP).

The most prevalent SRP studied in the literature is the so-called **Orienteering Problem** (OP), in which the salesman wishes to maximize the reward collected without exceeding a prescribed cost budget. The motivation for this problem is the sport of orienteering, concisely described by Chao et al. [43]:

> "Orienteering is an outdoor sport usually played in a mountainous or heavily forested area. Armed with a compass and map, competitors start at a specified control point, try to visit as many other control points as possible within a prescribed time limit, and return to a specified control point. Each control point has an associated score, so that the objective of orienteering is to maximize the total score. Competitors who arrive at the finish point after time has expired are disqualified, and the eligible competitor with the highest score is declared the winner. Since time is limited, competitors may not be able to visit all the control points. The competitors have to select a subset of control points to visit that will maximize their total score subject to the time restriction."

Hayes and Norman [138] model a real world orienteering event, the 1974 Lake District Mountain Trail, in England, as a Dynamic Program, to compare optimal paths against the actual routes selected by participants. However, they do not formulate a combinatorial optimization problem.

Tsiligirides [236] appears to be the first to consider the CO problem now known as the OP, although calling it the **Generalized Travelling Salesman Problem**. Tsiligirides created the first test instances for this problem, and these have subsequently become the standard benchmark instances for the OP. Golden et al. [124] coined the name *Orienteering Problem*, and compared a number of stochastic and deterministic subtour construction and improvement heuristics with those proposed by Tsiligirides. Golden et al. [125] improved these heuristic ideas with a multifaceted heuristic including center-of-gravity improvement, randomness, subgravity and a learning capability. Keller [158] adds a heuristic for the MVP (but restricted to the OP) to these computational comparisons. Ramesh and Brown [217] propose another heuristic for the OP employing local subtour operations including insertions, deletions and improvements; in their version, which they call the **Generalized Orienteering Problem** (GOP), the start and finish vertices may be different, but are still specified. Wang et al. [246] modify a neural network to find an initial feasible solution which is then improved using 2-exchanges and cheapest insertion. A local search heuristic was developed by Chao et al. [43], which outperformed most other heuristics on a set of 107 test problems. Sokkappa [227] also studied the OP, but called it the **Cost Constrained Travelling Salesman Problem** (CCTSP), as did Awerbuch et al. [11], calling the OP the **Bank Robber Problem**. Awerbuch et al. were also able to provide a poly-logarithmic performance guarantee for the OP and the PCTSP.

A number of algorithms have been proposed to find optimal solutions to the OP. Kataoka and Morito [156] introduced an equivalent problem, the **Maximum Collection Problem** (MCP), and proposed a branch-and-bound algorithm with an **Assignment Problem** (AP) relaxation. Laporte and Martello [162] provide an integer linear programming formulation of the **Selective Travelling Salesman Problem** (STSP), also equivalent to the OP, and suggest simple greedy heuristics, upper and lower bounding techniques and a branch-and-bound algorithm. Ramesh et al. [218] also propose a branch-and-bound algorithm, using a Lagrangean relaxation solved by a degree-constrained spanning tree procedure. Leifer and Rosenwein [167] add several strong linear programming relaxations for the OP by adding a sequence of valid inequalities; the solution of three successive linear programs provide upper bounds on the optimum. Gendreau et al. [109] and Fischetti et al. [98] give optimal branch-and-cut algorithms for the OP, Gendreau et al. for a version with the addition that a specified subset of the customers *must* be in the solution.

Arkin et al. [8] describe a version of the OP on networks. Their formulation requires rewards to be integer, as the objective was to maximize the number of customers visited, and therefore reward corresponds to repeated customers.

Diaby and Ramesh [71] consider a capacitated NSRP, called the **Distribution Problem with Carrier Service** (DPCS). Each customer has a given demand, the vehicle has a carrying capacity, and the entire operation must be completed within a certain time. An outside carrier is available for direct service of customers from the depot. The problem is to determine a feasible tour for the company vehicle and the customers to be serviced by the outside carrier such that the total cost of the operation is minimized.

Key features of this problem are feasibility with respect to vehicle load as well as the travel time constraint, penalty costs for not visiting a customer and no rewards.

Millar [191] and Millar and Kiragu [192] describe an application of the OP to a fisheries patrol problem in the Scotia-Fundy region of the Atlantic coast of Canada. Here the OP serves as a static snapshot of a more dynamic problem; the reward values are used to approximate *urgency* and *importance* criteria.

2.3.1.4 Multiple vehicle NSRP

In the **Multiple Travelling Salesman Problem** (MTSP), *m* salesmen start from a depot, each visiting a number of vertices and returning to the depot, such that every vertex is visited by at least one salesman and the sum of the distances travelled is minimized. The **Vehicle Routing Problem** (VRP) is simply a capacitated MTSP. Similarly, multiple versions of NSRPs have been defined.

Chao et al. [44] formulate the **Team Orienteering Problem** (TOP), where *m* team members cooperate to maximize the sum of reward collected by the team, subject to a common time limit. There are three interdependent decisions: which vertices to visit, to which team member each vertex should be allocated, and for each team member the sequence of in which to visit the vertices. This paper suggests modified versions of the local search heuristic of Chao et al. [43] and the stochastic heuristic of Tsiligirides [236], both originally for the OP.

Butt and Cavalier [38] and Butt and Ryan [37,39] consider an equivalent problem, the **Multiple Tour Maximum Collection Problem** (MTMCP). Butt and Cavalier provide an integer programming formulation and propose a local search heuristic, and Butt and Ryan suggest an optimal branch-and-bound algorithm.

Johnston [152] devotes a PhD thesis to an extensive investigation of a *competitive* reward collection problem, which he calls the **Competition Routing Problem** (CRP). In this problem there are multiple competing vehicles which seek to maximize their own reward, while minimizing their opponents. He focuses on the comparative performances of a wide range of *strategies* which incorporate some level of response to an opponent's actions.

2.3.1.5 Time dependent NSRP

Three forms of **Time Dependent NSRP** have been investigated in the literature: time dependent rewards, time dependent costs, and time windows.

Brideau and Cavalier [33] and Erkut and Zhang [86] look at the **Maximum Collection Problem with Time Dependent Rewards** (MCPTDR). They include reward values which decay to zero as a linear function of time. Malandraki and Dial [182] and Malandraki and Daskin [180] also consider time dependent costs.

Problems which specify when customers may be visited are known collectively as *Time Window Problems*. These problems specify for each customer an early time (before which they cannot be visited) and a late time (after which they cannot be visited). Kantor and Rosenwein [154] introduce the **Orienteering Problem with Time Windows** (OPTW) in which three interrelated decisions are

required: selection of customers to be serviced, sequencing of customers within the solution route, and scheduling of customer deliveries with respect to their time windows.

In his PhD thesis, Beale [19] considers the **Maximum Collection Problem** (MCP), in which he includes multiple vehicles, time dependent rewards, non-zero service times, penalties for non-service, and the completion of specified tasks associated with the service of each customer, which possibly include pickups and deliveries.

2.3.2 Arc subset routing problems

Golden et al. [126] briefly define the **Time-Constrained Travelling Salesman Problem** (TCTSP). This problem shares the features of an NSRP and an ASRP; all vertices are adjacent, as in node routing problems, but reward is distributed on edges, as in an arc routing problem. There is a reward associated with each edge and the salesman wishes to maximize the total reward over the edges traversed whilst not exceeding the prescribed cost budget. In keeping with the TSP formulation, though, all vertices are adjacent to all others; as with node routing problems any vertex may be followed by any other in a subtour. They suggest an iterative heuristic procedure, which we describe in Algorithm 2.2.

Let r_{ij} and c_{ij} respectively be the reward and cost associated with the traversal of edge (v_i, v_j). The cost budget is C; let the depot be vertex v_0. At each iteration k, let P and T denote the total reward and total cost of the subtour just generated. Also, ΔP and ΔT are the changes in total reward and cost associated with each permissible insertion.

Algorithm 2.2 heuristic TSTSP

Step 0. Set k, P and T to 0. Initialize parameter R_0. Choose parameter α.

Step 1. **for** $i = 1$ **to** $|V|$ **do**

$\Delta T \leftarrow c_{0i} + c_{i0}$

$\Delta P - R_0 \Delta T \leftarrow (r_{0i} + r_{i0}) - R_0 (c_{0i} + c_{i0})$

end

Find i^* such that $\Delta T \leq C$ and $\Delta P - R_0 \Delta T$ is maximized.

if no such point exists **then**

STOP

else

Record the subtour $(0, i^*, 0)$

$P \leftarrow P + \Delta P$

$T \leftarrow T + \Delta T$

end

Step 2. $k \leftarrow k + 1$

$R_k \leftarrow \alpha(P/T) + (1-\alpha) R_{k-1}$

Step 3. **for** each vertex v_k in the present subtour, and for each pair of vertices (v_i, v_j) which are adjacent in the subtour **do**

$\Delta T \leftarrow c_{ik} + c_{kj} - c_{ij}$

$\Delta P - R_k \Delta T \leftarrow (r_{ik} + r_{kj} - r_{ij}) - R_k (c_{ik} + c_{kj} - c_{ij})$

end

Find the triple i^*, j^*, k^* such that $T + \Delta T \leq C$ and $\Delta P - R_k \Delta T$ is maximized.

if no such triple exists **then**

goto *Step 4.*

else

Insert k^* between i^* and j^* in the subtour and record the subtour.

$P \leftarrow P + \Delta P$

$T \leftarrow T + \Delta T$

end

goto *Step 2.*

Step 4. From all the subtours recorded, select the one with the largest P.

end

Golden et al. [126] clarify several points about this heuristic as follows. The ratio P/T is the worth of a unit of cost in the current subtour, R_k is the best estimate of the worth of a unit of cost at iteration k; R_k is an estimate which takes into account all previous ratios P/T but weights the more recent ones more

heavily. R_0 should represent an educated guess (possibly based on preliminary analysis) of the reward to cost ratio for the optimal subtour. The parameter α should be chosen between 0 and 1.

Malandraki and Daskin [181] also introduce an ASRP, the **Maximum Benefit Chinese Postman Problem** (MBCPP). This problem is an **Unconstrained ASRP**, where the objective is to maximize the net profit (reward collected minus cost incurred). An additional feature is that an edge may be serviced more than once, with a diminishing reward for each traversal.Feillet et al. [90] introduce the Profitable Arc Tour Problem and give a branch-and-price algorithm for its solution. In this problem there is reward associated with each arc, and this reward may be collected up to a specified number of times. The objective is to find a set of cycles in the graph that maximize the profit, reward collected minus cost incurred, subject to a maximum length of cycles.

Archetti et al. [7] consider the Undirect Capacitated Arc Routing Problem with Profits, where a subset of edges of a graph have a profit and a demand, and a fleet of capacitated vehicles service them attempting to maximize profit. It is unclear form their description exactly why the demand is included, since it does not feature in their formulation.

Johnston and Chukova [151] briefly consider a version of the Arc Subset Routing Problem, which they call the Rural Postman Problem with Rewards, where the goal is to maximize collected reward subject to a cost constraint. They consider several swap operators in the context of a local search heuristic. Their swap operators are suited to the RPP, since they treat the *required* edges as nodes in a graph, and simply route between their end points with shortest paths. This idea neatly sidesteps the difficulty of using swap operators on arc routing problems.

Compared to the node-routing literature, there are still relatively few Arc Subset Routing Problems. Those that have been studied tend to attempt to maximize *profit*, rather than constraining cost. Two techniques that have been used by others that are potential building blocks are the profit-to-cost ratio used by Golden et al. [126] to identify the best "bang-for-buck" next move, and the concept of using swap operators for arc routing problems of Johnston and Chukova [151], which could be extended from *required* arcs to any arbitrarily-decided *valuable* arcs.

Coda

▼ Summary

In this chapter we have summarized the literature on arc routing problems, subset routing problems and the intersection of these: arc subset routing problems.

▼ Link

In the next chapter we formulate the Arc Subset Routing Problem that will be used for experimentation throughout the thesis, and perform a preliminary investigation as a traditional Operations Research probem; we develop several construction heuristics and test these in computational tournaments.

CHAPTER 3

Preliminary Investigation of the ASRP

The purpose of this chapter is to examine the Arc Subset Routing Problem in a traditional Operations Research investigation. We start with an exploration of mathematical programming formulations, and then develop a number of construction heuristics and test these extensively on grid graphs, along with some local search improvement routines. This chapter provides the only consideration of construction routines for the ASRP; the remainder of the thesis is concerned with local search approaches, where the initial solution is basically provided by a black box procedure. A key element is the development of a set of move-types for the ASRP, which are used throughout the rest of the thesis. We consider some elements of ASRP problem instance design, and develop two methods for problem generation, along with a number of metrics to characterize these problem instances.

3.1 Formulation of the ASRP

It is useful to have an accurate representation of the Arc Subset Routing Problem (ASRP) as an integer program. Having a mathematical programming formulation is essential to both designing problem instances with known optima and to the design of exact algorithms, and also formally defines the problem in an unambiguous way. There are several ways of formulating the ASRP, but they all have several constraint sets in common.

We define the ASRP as follows. Let $G(V, E)$ be an undirected graph, where V is the vertex set, E is the edge set, c_e (≥ 0) and r_e (≥ 0) are respectively the cost and reward associated with traversing edge $e \in E$ where the cost is incurred for each traversal and the reward is collected only on the first traversal. Further, let C be the maximum incurable cost: the cost *budget*. The ASRP is to determine a greatest-reward subset of edges and associated traversal frequencies that make up an Eulerian and connected subgraph incident on a given depot vertex. In other words, we want to find a subset of edges that are connected, and the number of times each of those edges is traversed. This is equivalent to finding a closed tour through a subset of the edges.

Any valid formulation needs to ensure the following:

- The solution results in a tour (each vertex is left the same number of times it is entered). This is accomplished by ensuring that every vertex has even degree.
- The graph contains no subtours (it is connected).
- The depot is incident on an included arc (it has positive degree).
- The cost budget is not exceeded.

In Section 3.1.1, we first give some definitions that will be utilized throughout. Then, in Section 3.1.2, we consider the various ways in which the restrictions above may be represented as constraints. Finally, in Sections 3.1.3-3.1.5 we present three alternative formulations, which use various ways of defining the traversal variables.

3.1.1 Definitions

Let $V^* \subseteq V$ be a subset of vertices and $E^* \subseteq E$ be a subset of edges. We define the *incidence set* $\delta(V^*)$ (often known as the *edge cutset*) of V^* to be the set of edges which are incident (touching) on exactly one of the vertices in V^*, so they have one end-point in V^* and one end-point in $V \setminus V^*$. Similarly, the incidence set $\delta(E^*)$ of E^* we define to be the set of vertices which are incident on exactly one of the edges in E^*. If $E^* = \{e\}$, then we write $\delta(e)$, not $\delta(\{e\})$, to denote the set of vertices incident on edge e. Similarly, we write $\delta(v)$ to denote the set of arcs incident on vertex v.

Note that edges are incident on vertices and vertices are incident on edges. We use the term *adjacent* to refer to edges that are incident on the same vertex, and to vertices that are incident on the same edge. We define the *adjacency* $\theta(V^*)$ of V^* to be the set of vertices which are incident on the incidence set of V^*, but are not themselves in V^*; the adjacency of a vertex set is all the vertices one arc away from the set. Similarly, the adjacency $\theta(E^*)$ of E^* is the set of edges which share one vertex with E^*.

Definitions.

The *total incidence* Δ of an edge (vertex) set is all the vertices (edges) which are incident on the individual edges (vertices).

The *joint incidence* $\hat{\partial}$ of an edge (vertex) set is all the vertices (edges) which are incident on all of the elements of the edge (vertex) set.

The *exclusive incidence* $\overline{\partial}$ of an edge (vertex) set is all the vertices (edges) which are only incident upon edges (vertices) of the set.

Similarly, the *total adjacency* Ξ of an edge (vertex) set is all the edges (vertices) which are adjacent to the individual edges (vertices) of the set.

The *joint adjacency* $\hat{\theta}$ of an edge (vertex) set is all the edges (vertices) which are adjacent to all the elements of the set.

The *exclusive adjacency* $\overline{\theta}$ of an edge (vertex) set is all the edges (vertices) which are only adjacent to elements of the set.

The formulae of these sets is given in Table 3.1.

Table 3.1: Definition of incidence and adjacency sets

	Vertex set V^*	**Edge set E^***
Total incidence	$\Delta(V^*) = \bigcup_{v \in V^*} \delta(v)$	$\Delta(E^*) = \bigcup_{e \in E^*} \delta(e)$
Joint incidence	$\hat{\delta}(V^*) = \bigcap_{v \in V^*} \delta(v)$	$\hat{\delta}(E^*) = \bigcap_{e \in E^*} \delta(e)$
Exclusive incidence	$\overline{\delta}(V^*) = \Delta(V^*) \setminus \delta(V^*)$	$\overline{\delta}(E^*) = \Delta(E^*) \setminus \delta(E^*)$
Total adjacency	$\Xi(V^*) = \bigcup_{v \in V^*} \theta(v)$	$\Xi(E^*) = \bigcup_{e \in E^*} \theta(e)$
Joint adjacency	$\hat{\theta}(V^*) = \bigcap_{v \in V^*} \theta(v)$	$\hat{\theta}(E^*) = \bigcap_{e \in E^*} \theta(e)$
Exclusive adjacency	$\overline{\theta}(V^*) = \Xi(V^*) \setminus \theta(V^*)$	$\overline{\theta}(E^*) = \Xi(E^*) \setminus \theta(E^*)$

The only other relationship worth mentioning is that the adjacency of an edge (vertex) set is the same as the incidence of the total incidence of the set.

$$\theta(V^*) = \delta(\Delta(V^*)) \qquad \theta(E^*) = \delta(\Delta(E^*))$$

Definition. A 0/1/2 edge is an edge whose associated traversal frequency variable can be equal to either 0, 1 or 2 in an optimal solution. Similarly, a 0/2 edge is an edge whose traversal frequency variable can be equal to either 0 or 2. The remaining edges are said to be 0/1. Furthermore, the sets of all these edges will be denoted as E_{012}, E_{02} and E_{01}, respectively.

Definition. We define edge b to be a *bridge* if its removal disconnects the graph. We define edge p to be a *pendant edge* if it is incident on a vertex of order 1. All pendant edges are also bridges. We let B be the set of edges which are bridges, and $P \subseteq B$ be the set of pendant edges.

3.1.2 Considering the constraints

In the following sections we need two types of variables. For each edge, we need a variable that tells us whether an edge has been included, and a variable that tells us how many times it has been included. In the later formulations, we define several ways of doing this, but for the purposes of this section we assume that for each edge e we have a variable x_e, which we set to 1 if the edge is included and 0 if it is not, and a variable s_e, which we set equal to the number of times e is included (integer).

3.1.2.1 Objective function

We simply take the sum of the rewards for included edges.

$$\sum_{e \in E} r_e x_e$$

3.1.2.2 Even degree constraint

We need to ensure that every vertex has even degree, as a requirement of an Euler graph. The following constraint ensures this by counting the number of times that the edges incident on the vertex are traversed, and ensuring that the sum is even.

$$\sum_{e \in \delta(v)} s_e = 0 \,(\text{mod } 2) \qquad \forall\, v \in V$$

This constraint has the disadvantage of non-linearity, but is a common enough method in arc routing problems (see [113] for examples). Another possible constraint could be the following.

$$\sum_{e \in \delta(v)} s_e = 2k_v \qquad \forall\, v \in V$$

Where $\{k_v\}$ are non-negative integers. This constraint is easier to use within standard IP algorithms, but still has the disadvantage of an introduced integer variable. Moreover, in a linear programming (LP) relaxation, the constraint would no longer ensure that each vertex has even degree.

3.1.2.3 Cost budget constraint

We ensure that the sum of the incurred costs for each edge is not greater than the budget.

$$\sum_{e \in E} c_e s_e \le C$$

3.1.2.4 Depot connection constraint

We ensure that the depot is incident on an included edge.

$$\sum_{e \in \delta(depot)} x_e \ge 1$$

3.1.2.5 Edge connectedness constraint

There are several options here that are variously utilized in the literature; the goal is to eliminate subtours. The basic method used is to define a proper subset of the vertices $S \subset V$ and then apply some check to this subset, and then repeat for *all* proper subsets of V. Valid checks include:

- Ensure that the total incidence of S is more than just the exclusive incidence (again, there are lots of possible inequalities which would work):

$$\sum_{e \in \Delta(S)} x_e - \sum_{e \in \bar{\delta}(S)} x_e > 0 \qquad \forall\, S \subset V$$

- Ensure that the incidence set of S is not empty.

$$\sum_{e \in \delta(S)} x_e > 0 \qquad \forall\, S \subset V$$

However, the ASRP has a complication that other problems do not. It is possible that in a feasible solution there are some vertices which are not visited (are not incident on an included edge); this allows the possibility of a subset $S^* \subset V$ for which $\Delta(S^*) = \{\varnothing\}$. We want to allow this possibility, which the above constraints would disallow.

To combat this, we utilize the standard form of if-then constraints. Suppose we have two functions f and g, and we want to ensure that when $f > 0$ then $g \geq 0$, i.e. $f > 0 \Rightarrow g \geq 0$. We introduce the following two constraints:

$$-g \leq My$$
$$f \leq M(1 - y)$$

Where M is a large number and y is binary. Consider: $f > 0 \Rightarrow y = 0 \Rightarrow g \geq 0$, as required. Of course, if $f \leq 0$ then y can be either 0 or 1, and g can take any value; the implication is only one way. For our purposes we can define f and g as follows:

$$f : \sum_{e \in \Delta(S)} s_e$$
$$g : \sum_{e \in \delta(S)} s_e - 2$$

So when there are any edges incident on one of the vertices in S, then we force the incidence set of S to contain at least two included edges. This results in the following equations, where for each subset S we have an associated binary variable y_S:

$$\sum_{e \in \Delta(S)} s_e \leq M(1 - y_S) \qquad \forall\, S \subset V$$
$$\sum_{e \in \delta(S)} s_e - 2 \geq -My_S \qquad \forall\, S \subset V$$

However, in our case we want to make the constraint that the incidence set of S be non-empty hold only if the total incidence of S is non-empty *and* the total incidence of $V \setminus S$ is non-empty. We need the

equivalent constraints to ensuring that $f > 0 \wedge h > 0 \Rightarrow g \geq 0$. We can easily accomplish this by exchanging the second constraint above with the following.

$$fh \leq M(1 - y)$$

However, this constraint is non-linear, so we instead use the following set of constraints.

$$-g \leq M(y + z)$$
$$f \leq M(1 - y)$$
$$h \leq M(1 - z)$$

Now g is forced to be non-negative only when both f and h are positive. This translates to the subtour elimination constraints as follows.

$$\sum_{e \in \Delta(S)} s_e \leq M(1 - y_S) \qquad \forall S \subset V$$
$$\sum_{e \in \Delta(V \setminus S)} s_e \leq M(1 - z_S) \qquad \forall S \subset V$$
$$\sum_{e \in \delta(S)} s_e - 2 \geq -M(y_S + z_S) \qquad \forall S \subset V$$

It is worth noting that even in an LP relaxation, these constraints will hold. In fact it is not necessary for y and z to be integer; the main characteristic they must have is that they can be 0 or greater than 0. The only modification is that M must be made large enough that no values of the variables will cause it to be met.

3.1.2.6 Other necessary constraints

There are several other constraints needed in the formulation to make sure the variables take appropriate values. We need to ensure that when s_e is positive, x_e is 1, and 0 otherwise. We need to ensure that the x, y and z variables are binary, and that s is non-negative and integer. We include the following constraints.

$$x_e \leq s_e \qquad \forall e \in E$$
$$s_e \geq 0 \qquad \forall e \in E$$
$$s_e \text{ integer} \qquad \forall e \in E$$
$$k_v \geq 0 \qquad \forall v \in V$$
$$k_v \text{ integer} \qquad \forall v \in V$$
$$x_e \text{ binary} \qquad \forall e \in E$$
$$y_S, z_S \text{ binary} \qquad \forall S \subset V$$

3.1.3 ASRP formulation 1

The first integer programming (IP) formulation is presented below. This formulation uses the definitions of variables used in the above section; x_e is 1 if edge e is included and 0 otherwise, s_e is the number of times it is included.

ASRP1:

$$\max \quad \sum_{e \in E} r_e x_e$$

$$\text{s.t.} \quad \sum_{e \in \delta(v)} s_e = 2k_v \qquad \forall\, v \in V$$

$$\sum_{e \in E} c_e s_e \le C$$

$$\sum_{e \in \delta(depot)} x_e \ge 1$$

$$\sum_{e \in \Delta(S)} s_e \le M(1 - y_S) \qquad \forall\, S \subset V$$

$$\sum_{e \in \Delta(V \setminus S)} s_e \le M(1 - z_S) \qquad \forall\, S \subset V$$

$$\sum_{e \in \delta(S)} s_e - 2 \ge -M(y_S + z_S) \qquad \forall\, S \subset V$$

$$x_e \le s_e \qquad \forall\, e \in E$$

$$s_e \ge 0 \qquad \forall\, e \in E$$

$$s_e \text{ integer} \qquad \forall\, e \in E$$

$$k_v \ge 0 \qquad \forall\, v \in V$$

$$k_v \text{ integer} \qquad \forall\, v \in V$$

$$x_e \text{ binary} \qquad \forall\, e \in E$$

$$y_S, z_S \text{ binary} \qquad \forall\, S \subset V$$

3.1.4 ASRP formulation 2

Broadly speaking, dominance relations are equalities or inequalities that reduce the set of feasible solutions to a smaller set that surely contains an optimal solution. Hence, a dominance relation is satisfied by at least one optimal solution for the problem but not necessarily by all feasible solutions.

Dominance Relation D1.

We define a dominance relation for the ASRP by noting that in an Euler tour (a minimum cost tour covering all the required edges), each edge is traversed at most twice. If an edge is traversed more than twice in a solution, it is possible to find another route through the same edges, which traverses each edge at most twice, at a not-greater cost. Let D1 be the dominance relation that $s_e \le 2$, for all edges $e \in E$.

Formulation 2 takes advantage of the above dominance relation, D1. Instead of x and s, variables to denote whether an edge is traversed, and the number of times it is traversed, we define variables p and q. Let $p_e = 1$ if edge e is traversed *once* in the solution, and 0 otherwise; let $q_e = 1$ if edge e is traversed *twice* in the solution. Clearly, then, $(p_e + q_e)$ is equivalent to x_e in ASRP1, and $(p_e + 2q_e)$ is equivalent to s_e.

We define the following integer program for the ASRP.

ASRP2:

$$\begin{aligned}
\text{max} \quad & \sum_{e \in E} r_e (p_e + q_e) & \\
\text{s.t.} \quad & \sum_{e \in \delta(v)} (p_e + 2q_e) = 2k_v & \forall\, v \in V \\
& \sum_{e \in E} c_e (p_e + 2q_e) \le C & \\
& \sum_{e \in \delta(depot)} (p_e + q_e) \ge 1 & \\
& \sum_{e \in \Delta(S)} (p_e + 2q_e) \le M(1 - y_S) & \forall\, S \subset V \\
& \sum_{e \in \Delta(V \setminus S)} (p_e + 2q_e) \le M(1 - z_S) & \forall\, S \subset V \\
& \sum_{e \in \delta(S)} (p_e + 2q_e) - 2 \ge -M(y_S + z_S) & \forall\, S \subset V \\
& p_e + q_e \le 1 & \forall\, e \in E \\
& p_e, q_e \text{ binary} & \forall\, e \in E \\
& k_v \ge 0 & \forall\, v \in V \\
& k_v \text{ integer} & \forall\, v \in V \\
& y_S, z_S \text{ binary} & \forall\, S \subset V
\end{aligned}$$

3.1.5 ASRP formulation 3

We can again make use of the dominance relation D1 and formulate the problem differently. For each edge e in the original graph G, we associate *two* binary variables x_e' and x_e'', each representing one traversal. In terms of the first formulation, $s_e = x_e' + x_e''$ and $x_e = x_e'$.

Now, we have the problem of how to deal with the reward. We have several options.

If the reward for traversing the original edge e was r_e, then we can let the reward for each of the new variables be r_e and change our reward function so that only one of the rewards is added:

$$\sum_{e \in E} x_e' r_e + \sum_{e \in E} x_e'' (1 - x_e') r_e$$

This method would work, but unfortunately the objective function is now non-linear – something we would prefer to avoid.

We could associate reward variables r_e' and r_e'' with the traversal variables x_e' and x_e'', respectively, and then introduce the following constraints to ensure that when one of the variables is 1, the reward variables are equal to the original reward, and when both of them are 1, then the reward variables are equal to half of the original reward. When neither is one, we don't really care.

$$r'_e + r''_e = (3 - x'_e - x''_e) r_e \qquad \forall\, e \in E$$

$$r_e - r_e = 0 \qquad \forall\, e \in E$$

The first equation ensures that the reward variables sum to the correct amount, and the second equation ensures that they are equal. This method would work, however it adds an additional 2*|*E*| variables and 2*|*E*| constraints to the model.

We could sum the rewards in the objective function only for the xe′ variables, and then add a constraint to ensure that xe″ is only equal to 1 when xe′ is already equal to 1 (if only one of the variables is selected, then it is x_e'). The objective function would then be:

$$\sum_{e \in E} x'_e r_e$$

And the added constraint set would be:

$$x''_e \leq x'_e \qquad \forall\, e \in E$$

This method is linear, and results in no new variables and only |*E*| new constraints, so it is preferable.

Dominance Relation 2.

All bridges are 0/2 edges. *Proof*: A bridge edge *b* may be equal to 0 by not including the bridge and by not including all the edges on the far side of the bridge from the depot. It may be equal to 2 so that it is traversed once on the way to the far side and once on the way back. If *b* equals 1 then any route would be able to get to the far side, but not return to the depot. Let D2 be the dominance relation that $x''_e = x'_e$, for all edges $e \in B$.

Note that we could also use D2 in ASRP2, by setting $p_e = 0\ \forall\, e \in B$.

The revised formulation is given below.

ASRP3:

$$\max \quad \sum_{e \in E} r_e x'_e$$

$$\text{s.t.} \quad \sum_{e \in \delta(v)} (x'_e + x''_e) = 2k_v \qquad \forall\, v \in V$$

$$\sum_{e \in E} c_e (x'_e + x''_e) \le C$$

$$\sum_{e \in \delta(depot)} (x'_e) \ge 1$$

$$\sum_{e \in \Delta(S)} (x'_e + x''_e) \le M(1 - y_S) \qquad \forall\, S \subset V$$

$$\sum_{e \in \Delta(V \setminus S)} (x'_e + x''_e) \le M(1 - z_S) \qquad \forall\, S \subset V$$

$$\sum_{e \in \delta(S)} (x'_e + x''_e) - 2 \ge -M(y_S + z_S) \qquad \forall\, S \subset V$$

$$x''_e \le x'_e \qquad \forall\, e \in E$$

$$x''_e = x'_e \qquad \forall\, e \in B$$

$$k_v \ge 0 \qquad \forall\, v \in V$$

$$k_v \text{ integer} \qquad \forall\, v \in V$$

$$x'_e, x''_e \text{ binary} \qquad \forall\, e \in E$$

$$y_S, z_S \text{ binary} \qquad \forall\, S \subset V$$

3.2 Construction heuristics

The existing heuristic concepts for arc routing problems are predominantly based on one of two ideas: augmenting the graph to make it Eulerian so that an Euler tour may be found (for single vehicle problems), and swapping edges between routes (for multiple vehicle problems). These methods are based on the premise that a *known* subset of the edges (usually all of them) is to be serviced. They manipulate the *order* of the edges. The difficulty for subset selection problems is that this premise is no longer true. In the ASRP, we have a complicating factor: we need to select only a subset of the arcs.

We present three construction heuristics for the ASRP. Our first two heuristics, PRUNE AND ROUTE and ROUTE AND PRUNE, exploit the properties of Eulerian graphs. We also present a constructive look-ahead heuristic, RICHEST NEIGHBOUR, which is based on the nearest neighbour principle with the addition of a budget-checking phase.

A key phase in both of the pruning heuristics is the solution of a Chinese Postman Problem, which is essentially equivalent to finding the solution to a minimum-weight matching problem. We use a variant of the algorithm designed by Edmonds and Johnson [78], modified from Evans and Minieka [87].

3.2.1 Prune and Route

Many of the algorithms and heuristics for arc routing problems, such as the Chinese Postman Problem and the Rural Postman Problem, attempt to transform the problem instance graph by adding arcs until the graph is Eulerian, and then using the resulting Euler tour as the solution. We adapt the same approach but, to compensate for our limited cost budget, we add an extra phase, the *pruning phase*. We make an analogy with solution methods for the Capacitated Arc Routing Problem (CARP), some of which are known as Cluster-First-Route-Second algorithms; our heuristic is a Prune-First-Route-Second heuristic.

There are three phases to the pruning heuristic. Phase 1: delete sufficient edges from G that when an Euler tour is found, the total cost is within budget; call this transformed graph G^-, and call its edge set E^-. Phase 2: augment G^- by replicating sufficient edges from E^- to make the graph Eulerian; this can be done by solving a matching problem on the odd-degree vertices. Call this augmented graph G^+. Phase 3: Find an Euler tour on G^+, this is our solution.

Phases 2 and 3 are together equivalent to solving a Chinese Postman Problem, and there are algorithms that easily deal with them. Phase 1 is more problematic. One method of deciding when to stop deleting edges is to actually perform phases 2 and 3 at each step, and repeat the whole process of prune-augment-route until the resulting Euler tour is within budget. However, this involves intensive computation. A modification which improves efficiency is to delete arcs until the sum of costs on the edges of the remaining graph is less than the cost budget; since we still have to augment the graph, we know that we won't delete too many. From there either a new estimate of how many arcs to delete can be made, or they can be deleted, and an Euler tour formed, one arc at a time.

An **Euler augmentation** G^E, of a graph G, is a least cost augmentation such that the resulting graph is Eulerian; every node has even degree. Let y_{ij} denote the number of times edge (v_i, v_j) occurs in G^E. We may assume that each edge has at most two copies, $y_{ij} \in \{0, 1, 2\}$, since any augmentation with greater than two copies of an arc is not least cost (see dominance relation D1 in Section 3.1.4).

There is also a decision to be made on the order in which the arcs should be deleted. There are two factors: we want to delete edges with low reward:cost ratio, and we want to delete edges that contribute to adding more arcs in phase 2. Under no circumstances will we permit the deletion of a *bridge*. (Recall that a bridge is an edge, the deletion of which disconnects the graph; *pendant edges* –edges that are incident on a vertex of degree 1 – may also be classified as bridges, but we will use the more restricted definition.) There are several possible schedules for the deletion of arcs, but for this chapter we will use the following three:

Schedule 1: First delete all the end edges (edges that are incident on a vertex of degree 1), in order of increasing reward:cost ratio. Next, delete all the edges that are not a bridge, in order of increasing reward:cost ratio. Note that the graph should be reassessed after each deletion. We desire to delete end edges first, because they necessarily must be traversed twice.

Schedule 2: Delete all the edges that are *either* pendant *or* are not a bridge, in order of increasing reward:cost ratio.

Schedule 3: The same as Schedule 2, except that the reward:cost ratio of end-edges is halved to make them more attractive to the selection. The reasoning behind this is that end edges must be traversed twice in succession; so, the deletion of them saves twice the cost that the deletion of other edges saves (assuming unit cost, this is not generally true).

The schedules are the same, except that in schedule 1, the two groups of candidate edges are deleted sequentially, and in schedule 2 they are deleted simultaneously. We check whether an edge (v_i, v_j) is a bridge by checking whether v_i and v_j are connected on the graph resulting from the deletion of edge (v_i, v_j).

Algorithm 3.1 heuristic PRUNE AND ROUTE

// Heuristic that iteratively prunes the graph and then finds the Euler augmentation

Scope: ASRP problems
Input: G *// The graph*
Output: R *// The solution route*

$k \leftarrow 0$
$G_k = (V_k, E_k) \leftarrow$ original graph G *// Initially set the reduced graph as the original graph*
$G_k^E \leftarrow$ the Euler augmentation of G_k
$C^* \leftarrow \sum y_{ij} c_{ij} \ni (v_i, v_j) \in E_k^E$
while $(C^* > C)$ **do**
  $k \leftarrow k + 1$
  Delete edge (v_a, v_b) from E_k, according to the schedule, such that (v_a, v_b) has minimum reward:cost ratio *and* (v_a, v_b) is not a bridge. The deletion of this edge must not disconnect the depot. Call the resulting graph G_k.
  $G_k^E \leftarrow$ the Euler augmentation of G_k
  $C^* \leftarrow \sum y_{ij} c_{ij} \ni (v_i, v_j) \in E_k^E$
end
$R \leftarrow$ Euler tour on G_k^E
end

3.2.2 Route and Prune

This heuristic, which may be considered a Route-First-Prune-Second method, is based on methods for the CARP that first form a *giant tour*, by solving the CPP relaxation, and then breaking that tour up into smaller tours. We proceed similarly, by solving the CPP relaxation of the ARCP (treating it as a CPP, where all arcs must be traversed), forming a *giant tour*, and then we delete sections of the tour until it is cost-feasible.

We use the following method to prune the giant tour. Identify all the *cycles* within the giant tour; a cycle is defined for our purposes to be a sequence of vertices, which starts and finishes at the same vertex. Consider the graph in Figure 3.1.

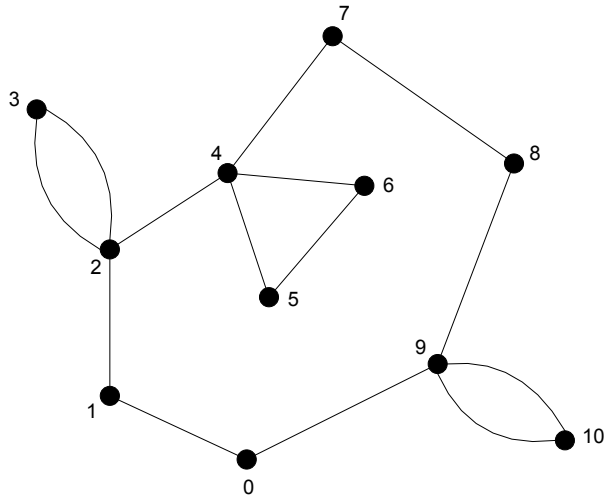


Figure 3.1: Examples of cycles

Starting at the depot (vertex 0), the route is R = (0, 1, 2, 3, 2, 4, 5, 6, 4, 7, 8, 9, 10, 9, 0). We identify the following cycles within R: (2, 3, 2), (4, 5, 6, 4), and (9, 10, 9); R itself is a cycle, but we exclude it from consideration. We then delete the cycles from our giant tour, by solving a bin-packing problem to minimize the loss of reward:cost while getting the cost within budget.

Note that we must be careful because some cycles are dependent on others; it is possible for there to be cycles-within-cycles, e.g. Figure 3.2.

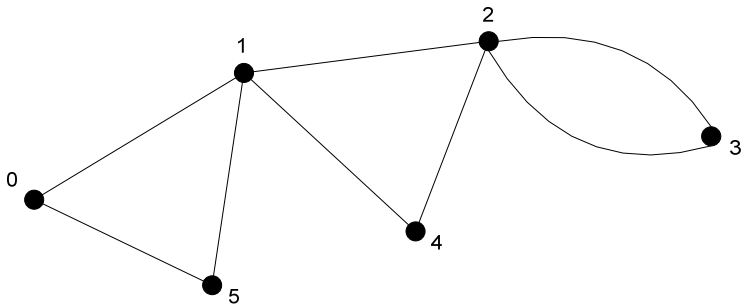


Figure 3.2: Example of nested cycles

In this example, where the route is R = (0, 1, 2, 3, 2, 4, 1, 5, 0), there are the following cycles: (1, 2, 3, 2, 4, 1), (2, 3, 2), and the full route is itself a cycle.

Algorithm 3.2 heuristic ROUTE AND PRUNE

// Heuristic that creates a giant CPP tour, and then deletes cycles until the
// subtour is feasible with respect to cost

Scope: ASRP problems
Input: G *// The graph*
Output: R *// The solution route*

$k \leftarrow 0$
$G_k = (V_k, E_k) \leftarrow$ original graph G *// Initially set the reduced graph as the original graph*
$G_k^E \leftarrow$ the Euler augmentation of G_k
$C^* \leftarrow \sum y_{ij} c_{ij} \ni (v_i, v_j) \in E_k^E$
$R_k \leftarrow$ Euler tour on G_k^E
while $(C^* > C)$ **do**
    Identify all the cycles in R_k.
    $k \leftarrow k + 1$
    Delete the cycle that has the minimum reward:cost ratio, and which also contains
        no other cycles. The deletion of this cycle must not disconnect the
        depot. Call the resulting route R_k.
    $G_k^E \leftarrow$ the Euler augmentation of G_k
    $C^* \leftarrow \sum y_{ij} c_{ij} \ni (v_i, v_j) \in R_k$
end
$R \leftarrow R_k$
end

3.2.3 Richest Neighbour

This heuristic is an adaptation of the classic Nearest Neighbour heuristic for the Travelling Salesman Problem, with the addition of a check to ensure continuing cost feasibility.

Starting at the depot, and at each step, a path is formed by adding an adjacent edge to the route if its reward:cost ratio is the largest of all edges, *and* if the shortest path back to the depot from the arc results in a cost-feasible solution. Once no more edges can be added, the shortest path to the depot is appended to the end of the path, forming a closed subtour.

The heuristic can be extended from a 1-period look-ahead to an n-period look-ahead by considering the added reward if the next n edges are added to the route, and then adding the first edge of the candidate which resulted in the greatest increase in reward. Only one edge is added to the route at a time, but more possibilities are considered for higher values of n. This is analogous to a *path-scanning* approach.

At each iteration the heuristic considers all other nodes in the graph to be the "destination node" e for the current iteration. The first node along this shortest path is denoted e_1. If the shortest path from the

current node to e is within the look-ahead (not greater than the look-ahead) *and* if the addition of the arc between the current node and e_1 allows a return to the depot from e_1 within the cost budget, then e is a valid destination node. The reward:cost ratio, for all untraversed arcs along the shortest path to e, is calculated; if it is better than any other valid destination nodes for this iteration it is selected. At the end of the iteration e_1 is added to the node route. If there were no valid destination nodes then the shortest path from the current node to the depot is added to the end of the route and the solution is returned.

Algorithm 3.3 gives the pseudo-code for the RICHEST NEIGHBOUR family of heuristics, with look-ahead n.

Algorithm 3.3 heuristic RICHEST NEIGHBOUR (n)

```
// Heuristic that constructs a path one arc at a time, and then takes the shortest
// path back to the depot

Input: G, SP(i, j), n   // The graph, the shortest path cost matrix between all
    pairs of nodes and the look-ahead parameter
Output: R   // The solution route, expressed as a sequence of nodes

k ← 0
R_k ← depot   // Set the depot node as the current node
finishedExpansion ← false
while (finishedExpansion = false) do
    // Find the next destination node (up to n away)
    rc* ← 0   // Initialize the best reward:cost ratio to zero
    e* ← null   // Initialize the best next destination node as null
    do e ∈ E \ R_k   // Loop through all the nodes in the graph except the current node
        e_1 ← the first node on the path from R_k to e
        if SP(R_k, e) ≤ n then
            if C + SP(e_1, depot) ≤ B then
                reward ← the sum of all the rewards of the currently untraversed
                            arcs in the shortest path from R_k to e
                rc ← reward / SP(R_k, e)   // Calculate the reward:cost ratio for this path
                if rc > rc* then   // If this destination node is better then update the best
                    rc* ← rc   // Update the best reward:cost ratio
                    e* ← e   // Update the best next destination node
                end
            end
        end
    end

    // Update the current node
    if e* is not null then
```

```
            e1 ← the first node on the path from Rk to e
            k ← k + 1
            Rk ← e1
        else
            finishedExpansion ← true
        end
    end

    // No more expansion can be performed, return to depot
    while (Rk is not depot) do
        e1 ← the first node on the path from Rk to depot
        k ← k + 1
        Rk ← e1
    end
end
```

3.3 Improvement procedures

We develop a number of local search move-types that can be applied to the solutions created by the constructive heuristics. These are incorporated into simple implementations of the Steepest Ascent and Tabu Search local search heuristics. Note that in this chapter we do not use the Modular Local Search framework; we simply construct traditional versions of these metaheuristics.

An important point to note is that these move-types all start with a valid ASRP tour, and the resulting solution is also a valid ASRP tour. It is not possible to arrive at a solution that is infeasible with respect to being a valid tour or having subtours, assuming that the initial solution was feasible to start with. For the ASRP we need only be concerned with infeasibility due to exeeding the cost budget.

We also define a procedure, Algorithm 3.4, that deletes redundancy in the completed route. A solution route to the ASRP may be thought of as an Euler tour on the subgraph which consists only of those edges which are traversed in the solution. For an Euler tour to be optimal for the subgraph, each edge must be traversed no more than twice.

This procedure reduces the number of times an edge is traversed and then finds a resulting Euler tour. If an edge is traversed an odd number of times in the original route, it is traversed once in the modified route, and if an edge is traversed an even number of times in the original route, then it is traversed twice in the modified route. The modified route will have the same reward, but its incurred cost will be less than, or equal to, that of the original route.

Since this procedure re-routes the solution through the same subset of arcs, the route obtained potentially traverses the arcs in a different order. This procedure can be used to “reshuffle” a solution, potentially creating new neighbours.

Algorithm 3.4 procedure DELETE REDUNDANCY

// Reduces the number of times an arc is traversed and re-routes the solution through
// the same set of arcs, possibly in a different order.

Scope: ASRP problems
Input: *R* *// The current solution route, expressed as an ordered sequence of arcs*
Output: *R** *// The new solution route*

$A \leftarrow$ the distinct set of arcs in R *// Each distinct arc appears in A only once*
$G^E \leftarrow$ construct Eulerian graph from A using the matching algorithm of Edmonds and Johnson [78]
$R^* \leftarrow$ CONSTRUCT EULER TOUR FROM EULERIAN GRAPH(G^E) (Algorithm 2.1)
end

3.3.1 Basic ASRP move-types for local search heuristics

The following procedures are combined as moves in local search heuristics, such as Tabu Search and Steepest Ascent. These move-types are defined on general graphs, not necessarily limited to the grid graphs that are used for the experimental phase, however they are most easily visualized on grids, as in Figure 3.3.

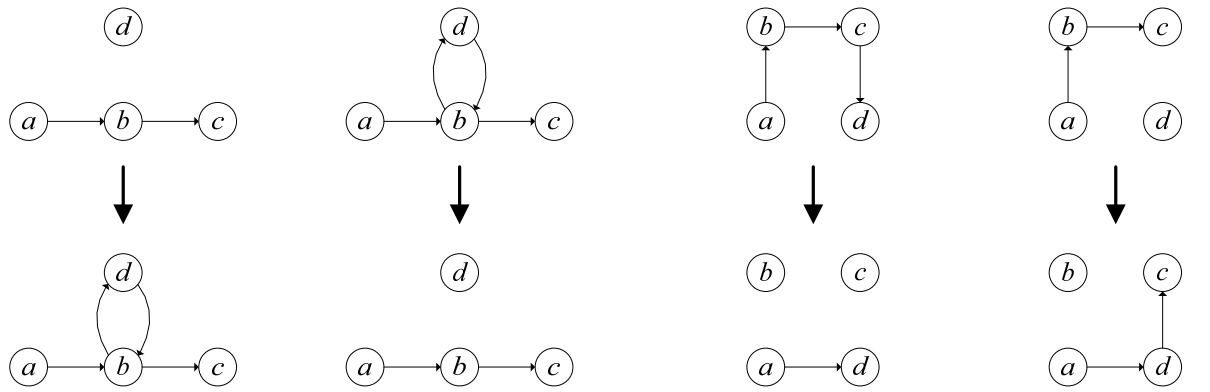


Figure 3.3: Basic ASRP move-types

3.3.1.1 Add

Given a route S traversing a subset $R \subseteq E$ of edges, and given an edge $(v_j, v_k) \in E \backslash R$, the ADD procedure adds this edge twice, inserting it between a pair of adjacent arcs.

For example, if a section of the route is (…, A, B, C, E, …) and we wish to add edge (C, D), then the new path is (…, A, B, C, D, C, E, …). The cost of the route is increased by $c_{CD} + c_{DC} = 2c_{CD}$, for a symmetric cost matrix.

3.3.1.2 Drop

DROP is the opposite procedure to ADD. Given a route S traversing a subset $R \subseteq E$ of edges, and given an edge $(v_j, v_k) \in R$, which is included at least twice consecutively in S, this procedure constructs a new, shorter, route without this edge.

For example, if a section of the route is (…, A, B, C, D, C, E, …), and we wish to DROP edge (C, D), then the new path is (…, A, B, C, E, …). The cost of the route is reduced by $c_{CD} + c_{DC} = 2c_{CD}$, for a symmetric cost matrix.

3.3.1.3 Shortcut

Given a route S traversing a subset $R \subseteq E$ of edges, and given a subpath of S, $S' = (v_i, v_j, v_m, v_n)$, if edge $(v_i, v_n) \in E$ then the SHORTCUT procedure deletes the edges (v_i, v_j), (v_j, v_m), and (v_m, v_n) from S' and replaces them with edge (v_i, v_n). On a grid, this is equivalent to traversing one side of a square, instead of the other three.

For example, if a section of the route is (…, A, B, C, D, E, F, …), and we wish to shortcut the subpath (B, C, D, E), then the new path is (…, A, B, E, F, …). The cost of the route has been reduced by $c_{BC} + c_{CD} + c_{DE} - c_{BE}$.

3.3.1.4 Detour

Given a route S traversing a subset $R \subseteq E$ of edges, and given a subpath of S, $S' = (v_i, v_j, v_k)$, if edge $(v_i, v_m) \in E$ and edge $(v_m, v_k) \in E$ then the DETOUR procedure deletes edges (v_i, v_j), (v_j, v_k) from S' and replaces them with edges (v_i, v_m), (v_m, v_k). On a grid, this is equivalent to traversing two sides of a square, rather than the other two.

For example, if a section of the route is (…, A, B, C, D, F, …), and we wish to detour the subpath (B, C, D) via vertex E, then the new path is (…, A, B, E, D, F, …). The cost of the route has been reduced by $c_{BC} + c_{CD} - c_{BE} - c_{ED}$.

3.3.2 Extended ASRP move-types

In addition to the basic move-types that are used in the improvement procedures in this chapter, we also define four *extended* move-types that will be used in later chapters. The extended move-types are generalizations of their basic counterparts. All the extended move-types are general to any type of graph; they are not limited to grid graphs.

The extended move-types are parameterized with a **look-ahead** parameter λ. The look-ahead determines the potential size of the move; $\lambda = 1$ makes the extended move-types equivalent to the basic move-types.

3.3.2.1 nAdd

NADD is the generalized version of ADD. Given a route S visiting a subset $R \subset V$ of vertices, and given a vertex $v_i \in R$ and a vertex $v_j \in V \setminus R$, let Y_{ij} be the length of the shortest path between v_i and v_j. If $Y_{ij} \leq \lambda$ then the NADD procedure inserts the shortest path between v_i and v_j into S, starting at v_i, and then inserts the shortest path between v_j and v_i into S directly after it to reconnect the route. The length of the route is increased by $2Y_{ij}$.

3.3.2.2 nDrop

NDROP is the generalized version of DROP. Given a route S with a sequence of arcs a_i, a_{i+1}, ..., a_n, NDROP deletes a *cycle* within this sequence, such that the length of the cycle does not exceed λ.

For example, if a section of the route is (..., A, B, C, D, C, F, B, ...), then there are two cycles that can be considered for deletion: (C, D, C) and (B, C, D, C, F, B). The cost of the route is reduced by the length of the cycle.

Note that the Route and Prune construction heuristic is equivalent to repeated application of NDROP, starting from the "giant" tour resulting from solving the CPP relaxation.

3.3.2.3 nShortcut

NSHORTCUT is the generalized version of SHORTCUT. Given a route S visiting a subset $R \subset V$ of vertices, and a sequence of vertices within S: v_i, v_{i+1}, ..., v_j, let Y_{ij} be the length of the shortest path between v_i and v_j. If $Y_{ij} \leq \lambda$ then the NSHORTCUT procedure replaces the path in the route from v_i to v_j with the shortest path from v_i to v_j.

3.3.2.4 nDetour

NDETOUR is the generalized version of DETOUR. Given a route S visiting a subset $R \subset V$ of vertices, and a sequence of vertices within S: v_i, ... v_p, ..., v_j, and let v_q be a vertex not in R, then let Y_{iq} be the length of the shortest path between v_i and v_q, and let Y_{qj} be the length of the shortest path between v_q and v_j.

If $Y_{iq} \leq \lambda$ and $Y_{qj} \leq \lambda$ then NDETOUR replaces the path in the route from v_i to v_j via v_p with the shortest path from v_i to v_q and the shortest path from v_q to v_j. Essentially the path from v_i to v_j detours through v_q.

3.3.3 Steepest Ascent

Ascent Search is a heuristic improvement procedure (descent for minimization problems). It starts from an initial solution and, at each step, selects an improving move and sets that as the current solution. It continues until no improving moves are available, and the current solution is then a local optimum. Steepest Ascent follows the same procedure, but it selects the *best* improving move at each step.

Our version of Steepest Ascent selects from the neighbourhood of solutions generated by the moves described above: ADD, DROP, SHORTCUT, and DETOUR.

3.3.4 Tabu Search

Tabu Search is a metaheuristic that has been applied with considerable success to routing problems. We consider a basic version, as an improvement procedure to the construction heuristics. Its basic operation is similar to Steepest Ascent. It starts with an initial solution and at each step, evaluates all the possible moves and compiles a list of *candidate* moves. It then checks the *tabu status* of each of the moves and, if the move is not tabu, or if it is tabu but satisfies the *aspiration criterion*, then the move is added to the list of *admissible* moves. Then, the best admissible move is selected and becomes the current solution and the tabu list is updated.

The tabu list works by making some moves unavailable for selection. The purpose of this is to encourage diversification and prevent cycling. This version of Tabu Search uses the four moves described above: ADD, DROP, SHORTCUT, and DETOUR. Any arcs which are added to, or dropped from, the route as a result of a move are made tabu for a set number of moves (known as the *tabu tenure*). The exception is when a move requires a tabu edge and that move would result in the satisfaction of the aspiration criterion. The most common aspiration criterion is that the resulting solution would be better than the best found so far. After each step, the number of tabu iterations remaining for each edge on the list is reduced by one.

Our version of Tabu Search for this chapter allows solutions that are infeasible with respect to the cost budget, but apply a penalty to each unit of cost over the cost budget that is incurred, and the best-solution-so-far is only updated with feasible solutions.

3.4 Problem generation principles

In this section we discuss some issues involved in problem generation for the ASRP.

3.4.1 Graph generation

All of our computational experiments are implemented on grid graphs with arcs of unit length. A **grid graph** is one in which all the arcs are laid out in the form of a grid, and we denote the size of a grid by the number of vertices down and across. For example, Figure 3.4 gives an example of a 6×6 grid graph; the arcs are laid out on a lattice of six rows by six columns of vertices.

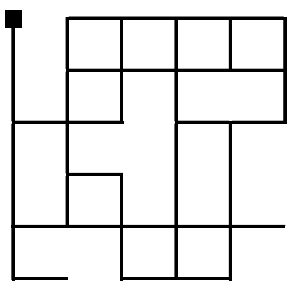


Figure 3.4: Example of a 6x6 grid graph

Grid graphs offer a number of advantages over more general graphs, and have been used several times by other researchers. Frizzell and Giffin [104] study a variation on the Vehicle Routing Problem, using quite small (6×6) square grid graphs with constant edge costs. They note that grid graphs have a layout that more closely models that of actual road networks, especially in an urban environment. From an experimental perspective, they also allow more finely grained control over the structure of the networks, reducing the degrees of freedom. Mohan et al. [195] study the Stochastic Eulerian Tour Problem on grid graphs of sizes 4×4, 5×5, 6×6, 7×7, 8×8 and 9×9. Johnston and Chukova [151] briefly study a version of the Arc Subset Routing Problem, which they call the Rural Postman Problem with Rewards, on grid graphs of size 9×9. For the purposes of exposition, we will limit ourselves to a 10×10 grid, and we define each arc to have unit length.

3.4.1.1 General graph characteristics

In order to compare and classify different graphs, we define several graph characteristics.

Given a graph, $G = (V, E)$, let

- $|E|$ be the cardinality of the edge set (the number of edges).
- $|V|$ be the cardinality of the vertex set. For a 10 × 10 grid, this is nominally 100, but, as can be seen in Figure 3.5, the graph generation techniques may result in some of the vertices being isolated.
- D_{v_i} be the **degree** of vertex v_i (the number of arcs incident on v_i).
- $\overline{D}^V$ be the average degree of the vertices in the graph.

$$\overline{D}^V = \frac{\sum_{v_i \in V} D_{v_i}}{|V|}$$

- $E^1 \subset E$ be the set of **pendant-edges**. A pendant-edge is an edge incident on a vertex v_i, with $D_{v_i} = 1$. Let $|E^1|$ be the cardinality of the set of pendant-edges.
- Υ be the matrix of shortest path lengths, where element Υ_{ij} is the length of the shortest path from vertex v_i to vertex v_j. Further, let $\overline{\Upsilon}_V$ be the average of the shortest distances between all pairs of vertices, and let $\overline{\Upsilon}_d$ be the average of the shortest distances from the depot to every other vertex.

3.4.1.2 Grid graph characteristics

We also define several measures specific to grid graphs.

Given a grid graph, $G = (V, E)$, let

- m be the number of rows of vertices, and n be the number of columns of vertices, in the grid. We then say that G is an $m \times n$ grid.

- E^C be the set of edges in the complete grid (one where all possible edges are included) corresponding to G. $E \subseteq E^C$. Let $|E^C|$ be the cardinality of the edges of the complete grid. For an $m \times n$ grid,

$$\left|E^C\right| = m(n-1) + n(m-1) = 2mn - m - n$$

- Ψ^G be the **density** of G. The density is the number of edges in the graph divided by the number of edges there would be in a complete grid.

$$\Psi^G = \frac{|E|}{\left|E^C\right|}$$

- $P^C = \left\{P_1^C, P_2^C, ..., P_k^C\right\}$ be a **partition** of E^C, such that every edge in E^C is an element of one, and only one, element of P; i.e. $P_i^C \subset E^C$ and $P_i^C \cap P_j^C = \varnothing \ \forall \ i,j$; and $P_1^C \cup P_2^C \cup \ldots \cup P_k^C = E^C$. Let $P = \{P_1, P_2, ..., P_k\}$ be the corresponding partition of $E \ni P_i \subseteq P_i^C \forall i$.

- Ψ^{P_i} be the density of set i of partition P.

$$\Psi^{P_i} = \frac{|P_i|}{\left|P_i^C\right|}$$

3.4.1.3 Example of a grid graph

In our experiments we use a 10 × 10 grid, so $\left|E^C\right|$ = 180.

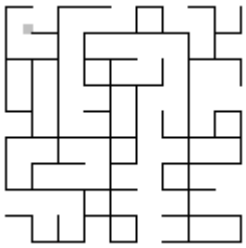


Figure 3.5. Example of a grid graph.

In Figure 3.5, we see an example of a 10×10 grid graph. This graph has the following characteristics: $|E|$ = 110, $|V|$ = 97, $\overline{D}^V$ = 2.27, $\left|E^1\right|$ = 17, Ψ^G = 0.61, $\overline{\Upsilon}_V$ = 11.72, $\overline{\Upsilon}_d$ = 13.21. We may derive other measures from these, if desired, for example, the proportion of end-edges is 17/120 = 0.155.

3.4.1.4 Graph generation methods

Two methods of generating random grid-graphs are presented. The main restriction on the generation of the graphs is that, for our purposes, the graphs must be connected. Clossey et al. [54] also used grid graphs, studying the Chinese Postman Problem with Turn Penalties. Their method of grid generation was to **grow** a connected graph by adding arcs adjacent to those already added. We believe that this

method will tend to create graphs with regions of high density, surrounded by regions of lower density. We refer to this method as GRID GROW, and it is described in Algorithm 3.5.

Algorithm 3.5 procedure GRID GROW

// Constructs a graph by iteratively adding arcs from the full grid template that are // adjacent to an already added arc

Scope: Routing problems

Input: E^C, Ψ *// The edges of the complete grid and the desired density of the generated subgrid*

Output: E *// The set of edges in the generated subgraph of the complete grid*

$E \leftarrow \varnothing$

Randomly choose an edge $e \in E^C$

Add e to E

repeat

Randomly choose an edge $e \in E^C \setminus E$

if e is adjacent to another arc in E **then** add e to E

until $|E| = \Psi \times |E^C|$ *// The desired density is achieved*

end

Our second method of grid graph generation is to randomly select one of the 180 arcs and mark it "selected", then select one of the remaining 179 arcs, etc., until we have a graph of the required density. To ensure the connectedness of the graph, we check to see if it is connected, and discard it if it is not. This method, which we call GRID SELECT, is more computationally intensive, but still quick enough that several hundred instances may be generated in no more than a few minutes.

Algorithm 3.6 procedure GRID SELECT

// Constructs a graph by adding a random subset of arcs from the complete grid and
// then checking to see if connected. This repeats until a connected graph is found.

Scope: Routing problems
Input: E^C, Ψ *// The edges of the complete grid and the desired density of the generated subgrid*
Output: E *// The set of edges in the generated subgraph of the complete grid*

repeat
$E \leftarrow \varnothing$
Randomly choose $E \subseteq E^C$, where $|E| = \Psi \times |E^C|$
Check whether E results in a connected graph
until E results in a connected graph
end

To explore whether the two methods generate different types of graphs we perform a small experiment, where we generate a large number of graphs by each method, and then compare them using the graph and grid characteristics.

Experiment: Generate 30 instances for each of the following edge set cardinalities, on a 10×10 structure: {90, 100, 110, 120, 130, 140, 150}. Do this for both the GRID GROW and the GRID SELECT methods. In order to compare the distribution of edges within the graphs, we define the following partition of the full grid, as illustrated in Figure 3.6:

$$P^C = \left\{P_1^C, P_2^C, P_3^C, P_4^C, P_5^C, P_6^C, P_7^C, P_8^C, P_9^C\right\}, \text{ where } \left|E^C\right| = 180 \Rightarrow \left|P_i^C\right| = 20.$$

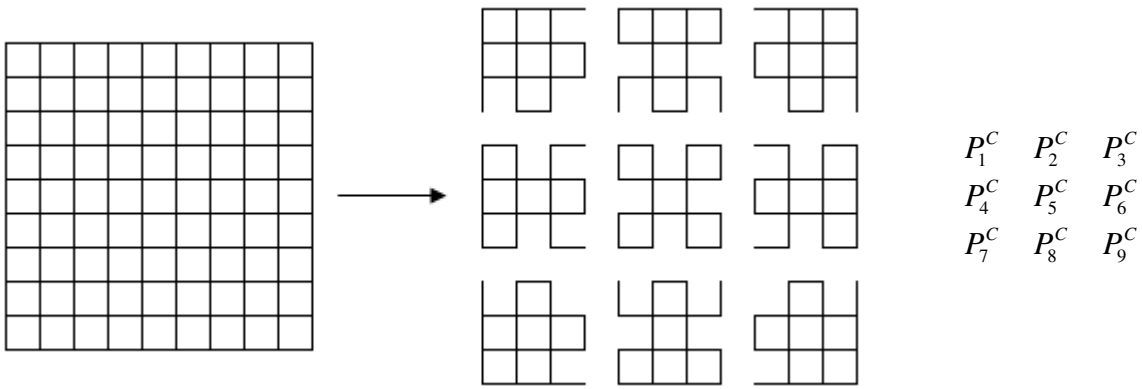


Figure 3.6: Graph partition

We used the multivariate analysis technique of *discriminant analysis* to determine if we could predict whether a particular problem instance was formed using GRID GROW or GRID SELECT, just from its

characteristics. There were 420 instances altogether, and we found a linear discriminant function (LDF) which was very successful (99.5%) in telling the two types of instance apart. It made two misclassifications. In Figure 3.7, we see some typical examples of grids generated using GRID GROW, in Figure 3.8, we see some generated using GRID SELECT. The two instances which were misclassified are shown in Figure 3.9; they were both generated using GRID GROW, and were classified by the LDF as GRID SELECT.

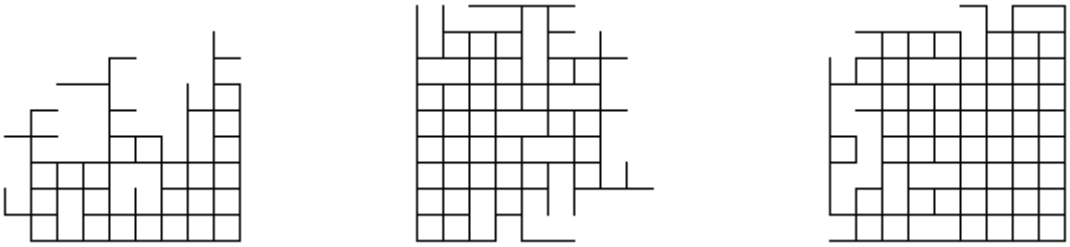


Figure 3.7: Grids generated using GRID GROW

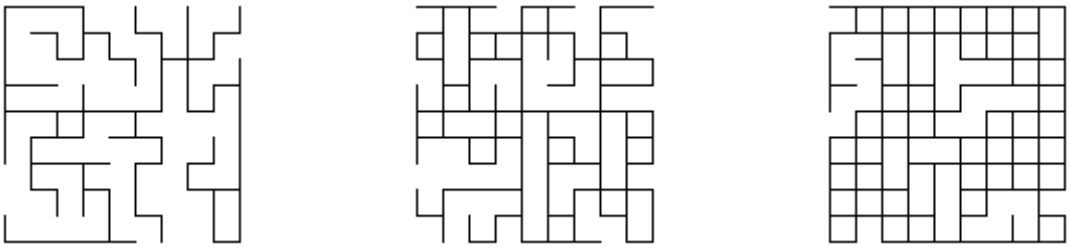


Figure 3.8: Grids generated using GRID SELECT

Figure 3.9: Grids which were misclassified

In the subsequent experiments, we use both types of grid graphs to test the heuristics.

3.4.2 Rewards and classes

Initially, the rewards are set randomly, according to a uniform distribution, U(a, b). In order to create interesting examples, we introduce the concept of *classes*. Each arc within a class has its reward drawn from the same distribution. For example, a problem instance might have two classes of arcs, normal

roads and arterial roads, and these might have different reward distributions, e.g. U(10, 20) and U(30, 35), respectively. A variety of class distributions are applied.

3.4.3 Cost budget

The best method of distinguishing the performance of the heuristics will be their relative performances for different levels of the cost budget, so this factor will be increased incrementally. Let us define a measure of the size of the cost budget relative to the total cost of the grid (the sum of all the costs on the arcs) as B = (cost budget)/(sum of arc costs). A value of $B = 1$ does not necessarily mean that all the arcs may be traversed, since some may be traversed more than once. On a complete $m \times m$ grid, $0.1*\Psi^G$ is the required budget to cross from one side of the grid to the other, and back. Therefore, a reasonable á priori range for B seems to be between 0.1 and 1.0.

3.5 Specific problem instances

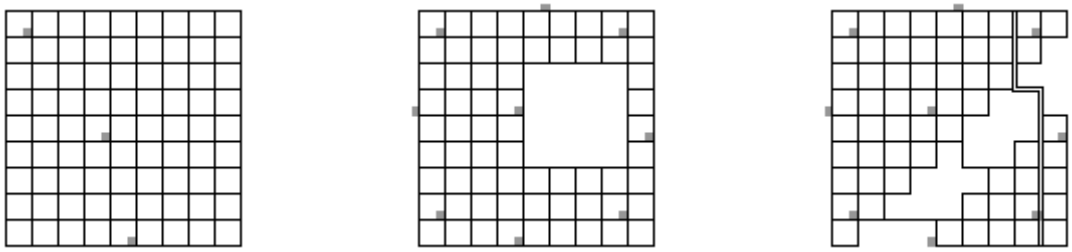
Two types of problem instance were generated. The first type were designed around a certain concept, which we call **designed** graphs. We generated three designed graphs and varied the depot location to give distinct instances. The second type were randomly generated graphs. There were 12 specifications of edge number and reward distribution, and two methods of generation, giving 24 distinct instances.

3.5.1 Designed graphs

For each of the designed grids, we used three different approximate depot locations: edge, corner, and middle. For the asymmetric graphs below, we used all combinations of depot locations.

The designed grids (shown in Figure 3.10) were of three stylized types, giving 21 problem instances in all. The grey squares correspond to the depot locations; the numbering is top to bottom, left to right (as a page of text would be read).

- **Complete grid**. This is a complete grid, 180 arcs, density = 1. The distribution of rewards was U(10, 20). There were three depot locations used for this grid; the graphs are labeled C1, C2, C3.
- **Lake**. 140 edges, density = 0.78. The distribution of rewards was U(10, 20). There were nine depot locations used for this grid; the graphs are labeled L1, L2, …, L9.
- **River**. 154 edges, density = 0.86. There were two classes of edge, with uniform rewards for each class; the single line edges were in the range U(10, 15), the double line edges were in the range U(15, 25). There were nine depot locations used for this grid; the graphs are labeled R1, R2,…, R9.

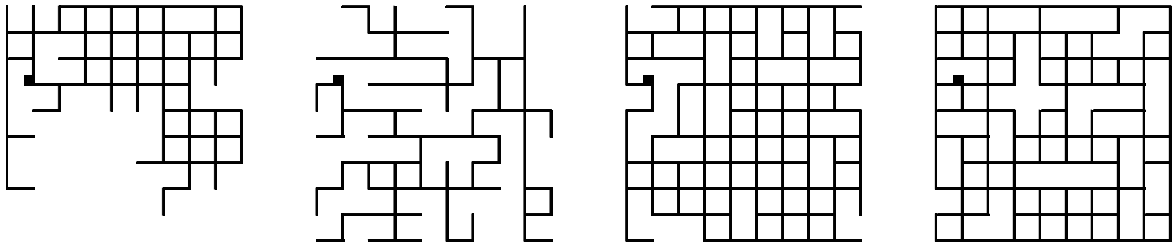


Complete – 3 depot locations Lake – 9 depot locations River – 9 depot locations

Figure 3.10: Designed problem instances

3.5.2 Random graphs

Two phases of experimentation were performed on random graphs. An initial phase and a more extensive phase 2. Figure 3.11 gives some examples of the graphs generated by the two generation methods.



GROW 90 edges SELECT 90 edges GROW 150 edges SELECT 150 edges

Figure 3.11: Examples of random grid graphs

All of the problem instances generated were connected grid graphs.

3.5.2.1 Phase 1 problem instances

Phase 1 involves a small set of specifications with the parameters in Table 3.2. These four specifications for the two methods of generation and the 15 cost budgets give 120 distinct problem instances. We generate five realizations of each instance, giving a total of 600 realizations. We call this problem set A.

Table 3.2: Specifications for problem set A of random graphs

Problem Set	No. Arcs	Class 1		Class 2	
		#	Dist^{n}.	#	Dist^{n}.
GRID GROW 1 (G1)	90	45	U(5, 10)	45	U(15, 20)
GRID GROW 2 (G2)	110	55	U(5, 10)	63	U(15, 20)
GRID GROW 3 (G3)	130	65	U(5, 10)	27	U(15, 20)
GRID GROW 4 (G4)	150	75	U(5, 10)	55	U(15, 20)
GRID SELECT 1 (S1)	90	45	U(5, 10)	45	U(15, 20)
GRID SELECT 2 (S2)	110	55	U(5, 10)	63	U(15, 20)
GRID SELECT 3 (S3)	130	65	U(5, 10)	27	U(15, 20)
GRID SELECT 4 (S4)	159	75	U(5, 10)	55	U(15, 20)

3.5.2.2 Phase 2 problem instances

Phase 2 involves a more extensive round of experimentation. Each of the twelve specifications given in Table 3.3 were generated using both GRID GROW and GRID SELECT, giving 120 instances.

There are three factors which are varied: the graph generation technique (GRID GROW and GRID SELECT), the cardinality of the edge set (90, 110, 130, 150 edges), and the proportion of edges in each class (50-50, 30-70, 70-30).

Table 3.3: Specifications for problem set B of random graphs

Problem Set	No. Arcs	Class 1		Class 2	
		#	Dist^{n}.	#	Dist^{n}.
GRID GROW 1 (G1)	90	45	U(5, 10)	45	U(15, 20)
GRID GROW 2 (G2)	90	27	U(5, 10)	63	U(15, 20)
GRID GROW 3 (G3)	90	63	U(5, 10)	27	U(15, 20)
GRID GROW 4 (G4)	110	55	U(5, 10)	55	U(15, 20)
GRID GROW 5 (G5)	110	33	U(5, 10)	77	U(15, 20)
GRID GROW 6 (G6)	110	77	U(5, 10)	33	U(15, 20)
GRID GROW 7 (G7)	130	65	U(5, 10)	65	U(15, 20)
GRID GROW 8 (G8)	130	39	U(5, 10)	91	U(15, 20)
GRID GROW 9 (G9)	130	91	U(5, 10)	39	U(15, 20)
GRID GROW 10 (G10)	150	75	U(5, 10)	75	U(15, 20)
GRID GROW 11 (G11)	150	45	U(5, 10)	105	U(15, 20)
GRID GROW 12 (G12)	150	105	U(5, 10)	45	U(15, 20)
GRID SELECT 1 (S1)	90	45	U(5, 10)	45	U(15, 20)
GRID SELECT 2 (S2)	90	27	U(5, 10)	63	U(15, 20)
GRID SELECT 3 (S3)	90	63	U(5, 10)	27	U(15, 20)
GRID SELECT 4 (S4)	110	55	U(5, 10)	55	U(15, 20)
GRID SELECT 5 (S5)	110	33	U(5, 10)	77	U(15, 20)
GRID SELECT 6 (S6)	110	77	U(5, 10)	33	U(15, 20)
GRID SELECT 7 (S7)	130	65	U(5, 10)	65	U(15, 20)
GRID SELECT 8 (S8)	130	39	U(5, 10)	91	U(15, 20)
GRID SELECT 9 (S9)	130	91	U(5, 10)	39	U(15, 20)
GRID SELECT 10 (S10)	150	75	U(5, 10)	75	U(15, 20)
GRID SELECT 11 (S11)	150	45	U(5, 10)	105	U(15, 20)
GRID SELECT 12 (S12)	150	105	U(5, 10)	45	U(15, 20)

Altogether, if we have a full factorial experimental design, with the following factors and levels, then we have $2 \times 4 \times 3 \times 5 = 120$ distinct problem specifications.

- Graph generation method. Two levels – GRID GROW and GRID SELECT.
- Edge set cardinality. Four levels – 90, 110, 130, 150 edges.
- Class proportions. Three levels – 50-50, 30-70, 70-30.
- Cost budget. 0.2, 0.4, 0.6, 0.8, 1.0 times the total grid cost.

3.6 Preliminary experimentation

The heuristics and experiments performed in this chapter were programmed in C++ in the early stages of the research. All of this code was later replaced with the Java object oriented framework developed for Modular Local Search. Computation was performed on an AMD Athlon4 processor, running under Windows XP, with 1.2 GHz and 240Mb RAM.

To begin with we perform a simple sensitivity analysis on the parameters for RICHEST NEIGHBOUR (the look-ahead period) and for TABU SEARCH (the penalty for infeasible solutions, the number of iterations, and the tabu list size). We then perform a more detailed investigation of the construction heuristics and improvement procedures.

3.6.1 Richest Neighbour look-ahead period

In principle, we can set the look-ahead to whatever we wish; in practice the computation times increase geometrically, quickly becoming impractical for large numbers of experiments. Table 3.4 shows the reward values and computation times (in seconds) for look-ahead periods 1-13 on a sample of instance from problem set B. The highlighted values are the maxima achieved for each problem instance. The problem names are constructed from the generation specification (see Table 3.3) and the cost budget, which is calculated as a proportion of the total cost.

Table 3.4. Rewards and computation times for RICHEST NEIGHBOUR sensitivity analysis

REWARD

Heuristic	G1-144	G1-216	G1-72	G10-144	G10-216	G10-72	G7-144	G7-216	G7-72	Grand Total
RN1	1022.86	1109.38	831.96	1414.28	1709.46	826.99	1366.83	1484.57	704.27	10470.61
RN2	**1071.47**	**1132.98**	853.33	1619.82	1780.59	952.80	1486.98	**1644.01**	970.35	11512.32
RN3	692.79	692.79	692.79	1254.65	1254.65	962.61	1300.93	1300.93	1029.10	9181.24
RN4	888.09	888.09	888.09	1626.53	1697.52	964.88	1382.86	1382.86	1012.51	10731.42
RN5	884.31	884.31	884.31	1542.02	1542.02	**1045.14**	1373.91	1373.91	1000.77	10530.71
RN6	904.04	904.04	898.10	1455.94	1455.94	994.57	1507.86	1530.06	996.85	10647.40
RN7	737.80	737.80	737.80	1584.34	1584.34	1024.07	1368.77	1368.77	1044.82	10188.54
RN8	894.26	894.26	894.26	1628.22	1628.22	1015.64	1494.69	1557.21	1038.78	11045.54
RN9	947.63	947.63	868.95	1630.17	1768.19	1012.64	1495.47	1530.05	1033.36	11234.08
RN10	1021.43	1021.43	831.68	1653.92	1776.44	1011.25	1506.20	1577.60	1044.45	11444.38
RN11	992.16	992.16	**908.96**	**1656.06**	1766.45	1012.09	1495.56	1508.83	1036.28	11368.55
RN12	890.75	890.75	890.75	1652.74	1820.89	1021.99	**1534.63**	1544.79	1012.40	11259.68
RN13	1016.55	1016.55	859.19	1650.25	**1860.82**	1008.31	1524.79	1549.34	**1057.33**	**11543.13**
maxima	1071.47	1132.98	908.96	1656.06	1860.82	1045.14	1534.63	1644.01	1057.33	11543.13

COMPUTATION TIME

Heuristic	G1-144	G1-216	G1-72	G10-144	G10-216	G10-72	G7-144	G7-216	G7-72	Grand Total	Ratio
RN1	0.02	0.02	0.01	0.02	0.01	0.02	0.02	0.02	0.02	0.16	65441.3
RN2	0.02	0.02	0.01	0.02	0.02	0.02	0.02	0.02	0.02	0.17	67719.5
RN3	0.02	0.01	0.02	0.03	0.03	0.02	0.03	0.03	0.02	0.21	43720.2
RN4	0.05	0.03	0.02	0.06	0.07	0.04	0.03	0.08	0.04	0.42	25551.0
RN5	0.12	0.16	0.08	0.15	0.20	0.09	0.17	0.11	0.10	1.18	8916.8
RN6	0.33	0.39	0.12	0.34	0.22	0.13	0.41	0.55	0.29	2.78	3824.5
RN7	0.63	0.78	0.33	0.81	0.91	0.54	1.07	1.72	0.63	7.43	1371.1
RN8	1.98	2.47	1.20	2.52	2.72	1.50	3.33	4.58	1.69	22.00	502.0
RN9	8.43	12.19	4.22	9.00	12.91	4.34	12.61	18.12	6.05	87.86	127.9
RN10	29.41	42.89	16.37	32.24	48.47	18.05	38.65	62.32	22.02	310.42	36.9
RN11	75.86	91.75	55.19	110.30	170.53	61.08	149.27	218.75	89.65	1022.36	11.1
RN12	424.00	651.06	183.31	373.75	591.19	201.43	569.34	724.24	328.91	4047.23	2.8
RN13	1532.73	2574.68	723.26	1293.10	2021.41	665.31	2045.57	2334.60	1247.72	14438.38	0.8

It can be seen that for specific problem instances, the increase in the objective function (reward) does not increase monotonically with increasing look-ahead period. We note that RN2, especially, has as much success as higher look-ahead values. We can calculate the ratio of reward collected to computation time, to find the "bang-for-buck" values; we use this ratio as a measure of the efficiency of the heuristic.

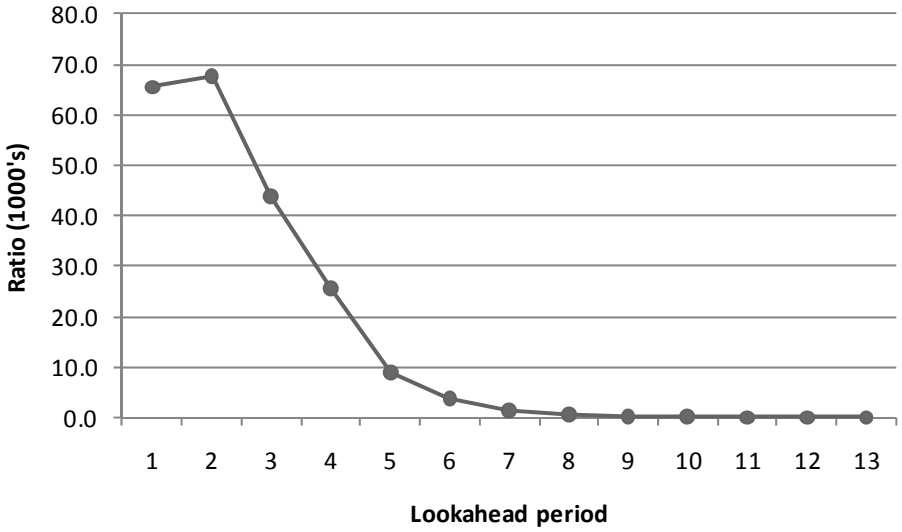


Figure 3.12. Ratio of reward to computation time (efficiency)

Figure 3.12 shows that both RN1 and RN2 have the highest efficiency, and since RN2 also has high performance (reward) this heuristic is used in the full experiments.

3.6.2 Tabu Search parameter values

There are three parameter values for the TABU SEARCH heuristic: the penalty for each unit of cost over the budget in intermediate solutions, the number of iterations that the heuristic runs for, and the tabu list size/tabu tenure. Initial experiments revealed that when the penalty was 0, there was no improvement over the initial constructive solution. They also revealed that an iteration counts of 50 and 100 were completely dominated by an iteration count of 200.

An experiment was performed on some problems from problem set B, using RICHEST NEIGHBOUR with look-ahead period 1 to construct the initial solution. Penalties of 10 and 100, iteration counts of 200 and 500, and tabu tenures of 2, 4, …, 20 were varied in a full factorial design, for a subset of instances. The results are displayed in Table 3.5. The *ghosted* values are those which are not improved from the initial solution, while those **highlighted** are the maximum achieved by any parameter specification.

As can be seen, the best results were achieved with 500 iteration, and a penalty of 10, with a tabu tenure of 16. This tabu tenure seems quite high; in the literature (for example Glover [119]) the optimal tabu tenure seems to be about 7. For the subsequent experiments in this chapter we will use TABU SEARCH with 500 iterations, a penalty of 10, and a tabu tenure of 16.

Table 3.5. Results from Tabu Search sensitivity analysis

Iterations	Penalty	Tenure	G1	G4	G7	G10	Grand Total
200	10	2	862.36	*748.74*	748.47	991.66	3351.23
		4	862.36	835.56	942.97	991.66	3632.55
		6	862.36	851.79	940.67	991.66	3646.48
		8	862.36	854.31	940.67	991.58	3648.91
		10	**873.04**	873.36	940.67	991.58	3678.65
		12	**873.04**	857.18	962.72	991.66	3684.60
		14	872.71	855.49	**974.19**	991.66	3694.05
		16	**873.04**	862.78	972.84	991.66	3700.32
		18	866.42	882.18	942.75	991.66	3683.00
		20	871.06	873.34	952.37	991.66	3688.43
	100	2	824.40	*748.74*	748.47	*863.61*	3185.22
		4	824.40	786.28	748.47	*863.61*	3222.76
		6	824.40	786.28	832.48	901.88	3345.04
		8	844.97	768.65	855.49	863.61	3332.72
		10	858.27	768.65	832.48	*863.61*	3323.01
		12	869.34	774.48	840.82	*863.61*	3348.25
		14	864.10	849.02	897.51	960.27	3570.89
		16	861.22	850.05	914.69	946.13	3572.09
		18	850.13	817.86	832.48	915.23	3415.70
		20	853.55	836.72	832.48	863.61	3386.36
500	10	2	862.36	*748.74*	748.47	991.66	3351.23
		4	862.36	835.56	942.97	991.66	3632.55
		6	862.36	854.88	940.67	991.66	3649.57
		8	862.36	854.31	940.67	991.58	3648.91
		10	**873.04**	873.36	940.67	991.58	3678.65
		12	**873.04**	857.18	962.72	991.66	3684.60
		14	872.71	867.00	**974.19**	991.66	3705.56
		16	**873.04**	**903.89**	973.86	991.66	**3742.46**
		18	866.42	882.18	955.99	991.66	3696.24
		20	871.06	873.34	958.55	991.66	3694.61
	100	2	824.40	*748.74*	748.47	*863.61*	3185.22
		4	824.40	786.28	748.47	*863.61*	3222.76
		6	824.40	786.28	832.48	901.88	3345.04
		8	844.97	768.65	855.49	*863.61*	3332.72
		10	858.27	768.65	832.48	*863.61*	3323.01
		12	869.34	774.48	840.82	*863.61*	3348.25
		14	864.10	849.02	897.51	960.27	3570.89
		16	861.22	850.05	914.69	982.32	3608.29
		18	850.13	817.86	832.48	915.23	3415.70
		20	866.02	836.72	832.48	**1110.66**	3645.88
		maxima	873.04	903.89	974.19	1110.66	3742.46
		available	1121.81	1408.32	1632.06	1917.71	6079.90
		RN1 value	820.36	748.74	728.86	863.61	3161.57

3.7 Phase 1 experimentation

The experiments in this phase were performed on problem set A, and the intention was to become familiar with the types of results that are obtained on the ASRP, and to examine a few features of the

generated problems. In phase 2 we perform a more extensive set of tournaments, with more heuristics and more problem instances.

3.7.1 Phase 1 heuristics

We compare the following 6 heuristics, described in Table 3.6. We consider three constructive heuristics, both unimproved and with a Tabu Search improvement phase.

Table 3.6. Heuristics used in phase 1experiments

Label	Description
RN2	RICHEST NEIGHBOUR construction with look-ahead 2 periods.
RN2-TS	RICHEST NEIGHBOUR-2 construction followed by TABU SEARCH improvement.
PR	PRUNE THEN ROUTE construction with Schedule 1
PR-TS	PRUNE THEN ROUTE-1 construction followed by TABU SEARCH improvement.
RP	ROUTE THEN PRUNE construction.
RP-TS	ROUTE THEN PRUNE construction followed by TABU SEARCH improvement.

3.7.2 Effect of depot location on complete grids

The next experiment involved examining the heuristics' performances on the complete grids. One of the points of interest was to determine the effect of depot location. In the tables below, the results from five problem instances were averaged to give the reward values. Table 3.7 gives the results for the unimproved heuristics.

Table 3.7: Results for unimproved heuristics on complete grids

	Budget														
Heuristic	**18**	**36**	**54**	**72**	**90**	**108**	**126**	**144**	**162**	**180**	**198**	**216**	**234**	**252**	**Average**
Corner Depot															
RN2	287	569	854	1146	1381	1621	1839	2061	2277	2417	2517	2630	2683	2692	**1784**
PR	161	410	626	908	1179	1446	1703	1901	2138	2381	2692	2692	2692	2692	**1687**
RP	55	465	688	1009	1371	1615	1886	2129	2382	2576	2692	2692	2692	2692	**1782**
Middle Depot															
RN2	295	576	880	1161	1413	1645	1878	2069	2220	2425	2567	2653	2690	2692	**1797**
PR	168	376	651	908	1178	1433	1706	1901	2138	2381	2692	2692	2692	2692	**1686**
RP	167	437	795	1080	1378	1640	1908	2158	2392	2589	2692	2692	2692	2692	**1808**
Edge Depot															
RN2	272	579	878	1131	1379	1637	1841	2066	2240	2371	2548	2641	2692	2692	**1783**
PR	166	364	633	901	1171	1456	1703	1901	2137	2381	2692	2692	2692	2692	**1684**
RP	149	479	766	1118	1370	1654	1897	2154	2405	2587	2692	2692	2692	2692	**1811**
All Complete Grids															
RN2	284	574	871	1146	1391	1634	1853	2065	2246	2404	2544	2641	2689	2692	**1788**
PR	165	383	637	906	1176	1445	1704	1901	2138	2381	2692	2692	2692	2692	**1686**
RP	124	460	750	1069	1373	1636	1897	2147	2393	2584	2692	2692	2692	2692	**1800**

The budget is partitioned into four intervals, based on which heuristic(s) found the best solution: RN2, RP, PR and RP, or all three. The divisions occur at approximately the same value of the cost budget for the three depot locations. Note that the second division must occur at the same place, since it marks the point where PR and RP find the optimum: a covering tour which requires no pruning, which for a

complete 10×10 grid requires 196 edge traversals. The first division occurs where RP becomes better than RN2, and the third division begins where RN2 attains the optimum.

The only difference between the depot locations was for the edge location. RN2 manages to find the optimum at a lower budget than for the middle and corner locations. The other difference is that RP takes prime position from RN2 at a lower budget for the edge location. We can probably attribute these differences to differing regions of reward density; they would likely reduce if the results of more realizations were averaged. Apart from these differences, it appears that depot location induces no systematic influence on the relative heuristic performances.

Applied to this data, for C approximately less than or equal to half the total grid cost, RN2 finds the best solutions. For higher values of C, RP provides solutions as good as or better than RN2 and PR. Overall, RP is the best unimproved heuristic. Figure 3.13 gives a graphical representation of the performances. It makes intuitive sense that RP will perform poorly, on average, for low budget problems. To produce a feasible solution the heuristic must delete many cycles; as the smaller cycles are deleted only large cycles remain to be deleted, so the resulting solution may not utilize all the available budget.

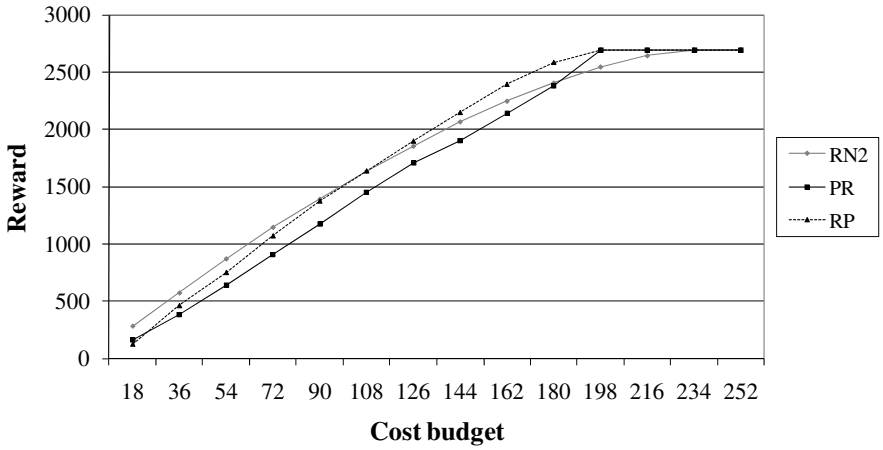


Figure 3.13: Results of constructive heuristics on complete grids with increasing budget

Table 3.8 gives the results for the improved heuristics. As can be seen, this version of Tabu Search is very sensitive to the initial solution. A similar phenomenon of regions may be seen, but they are not as definite as for the unimproved heuristics. In particular, when the depot is located at the edge of the grid, PR-TS does the best for mid range cost budgets. Also, the RN heuristic performs the best for higher values of C, although the difference in performance quality among the three improved heuristics is slight.

Table 3.8: Results for improved heuristics on complete grids

	Budget														
Heuristic	**18**	**36**	**54**	**72**	**90**	**108**	**126**	**144**	**162**	**180**	**198**	**216**	**234**	**252**	**Average**
Corner Depot															
RN2-TS	297	590	879	1178	1435	1693	1924	2165	2375	2534	2630	2692	2692	2692	**1841**
PR-TS	284	578	851	1145	1400	1698	1918	2094	2280	2463	2692	2692	2692	2692	**1820**
RP-TS	119	570	860	1133	1403	1656	1922	2169	2396	2576	2692	2692	2692	2692	**1827**
Middle Depot															
RN2-TS	297	594	889	1172	1436	1704	1940	2165	2333	2496	2622	2692	2692	2692	**1838**
PR-TS	279	562	855	1143	1406	1684	1919	2118	2286	2468	2692	2692	2692	2692	**1820**
RP-TS	283	572	865	1146	1418	1680	1939	2192	2421	2589	2692	2692	2692	2692	**1848**
Edge Depot															
RN2-TS	288	593	889	1156	1428	1692	1933	2166	2357	2502	2627	2686	2692	2692	**1836**
PR-TS	291	566	856	1133	1431	1693	1936	2139	2292	2481	2692	2692	2692	2692	**1828**
RP-TS	230	576	865	1146	1406	1672	1927	2180	2410	2587	2692	2692	2692	2692	**1841**
All Complete Grids															
RN2-TS	294	592	886	1169	1433	1696	1932	2166	2355	2511	2626	2690	2692	2692	**1838**
PR-TS	284	569	854	1140	1412	1692	1924	2117	2286	2471	2692	2692	2692	2692	**1823**
RP-TS	211	572	863	1142	1409	1670	1929	2180	2409	2584	2692	2692	2692	2692	**1838**

3.7.3 Results for random graphs

Table 3.9 summarizes the results summaries for the random graphs. Similar patterns to those with the complete grids are evident.

Table 3.9: Results for set A random graphs

	Budget														
Heuristic	**18**	**36**	**54**	**72**	**90**	**108**	**126**	**144**	**162**	**180**	**198**	**216**	**234**	**252**	**Average**
All 'GROW' Graphs															
RN2	233	460	680	872	1044	1169	1267	1368	1433	1473	1502	1512	1516	1517	**1146**
PR	171	377	628	848	1020	1182	1311	1414	1480	1512	1517	1517	1517	1517	**1143**
RP	99	414	654	873	1057	1226	1353	1439	1492	1514	1517	1517	1517	1517	**1156**
RN2-TS	244	498	750	957	1125	1263	1361	1434	1483	1507	1517	1517	1517	1517	**1192**
PR-TS	229	486	731	946	1116	1259	1364	1441	1493	1514	1517	1517	1517	1517	**1189**
RP-TS	162	475	729	940	1121	1270	1378	1448	1494	1514	1517	1517	1517	1517	**1186**
All 'SELECT' Graphs															
RN2	194	419	594	749	909	1070	1190	1309	1393	1450	1493	1515	1523	1524	**1095**
PR	147	330	545	731	947	1120	1275	1390	1479	1517	1524	1524	1524	1524	**1113**
RP	86	348	568	770	966	1142	1290	1416	1493	1520	1524	1524	1524	1524	**1121**
RN2-TS	208	448	656	837	1025	1189	1310	1412	1480	1512	1523	1524	1524	1524	**1155**
PR-TS	192	403	613	806	1018	1187	1324	1423	1490	1520	1524	1524	1524	1524	**1148**
RP-TS	153	435	654	848	1026	1189	1326	1429	1495	1520	1524	1524	1524	1524	**1155**
All Random Graphs ('GROW' and 'SELECT')															
RN2	213	440	637	810	976	1119	1228	1338	1413	1461	1498	1514	1520	1520	**1121**
PR	159	353	587	789	984	1151	1293	1402	1479	1514	1520	1520	1520	1520	**1128**
RP	93	381	611	822	1012	1184	1322	1428	1492	1517	1520	1520	1520	1520	**1139**
RN2-TS	226	473	703	897	1075	1226	1336	1423	1481	1510	1520	1520	1520	1520	**1174**
PR-TS	211	445	672	876	1067	1223	1344	1432	1491	1517	1520	1520	1520	1520	**1169**
RP-TS	158	455	691	894	1073	1230	1352	1439	1495	1517	1520	1520	1520	1520	**1170**

There is additional information in the breakdown by problem instance, as summarized by Figure 3.14, which shows where the divisions occur. The general trend is that the more edges in the graph, the later the division, with respect to cost budget.

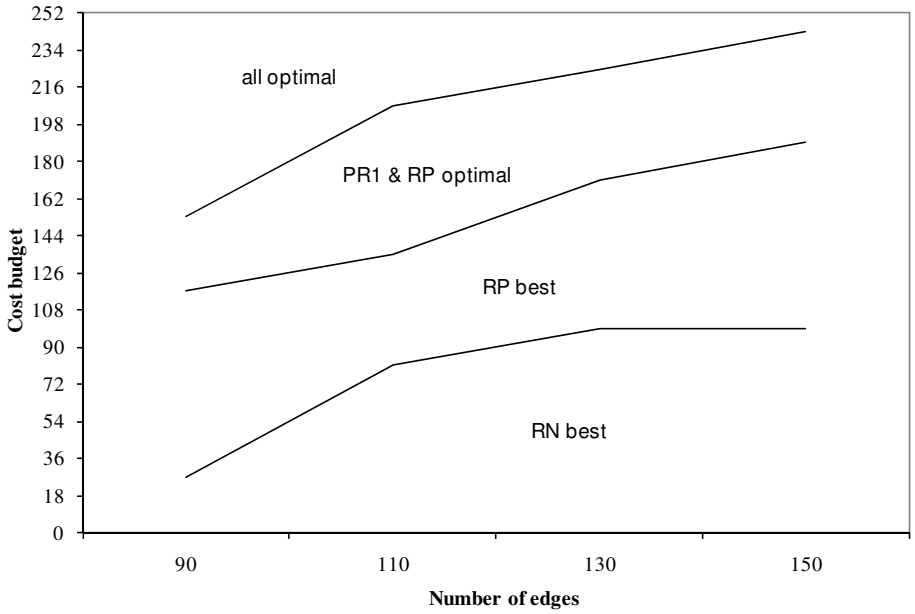


Figure 3.14: Approximate divisions of heuristic performance for unimproved heuristics

3.8 Phase 2 experimentation

The experiments in this phase are performed on problem set B, with more heuristics. The goal is to build on the results from phase 1.

3.8.1 Phase 2 heuristics

We compare the following 14 heuristics, described in Table 3.10. In addition to those examined in phase 1, we add an extra version of PRUNE THE ROUTE with schedule 3, we add an additional improvement procedure based on Steepest Ascent, and we compare versions of RN2 both with, and without, a DELETE REDUNDANCY step before the improvement procedures.

Table 3.10. Heuristics used in phase 2 experiments

Label	Description
RN2	RICHEST NEIGHBOUR construction with look-ahead 2 periods.
RN2-TS	RICHEST NEIGHBOUR-2 construction followed by TABU SEARCH improvement.
RN2-SA	RICHEST NEIGHBOUR-2 construction followed by STEEPEST ASCENT improvement.
RN2-DR-TS	RICHEST NEIGHBOUR-2 construction followed by DELETE REDUNDANCY and then TABU SEARCH improvement.
RN2-DR-SA	RICHEST NEIGHBOUR-2 construction followed by DELETE REDUNDANCY and then STEEPEST ASCENT improvement.
PR1	PRUNE THEN ROUTE construction with Schedule 1
PR1-TS	PRUNE THEN ROUTE-1 construction followed by TABU SEARCH improvement.
PR1-SA	PRUNE THEN ROUTE-1 construction followed by STEEPEST ASCENT improvement.
PR3	PRUNE THEN ROUTE construction with Schedule 3.
PR3-TS	PRUNE THEN ROUTE-3 construction followed by TABU SEARCH improvement.
PR3-SA	PRUNE THEN ROUTE-3 construction followed by STEEPEST ASCENT improvement.
RP	ROUTE THEN PRUNE construction.
RP-TS	ROUTE THEN PRUNE construction followed by TABU SEARCH improvement.
RP-SA	ROUTE THEN PRUNE construction followed by STEEPEST ASCENT improvement.

3.8.2 Results from Experiments on Designed Graphs

Experiments were executed comparing the 14 heuristics on the 21 designed problem instances. In the following results:

- The maximum value for a given instance is highlighted.
- If a heuristic collects all the reward available on the graph, then the value is **bold**ed.
- In the totals columns, the *constructive* (unimproved) heuristic with the highest value is *italicized*.

3.8.2.1 Designed graphs with C = 36

- RN2 was the best constructive heuristic overall and for each of the classes.
- The best heuristic overall was RN2-DR-TS. It was also the best heuristic for each of the classes.
- The Euler tour-based heuristics did not perform as well, in general, as the look-ahead heuristics.
- Several of the RP solutions were extremely bad (one was zero, corresponding to the whole route being deleted as a cycle).

Table 3.11. Results from experiments on designed graphs with C = 36

Heuristic	C1	C2	C3	L1	L2	L3	L4	L5	L6	L7	L8	L9
RN2	611.54	595.99	605.91	597.04	582.14	594.86	545.83	579.48	530.59	542.38	577.91	540.11
RN2-TS	634.67	595.99	619.79	597.04	582.14	609.82	563.05	586.97	548.60	542.38	597.61	559.08
RN2-SA	612.00	595.99	605.91	598.06	582.14	609.82	563.05	579.48	548.60	555.75	577.91	549.27
RN2-DR-TS	623.92	614.81	619.79	597.04	582.14	609.82	593.32	586.97	571.08	542.38	597.61	592.82
RN2-DR-SA	624.37	603.49	605.91	598.06	582.14	609.82	561.21	579.48	548.60	555.75	577.91	559.59
PR1	506.60	506.25	453.26	308.79	306.11	313.53	315.66	321.65	389.52	386.94	388.88	389.52
PR1-TS	595.94	591.23	597.70	579.53	549.37	555.87	585.84	591.18	561.99	593.40	568.79	548.15
PR1-SA	601.66	597.56	597.70	534.95	520.68	532.19	546.39	535.80	519.19	555.80	552.49	519.19
PR3	506.60	506.25	453.26	308.79	306.11	313.53	315.66	321.65	389.52	386.94	388.88	389.52
PR3-TS	595.94	591.23	597.70	579.53	549.37	555.87	585.84	591.18	561.99	593.40	568.79	548.15
PR3-SA	601.66	597.56	597.70	534.95	520.68	532.19	546.39	535.80	519.19	555.80	552.49	519.19
RP	523.95	575.84	565.81	72.88	523.87	514.80	0.00	566.80	17.78	66.25	255.95	61.90
RP-TS	577.05	613.99	599.69	571.46	539.95	537.33	582.99	572.11	540.16	582.99	584.15	566.24
RP-SA	572.34	621.33	589.02	504.19	539.95	523.94	531.62	572.11	536.68	531.62	542.89	498.84
maxima	634.67	621.33	619.79	598.06	582.14	609.82	593.32	591.18	571.08	593.40	597.61	592.82
available	2722.64	2722.64	2722.64	2374.51	2374.51	2374.51	2374.51	2374.51	2374.51	2374.51	2374.51	2374.51

Heuristic	R1	R2	R3	R4	R5	R6	R7	R8	R9	Total C	Total L	Total R	Grand Total
RN2	422.03	494.62	495.60	404.50	481.06	434.54	521.65	499.85	437.62	*1813.44*	*5090.33*	*4191.46*	*11095.23*
RN2-TS	422.97	498.69	495.60	404.50	481.06	445.36	521.65	499.85	437.62	1850.45	5186.69	4207.30	11244.43
RN2-SA	422.97	498.69	499.67	404.50	482.01	445.36	521.65	499.85	437.62	1813.90	5164.09	4212.32	11190.31
RN2-DR-TS	422.97	498.69	495.60	408.25	481.06	469.33	521.65	499.85	437.62	1858.51	5273.18	4235.02	11366.70
RN2-DR-SA	422.97	498.69	499.67	408.25	482.01	442.86	521.65	499.85	437.62	1833.78	5172.57	4213.57	11219.91
PR1	223.97	386.52	361.73	242.99	222.58	358.10	293.61	317.11	388.26	1466.11	3120.59	2794.87	7381.58
PR1-TS	372.08	461.01	430.97	410.60	365.91	435.35	417.59	441.03	460.31	1784.87	5134.12	3794.85	10713.84
PR1-SA	349.04	461.01	468.43	370.92	357.68	428.22	366.72	406.78	417.24	1796.92	4816.69	3626.02	10239.64
PR3	223.97	386.52	361.73	242.99	222.58	358.10	293.61	317.11	388.26	1466.11	3120.59	2794.87	7381.58
PR3-TS	372.08	461.01	430.97	410.60	365.91	435.35	417.59	441.03	460.31	1784.87	5134.12	3794.85	10713.84
PR3-SA	349.04	461.01	468.43	370.92	357.68	428.22	366.72	406.78	417.24	1796.92	4816.69	3626.02	10239.64
RP	356.59	332.36	358.58	358.11	432.86	260.52	345.00	347.82	278.14	1665.60	2080.22	3069.99	6815.81
RP-TS	409.27	425.70	445.36	368.22	467.75	454.23	440.60	373.31	469.99	1790.73	5077.37	3854.43	10722.53
RP-SA	404.14	395.81	429.04	377.62	455.70	440.62	406.23	373.31	429.73	1782.69	4781.84	3712.20	10276.74
maxima	422.97	498.69	499.67	410.60	482.01	469.33	521.65	499.85	469.99	1858.51	5273.18	4235.02	11366.70
available	1659.28	1659.28	1659.28	1659.28	1659.28	1659.28	1659.28	1659.28	1659.28	8167.92	21370.59	14933.52	44472.03

3.8.2.2 Designed graphs with C=72

- RN2 was the best constructive heuristic overall and for each of the classes.
- The best heuristic overall was RN2-DR-TS. It was also the best heuristic for classes L and R; the best heuristics for class C were RN2-TS and RN2-SA.
- The PR-TS heuristics found the best solution for a number of R problems; they were just behind the RN2 heuristics in effectiveness. The RP heuristics were not as effective for R problem instances, but found some good solutions for the L problem instances.

Table 3.12. Results from experiments on designed graphs with C = 72

Heuristic	C1	C2	C3	L1	L2	L3	L4	L5	L6	L7	L8	L9
RN2	1170.64	1217.19	1124.23	1150.34	1045.74	1162.22	1155.09	1122.75	1052.99	980.30	1080.64	1089.17
RN2-TS	1197.02	1226.57	1131.85	1180.02	1086.21	1196.90	1169.26	1164.14	1056.50	1041.73	1120.44	1107.82
RN2-SA	1197.02	1226.57	1131.85	1165.99	1045.74	1177.19	1169.26	1123.56	1056.50	990.88	1097.61	1089.17
RN2-DR-TS	1197.02	1226.57	1124.23	1180.02	1193.88	1209.34	1197.21	1163.32	1169.70	1023.00	1163.97	1139.03
RN2-DR-SA	1197.02	1226.57	1124.23	1165.99	1057.16	1177.19	1165.81	1135.20	1070.30	1011.71	1110.77	1089.17
PR1	922.18	972.50	972.50	699.24	745.54	745.54	702.04	747.01	745.54	751.51	747.81	745.54
PR1-TS	1160.28	1103.02	1067.86	1140.67	1165.03	1165.03	1073.27	1143.27	1137.22	1173.20	1121.47	1132.33
PR1-SA	1057.27	1061.59	1067.86	1037.72	1059.25	1059.25	1073.27	1060.41	1059.25	1110.94	1093.11	1059.25
PR3	922.18	972.50	972.50	699.24	745.54	745.54	702.04	747.01	745.54	751.51	747.81	745.54
PR3-TS	1160.28	1103.02	1067.86	1140.67	1165.03	1165.03	1073.27	1143.27	1137.22	1173.20	1121.47	1132.33
PR3-SA	1057.27	1061.59	1067.86	1037.72	1059.25	1059.25	1073.27	1060.41	1059.25	1110.94	1093.11	1059.25
RP	923.85	1132.14	1133.69	1074.84	788.13	1104.36	1075.49	777.27	1097.03	1137.79	1063.24	1127.72
RP-TS	1131.35	1188.12	1161.66	1142.13	1138.73	1139.66	1179.47	1148.39	1177.39	1150.86	1120.04	1167.13
RP-SA	1109.38	1188.12	1161.02	1110.65	1027.36	1112.70	1130.93	1083.66	1145.62	1160.83	1092.87	1153.02
maxima	1197.02	1226.57	1161.66	1180.02	1193.88	1209.34	1197.21	1164.14	1177.39	1173.20	1163.97	1167.13
available	2722.64	2722.64	2722.64	2374.51	2374.51	2374.51	2374.51	2374.51	2374.51	2374.51	2374.51	2374.51

Heuristic	R1	R2	R3	R4	R5	R6	R7	R8	R9	Total C	Total L	Total R	Grand Total
RN2	882.08	854.08	865.76	900.17	882.91	862.40	896.33	880.90	894.92	*3512.06*	*9839.25*	*7919.54*	*21270.84*
RN2-TS	905.12	903.70	903.39	911.07	882.91	900.09	896.33	904.10	894.92	3555.44	10123.00	8101.62	21780.06
RN2-SA	882.08	863.20	874.88	900.17	882.91	878.03	896.33	893.23	894.92	3555.44	9915.91	7965.74	21437.09
RN2-DR-TS	908.35	909.03	918.44	912.44	901.00	898.92	896.33	906.66	896.33	3547.82	10439.48	8147.50	22134.80
RN2-DR-SA	882.43	854.08	865.76	902.98	897.50	898.92	896.33	904.10	894.92	3547.82	9983.29	7997.02	21528.14
PR1	808.81	806.96	810.14	808.81	816.06	700.88	808.81	811.42	808.81	2867.19	6629.76	7180.72	16677.66
PR1-TS	912.55	905.42	905.04	915.56	920.54	781.13	897.83	891.34	883.53	3331.16	10251.48	8012.94	21595.58
PR1-SA	848.81	846.28	842.90	848.14	854.37	764.16	835.62	855.72	836.99	3186.71	9612.47	7533.00	20332.19
PR3	808.81	806.96	810.14	808.81	816.06	700.88	808.81	811.42	808.81	2867.19	6629.76	7180.72	16677.66
PR3-TS	912.55	905.42	905.04	915.56	920.54	781.13	897.83	891.34	883.53	3331.16	10251.48	8012.94	21595.58
PR3-SA	848.81	846.28	842.90	848.14	854.37	764.16	835.62	855.72	836.99	3186.71	9612.47	7533.00	20332.19
RP	768.19	746.29	768.62	769.03	588.14	788.40	775.58	796.37	806.02	3189.68	9245.86	6806.64	19242.19
RP-TS	812.65	857.12	851.57	779.14	914.12	864.11	846.83	826.72	870.37	3481.12	10363.80	7622.65	21467.57
RP-SA	805.24	805.51	833.37	788.54	839.81	838.67	813.62	816.89	836.83	3458.52	10017.65	7378.46	20854.63
maxima	912.55	909.03	918.44	915.56	920.54	900.09	897.83	906.66	896.33	3555.44	10439.48	8147.50	22134.80
available	1659.28	1659.28	1659.28	1659.28	1659.28	1659.28	1659.28	1659.28	1659.28	8167.92	21370.59	14933.52	44472.03

3.8.2.3 Designed graphs with C = 108

- RP was the best constructive heuristic overall and for classes L and R; RN2 was the best constructive heuristic for class C.
- The best heuristic overall was RP-TS. It was also the best heuristic for class L; the best heuristic for classes C and R was RN2-DR-TS.
- The PR heuristics were not as effective as the RN2 and RP heuristics. The RN2 and RP heuristics both found a high number of best solutions.

Table 3.13. Results from experiments on designed graphs with C = 108

Heuristic	C1	C2	C3	L1	L2	L3	L4	L5	L6	L7	L8	L9
RN2	1657.07	1710.04	1598.26	1494.93	1538.00	1522.12	1528.92	1611.77	1604.54	1557.88	1637.92	1580.97
RN2-TS	1683.45	1727.58	1680.11	1511.98	1563.46	1543.87	1605.98	1691.96	1693.55	1618.54	1662.02	1701.44
RN2-SA	1683.45	1727.58	1611.63	1494.93	1538.00	1526.44	1532.38	1622.62	1647.39	1557.88	1637.92	1581.79
RN2-DR-TS	1739.96	1733.98	1654.31	1507.07	1541.20	1547.84	1658.20	1686.76	1700.92	1689.55	1691.51	1675.56
RN2-DR-SA	1689.55	1710.04	1598.26	1507.07	1541.20	1525.63	1530.37	1634.26	1647.39	1577.69	1637.92	1600.42
PR1	1456.31	1456.31	1456.31	1493.72	1493.72	1493.72	1493.72	1496.62	1493.72	1493.72	1493.72	1493.72
PR1-TS	1609.36	1620.64	1575.10	1647.73	1583.14	1583.14	1663.23	1648.70	1583.14	1644.55	1658.48	1650.82
PR1-SA	1595.14	1568.83	1575.10	1567.71	1583.14	1583.14	1587.07	1578.90	1583.14	1580.49	1567.71	1563.47
PR3	1456.31	1456.31	1456.31	1493.72	1493.72	1493.72	1493.72	1496.62	1493.72	1493.72	1493.72	1493.72
PR3-TS	1609.36	1620.64	1575.10	1647.73	1583.14	1583.14	1663.23	1648.70	1583.14	1644.55	1658.48	1650.82
PR3-SA	1595.14	1568.83	1575.10	1567.71	1583.14	1583.14	1587.07	1578.90	1583.14	1580.49	1567.71	1563.47
RP	1637.85	1635.27	1678.83	1573.85	1593.74	1655.06	1608.63	1637.99	1587.68	1668.58	1605.04	1637.28
RP-TS	1676.56	1714.66	1721.13	1687.11	1658.94	1666.97	1646.73	1653.28	1611.56	1689.97	1627.70	1671.32
RP-SA	1676.56	1709.31	1720.50	1633.29	1616.97	1655.06	1620.09	1640.70	1608.41	1672.87	1627.70	1657.22
maxima	1739.96	1733.98	1721.13	1687.11	1658.94	1666.97	1663.23	1691.96	1700.92	1689.97	1691.51	1701.44
available	2722.64	2722.64	2722.64	2374.51	2374.51	2374.51	2374.51	2374.51	2374.51	2374.51	2374.51	2374.51

Heuristic	R1	R2	R3	R4	R5	R6	R7	R8	R9	Total C	Total L	Total R	Grand Total
RN2	1219.90	1203.88	1153.79	1193.51	1175.44	1146.17	1126.62	1175.62	1121.13	*4965.37*	14077.05	10516.05	29558.48
RN2-TS	1223.16	1203.88	1215.06	1223.02	1241.39	1169.69	1145.60	1227.66	1131.89	5091.15	14592.80	10781.34	30465.28
RN2-SA	1219.90	1203.88	1179.18	1193.51	1175.44	1156.93	1126.62	1186.38	1131.89	5022.66	14139.35	10573.72	29735.72
RN2-DR-TS	1229.17	1224.38	1216.60	1226.71	1211.78	1209.81	1143.99	1183.15	1159.03	5128.25	14698.62	10804.62	30631.48
RN2-DR-SA	1219.90	1203.88	1183.54	1216.15	1175.44	1158.66	1136.53	1183.15	1159.03	4997.85	14201.94	10636.28	29836.08
PR1	1096.47	1096.47	1097.33	1096.47	1103.25	1096.47	1096.47	1103.05	1096.47	4368.92	13446.38	9882.47	27697.77
PR1-TS	1140.57	1212.32	1209.89	1133.31	1215.26	1197.76	1140.57	1218.73	1187.52	4805.10	14662.92	10655.94	30123.96
PR1-SA	1140.57	1153.23	1160.83	1133.31	1157.13	1153.45	1140.57	1147.15	1155.69	4739.07	14194.76	10341.94	29275.76
PR3	1096.47	1096.47	1097.33	1096.47	1103.25	1096.47	1096.47	1103.05	1096.47	4368.92	13446.38	9882.47	27697.77
PR3-TS	1140.57	1212.32	1209.89	1133.31	1215.26	1197.76	1140.57	1218.73	1187.52	4805.10	14662.92	10655.94	30123.96
PR3-SA	1140.57	1153.23	1160.83	1133.31	1157.13	1153.45	1140.57	1147.15	1155.69	4739.07	14194.76	10341.94	29275.76
RP	1132.59	1170.09	1173.39	1171.45	1156.19	1178.93	1160.02	1188.46	1193.61	4951.95	*14567.84*	*10524.72*	*30044.51*
RP-TS	1186.12	1179.44	1201.96	1177.10	1191.11	1214.61	1204.14	1211.31	1214.78	5112.35	14913.57	10780.56	30806.48
RP-SA	1154.74	1176.67	1188.13	1177.10	1183.72	1194.47	1195.77	1188.46	1214.78	5106.37	14732.30	10673.84	30512.51
maxima	1229.17	1224.38	1216.60	1226.71	1241.39	1214.61	1204.14	1227.66	1214.78	5128.25	14913.57	10804.62	30806.48
available	1659.28	1659.28	1659.28	1659.28	1659.28	1659.28	1659.28	1659.28	1659.28	8167.92	21370.59	14933.52	44472.03

3.8.2.4 Designed graphs with C = 144

- RP was the best constructive heuristic overall and for each of the classes.
- The best heuristic overall was RP-TS. It was also the best heuristic for each of the classes.
- Neither the RN2 heuristics or the PR heuristics performed as well as the RP heuristics, but RN2-DR-TS did produce the best solution for three of the problem instances.
- For many of the problem instances, RP produced the best solution and TS and SA could not improve it.

Table 3.14. Results from experiments on designed graphs with C = 144

Heuristic	C1	C2	C3	L1	L2	L3	L4	L5	L6	L7	L8	L9
RN2	2120.55	2064.29	2067.77	1946.10	1862.26	1904.34	2008.56	1953.21	2042.96	1751.12	1935.92	1931.85
RN2-TS	2175.58	2088.74	2101.92	1982.28	1898.07	1985.55	2075.09	1980.71	2071.00	1801.02	1940.36	1998.50
RN2-SA	2151.96	2064.29	2125.99	1951.02	1878.28	1933.34	2031.30	1963.24	2071.00	1751.12	1940.36	1982.09
RN2-DR-TS	2218.17	2204.89	2096.68	2038.22	1957.19	1971.92	2086.84	1980.71	2113.90	1842.02	1938.78	2079.06
RN2-DR-SA	2151.96	2115.17	2067.77	1965.53	1938.67	1971.92	2022.57	1980.71	2091.11	1842.02	1938.78	2018.95
PR1	1882.26	1882.26	1882.26	1920.08	1920.08	1920.08	1920.08	1920.08	1920.08	1920.08	1920.08	1920.08
PR1-TS	1997.02	2140.26	2144.99	2057.97	2035.67	2012.50	2007.18	2038.82	2083.08	2010.63	2016.80	2038.91
PR1-SA	1967.24	1986.35	1986.35	1993.69	1993.69	1993.69	2007.18	2000.49	1993.69	2010.63	2016.80	2021.41
PR3	1882.26	1882.26	1882.26	1920.08	1920.08	1920.08	1920.08	1920.08	1920.08	1920.08	1920.08	1920.08
PR3-TS	1997.02	2140.26	2144.99	2057.97	2035.67	2012.50	2007.18	2038.82	2083.08	2010.63	2016.80	2038.91
PR3-SA	1967.24	1986.35	1986.35	1993.69	1993.69	1993.69	2007.18	2000.49	1993.69	2010.63	2016.80	2021.41
RP	2171.63	2188.99	2221.18	2058.48	2044.52	2076.05	2067.07	2104.96	2070.43	2118.94	2046.81	2070.43
RP-TS	2193.78	2219.66	2221.76	2101.58	2068.64	2095.99	2067.07	2104.96	2092.52	2118.94	2085.59	2091.08
RP-SA	2193.78	2219.66	2221.18	2058.48	2068.64	2095.99	2067.07	2104.96	2091.08	2118.94	2085.59	2091.08
maxima	2218.17	2219.66	2221.76	2101.58	2068.64	2095.99	2086.84	2104.96	2113.90	2118.94	2085.59	2091.08
available	2722.64	2722.64	2722.64	2374.51	2374.51	2374.51	2374.51	2374.51	2374.51	2374.51	2374.51	2374.51

Heuristic	R1	R2	R3	R4	R5	R6	R7	R8	R9	Total C	Total L	Total R	Grand Total
RN2	1337.82	1265.53	1377.65	1332.11	1329.31	1377.45	1398.29	1279.45	1381.43	6252.61	17336.33	12079.04	35667.98
RN2-TS	1370.94	1265.53	1394.04	1332.11	1362.43	1377.45	1398.29	1302.33	1381.43	6366.25	17732.58	12184.56	36283.39
RN2-SA	1337.82	1265.53	1379.78	1332.11	1329.31	1377.45	1398.29	1302.33	1381.43	6342.24	17501.75	12104.05	35948.04
RN2-DR-TS	1356.04	1412.58	1422.17	1420.10	1362.43	1436.63	1416.51	1336.10	1399.33	6519.75	18008.63	12561.89	37090.28
RN2-DR-SA	1356.04	1397.94	1422.17	1420.10	1362.43	1421.06	1416.51	1317.55	1396.33	6334.90	17770.25	12510.14	36615.29
PR1	1394.51	1394.51	1394.51	1394.51	1394.51	1394.51	1394.51	1379.63	1394.51	5646.78	17280.73	12535.67	35463.19
PR1-TS	1441.55	1399.73	1399.73	1420.16	1399.73	1404.98	1433.55	1428.43	1437.93	6282.27	18301.57	12765.78	37349.62
PR1-SA	1399.73	1399.73	1399.73	1399.73	1399.73	1404.98	1399.73	1384.85	1401.97	5939.94	18031.29	12590.15	36561.39
PR3	1378.87	1378.87	1378.87	1378.87	1378.87	1378.87	1378.87	1378.87	1378.87	5646.78	17280.73	12409.83	35337.34
PR3-TS	1418.56	1402.35	1402.35	1437.83	1407.17	1479.26	1451.52	1412.09	1426.82	6282.27	18301.57	12837.95	37421.79
PR3-SA	1401.67	1402.35	1402.35	1407.28	1407.17	1438.14	1412.09	1412.09	1421.29	5939.94	18031.29	12704.44	36675.68
RP	1473.90	1502.84	1507.53	1494.61	1516.63	1513.07	1521.52	1487.51	1509.62	*6581.80*	*18657.69*	*13527.22*	*38766.71*
RP-TS	1487.75	1502.84	1507.53	1497.50	1516.63	1513.07	1521.52	1519.31	1509.62	6635.20	18826.36	13575.76	39037.32
RP-SA	1487.75	1502.84	1507.53	1497.50	1516.63	1513.07	1521.52	1519.31	1509.62	6634.62	18781.82	13575.76	38992.20
maxima	1487.75	1502.84	1507.53	1497.50	1516.63	1513.07	1521.52	1519.31	1509.62	6635.20	18826.36	13575.76	39037.32
available	1659.28	1659.28	1659.28	1659.28	1659.28	1659.28	1659.28	1659.28	1659.28	8167.92	21370.59	14933.52	44472.03

3.8.2.5 Designed graphs with C = 180

- RP was the best constructive heuristic overall and for each of the classes.
- The best heuristics overall were all three RP heuristics. They were also the best heuristics for each of the classes; and the improved PR heuristics also produced the best solutions for classes L and R.
- Neither the RN2 heuristics or the PR heuristics performed as well as the RP heuristics for class C, but the PR heuristics performed as well as the RP heuristics for classes L and R.
- For many of the problem instances, RP produced the best solution and TS and SA could not improve it.
- The RP heuristics and the improved PR heuristics managed to find many solutions that collected **all** the reward available on the graph. In this case the cost of the CPP relaxation was less than the budget, so could be solved using the CPP algorithm.

Table 3.15. Results from experiments on designed graphs with C = 180

Heuristic	C1	C2	C3	L1	L2	L3	L4	L5	L6	L7	L8	L9
RN2	2492.09	2393.85	2430.73	2192.16	2165.27	2201.31	2165.70	2235.40	2280.00	1944.85	2206.42	2195.66
RN2-TS	2515.33	2393.85	2430.73	2234.69	2181.28	2246.99	2192.48	2245.44	2308.72	2054.62	2273.71	2196.66
RN2-SA	2492.09	2393.85	2430.73	2204.73	2181.28	2229.52	2192.48	2245.44	2290.03	1944.85	2210.85	2196.66
RN2-DR-TS	2538.82	2411.53	2463.64	2290.47	2291.29	2245.53	2246.31	2275.26	2342.68	2183.61	2279.03	2259.52
RN2-DR-SA	2515.33	2411.53	2430.73	2247.75	2291.29	2245.53	2228.85	2262.90	2342.68	2137.11	2243.45	2245.51
PR1	2362.11	2362.11	2362.11	2364.50	2364.50	2364.50	2364.50	2364.50	2364.50	2364.50	2364.50	2364.50
PR1-TS	2474.15	2456.47	2396.39	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**
PR1-SA	2399.94	2430.17	2386.23	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**
PR3	2362.11	2362.11	2362.11	2364.50	2364.50	2364.50	2364.50	2364.50	2364.50	2364.50	2364.50	2364.50
PR3-TS	2474.15	2456.47	2396.39	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**
PR3-SA	2399.94	2430.17	2386.23	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**
RP	2602.86	2620.77	2620.77	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**
RP-TS	2602.86	2620.77	2620.77	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**
RP-SA	2602.86	2620.77	2620.77	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**	**2374.51**
maxima	2602.86	2620.77	2620.77	2374.51	2374.51	2374.51	2374.51	2374.51	2374.51	2374.51	2374.51	2374.51
available	2722.64	2722.64	2722.64	2374.51	2374.51	2374.51	2374.51	2374.51	2374.51	2374.51	2374.51	2374.51

Heuristic	R1	R2	R3	R4	R5	R6	R7	R8	R9	Total C	Total L	Total R	Grand Total
RN2	1459.45	1452.92	1498.21	1459.45	1446.26	1510.43	1445.83	1563.89	1547.04	7316.67	19586.78	13383.49	40286.94
RN2-TS	1477.67	1467.82	1527.46	1477.67	1470.83	1543.53	1464.05	1613.19	1547.04	7339.91	19934.59	13589.27	40863.77
RN2-SA	1459.45	1452.92	1498.21	1459.45	1459.45	1532.77	1453.18	1573.11	1547.04	7316.67	19695.85	13435.59	40448.11
RN2-DR-TS	1559.54	1609.34	1579.68	1541.40	1582.29	1618.39	1599.20	1642.08	1628.89	7413.99	20413.71	14360.81	42188.52
RN2-DR-SA	1559.54	1609.34	1555.16	1541.40	1582.29	1618.39	1599.20	1628.89	1614.79	7357.59	20245.07	14309.00	41911.66
PR1	1654.06	1654.06	1654.06	1654.06	1654.06	1654.06	1654.06	1654.06	1654.06	7086.34	21280.50	14886.58	43253.43
PR1-TS	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	7327.01	**21370.58**	**14933.56**	43631.15
PR1-SA	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	7216.35	**21370.58**	**14933.56**	43520.48
PR3	1654.06	1654.06	1654.06	1654.06	1654.06	1654.06	1654.06	1654.06	1654.06	7086.34	21280.50	14886.58	43253.43
PR3-TS	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	7327.01	**21370.58**	**14933.56**	43631.15
PR3-SA	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	7216.35	**21370.58**	**14933.56**	43520.48
RP	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	*7844.40*	***21370.57***	***14933.56***	*44148.53*
RP-TS	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	7844.40	**21370.57**	**14933.56**	44148.53
RP-SA	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	**1659.28**	7844.40	**21370.58**	**14933.56**	44148.53
maxima	1659.28	1659.28	1659.28	1659.28	1659.28	1659.28	1659.28	1659.28	1659.28	7844.40	21370.58	14933.56	44148.53
available	1659.28	1659.28	1659.28	1659.28	1659.28	1659.28	1659.28	1659.28	1659.28	8167.92	21370.59	14933.52	44472.03

3.8.3 Results from Experiments on Random Graphs

Experiments were executed comparing the 14 heuristics on the 24 random problem instances. In the following results:

- The maximum value for a given instance is highlighted.
- If a heuristic collects all the reward available on the graph, then the value is **bold**ed.
- In the totals columns, the constructive (unimproved) heuristic with the highest value is *italicized*.

3.8.3.1 Random graphs with C = 36

- RN2 was the best constructive heuristic overall and for both of the classes.
- The best heuristic overall, and for each of the classes, was RP-TS.
- All three heuristic types (RN2, PR, RP) achieved the best solution on some problem instances.
- For some problem instances, RP performed very poorly (zero for S12).

Table 3.16. Results from experiments on random graphs with C = 36

Heuristic	G1	G2	G3	G4	G5	G6	G7	G8	G9	G10	G11	G12	Total
RN2	405.64	583.09	344.51	503.34	484.92	405.25	463.52	503.09	422.08	518.49	555.68	349.41	*5539.03*
RN2-TS	445.42	622.12	398.66	503.76	484.92	410.79	466.33	503.09	436.00	518.49	564.48	349.41	5703.47
RN2-SA	405.64	598.56	349.87	503.34	484.92	405.25	505.27	503.09	436.00	518.49	555.68	349.41	5615.52
RN2-DR-TS	448.77	622.12	398.66	503.76	529.09	420.10	530.35	518.57	458.93	518.49	590.04	372.95	5911.84
RN2-DR-SA	425.31	622.12	349.87	503.34	504.08	405.25	464.71	512.82	431.42	518.49	590.04	365.22	5692.66
PR1	426.16	524.80	299.63	354.08	431.85	303.70	377.41	341.88	321.28	365.04	365.32	307.08	4418.22
PR1-TS	474.33	577.83	381.10	482.08	468.95	431.93	502.80	394.91	442.05	524.83	550.34	434.56	5665.71
PR1-SA	441.31	543.65	326.87	445.64	468.95	431.93	464.01	394.91	358.59	507.85	547.22	392.59	5323.51
PR3	282.11	524.80	234.14	354.08	431.85	318.04	365.83	341.88	261.04	365.04	365.32	247.95	4092.06
PR3-TS	392.87	577.83	314.82	482.08	468.95	425.03	511.54	394.91	425.83	524.83	550.34	395.96	5464.99
PR3-SA	322.98	543.65	303.68	445.64	468.95	396.04	499.13	394.91	401.33	507.85	547.22	387.19	5218.55
RP	437.02	502.84	371.17	226.18	482.81	404.30	502.68	453.42	312.42	479.70	462.91	329.25	4964.71
RP-TS	489.70	544.41	425.32	447.19	539.94	413.98	519.44	509.65	458.11	518.31	589.80	455.21	5911.04
RP-SA	481.05	544.41	397.01	405.12	527.10	413.98	519.44	509.65	420.97	518.31	557.43	383.85	5678.30
maxima	489.70	622.12	425.32	503.76	539.94	431.93	530.35	518.57	458.93	524.83	590.04	455.21	5911.84
available	1121.81	1307.16	954.24	1408.32	1602.98	1199.33	1632.06	1881.29	1404.24	1917.71	2189.75	1627.33	18246.22

Heuristic	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	Total	Grand Total
RN2	277.77	273.97	288.24	392.14	397.46	283.16	413.78	405.83	384.43	529.53	622.92	361.47	*4630.69*	*10169.72*
RN2-TS	346.80	273.97	317.29	413.17	408.95	348.37	415.32	485.27	399.10	543.96	622.92	424.78	4999.89	10703.36
RN2-SA	287.74	273.97	288.24	392.14	397.46	284.09	424.67	405.83	391.33	543.96	622.92	361.47	4673.82	10289.34
RN2-DR-TS	346.80	273.97	317.29	413.17	399.76	360.17	423.79	484.51	428.87	543.96	635.94	396.01	5024.24	10936.08
RN2-DR-SA	295.86	273.97	288.24	392.14	397.46	319.98	424.67	425.35	391.45	543.96	622.92	378.49	4754.48	10447.15
PR1	228.63	332.93	243.87	278.83	314.39	247.82	332.24	415.36	240.52	308.31	526.79	278.96	3748.64	8166.86
PR1-TS	290.38	333.10	243.87	325.80	399.89	339.96	373.17	487.20	280.83	469.60	554.98	434.19	4532.97	10198.68
PR1-SA	236.37	332.93	243.87	288.50	314.39	299.25	373.17	474.03	280.83	458.58	542.48	333.46	4177.86	9501.37
PR3	220.66	286.79	202.61	307.95	316.20	219.39	314.50	415.36	203.75	308.31	526.79	237.27	3559.58	7651.64
PR3-TS	290.38	330.74	260.03	323.61	399.89	358.09	413.23	487.20	252.74	469.60	554.98	349.50	4489.99	9954.98
PR3-SA	236.37	296.98	202.61	307.95	316.20	299.25	366.13	474.03	230.38	458.58	542.48	329.55	4060.51	9279.06
RP	338.99	333.10	303.86	401.74	524.09	39.43	361.07	78.02	312.41	423.03	564.79	0.00	3680.51	8645.22
RP-TS	373.16	333.10	336.28	415.41	524.09	371.33	433.96	444.23	384.72	446.58	604.94	428.30	5096.09	11007.13
RP-SA	338.99	333.10	303.86	401.74	524.09	319.93	433.96	399.30	365.54	431.86	606.33	364.06	4822.77	10501.06
maxima	373.16	333.10	336.28	415.41	524.09	371.33	433.96	487.20	428.87	543.96	635.94	434.19	5096.09	11007.13
available	1152.08	1319.15	975.13	1416.69	1628.88	1182.72	1615.60	1886.02	1381.33	1910.98	2204.97	1587.33	18260.88	35369.68

3.8.3.2 Random graphs with C = 72

- RP was the best constructive heuristic overall and for class S; RN2 was the best constructive heuristic for class G.
- The best heuristic overall, and for class S, was RP-TS; for class G the best heuristic was PR1-TS.
- All three heuristic types (RN2, PR, RP) achieved the best solution on some problem instances.

Table 3.17. Results from experiments on random graphs with C = 72

Heuristic	G1	G2	G3	G4	G5	G6	G7	G8	G9	G10	G11	G12	Total
RN2	828.96	985.38	690.46	773.12	889.47	763.99	911.90	1038.43	701.25	1016.87	1072.07	772.57	*10444.47*
RN2-TS	864.60	1020.50	704.85	911.10	1030.85	867.67	944.20	1094.55	701.25	1016.87	1072.07	773.48	11001.98
RN2-SA	828.96	985.38	695.82	789.22	889.47	785.03	911.90	1038.43	701.25	1016.87	1072.07	773.48	10487.88
RN2-DR-TS	872.68	1030.30	704.85	927.25	926.38	868.19	997.29	1108.28	755.83	1040.73	1111.25	881.86	11224.87
RN2-DR-SA	830.95	1004.65	695.82	792.03	926.29	786.76	911.90	1038.43	719.28	1040.73	1101.40	793.15	10641.39
PR1	790.19	997.55	653.39	843.93	956.49	732.02	883.61	908.87	745.01	951.58	1001.46	702.72	10166.83
PR1-TS	858.29	1020.29	698.26	929.88	1019.77	828.02	999.61	1022.07	845.58	1091.71	1067.04	855.60	11236.11
PR1-SA	793.21	997.55	662.59	896.88	970.69	761.10	959.77	1002.98	756.56	1007.13	1067.04	783.09	10658.59
PR3	647.18	931.86	474.64	753.24	919.39	578.85	742.04	908.87	551.91	883.21	1001.46	513.27	8905.91
PR3-TS	776.75	1027.46	662.59	921.90	946.21	828.85	975.99	1022.07	713.37	1007.79	1067.04	832.88	10782.89
PR3-SA	695.97	1005.93	612.51	832.58	935.85	732.24	886.79	1002.98	713.37	1007.79	1067.04	783.84	10276.89
RP	821.39	969.20	510.06	773.48	910.81	763.99	919.02	1065.18	747.78	1017.99	1072.52	774.03	10345.46
RP-TS	863.88	1005.46	663.75	892.64	1030.54	820.04	1013.45	1069.05	858.46	1042.64	1120.00	843.73	11223.64
RP-SA	857.48	1004.51	659.30	877.31	958.07	788.64	959.68	1069.05	807.42	1027.31	1109.75	818.22	10936.72
maxima	872.68	1030.30	704.85	929.88	1030.85	868.19	1013.45	1108.28	858.46	1091.71	1120.00	881.86	11236.11
available	1121.81	1307.16	954.24	1408.32	1602.98	1199.33	1632.06	1881.29	1404.24	1917.71	2189.75	1627.33	18246.22

Heuristic	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	Total	Grand Total
RN2	571.47	550.59	521.68	500.29	918.85	599.63	616.83	935.38	610.24	957.36	971.32	702.20	8455.85	18900.32
RN2-TS	625.28	634.13	534.66	693.34	960.79	608.02	616.83	983.53	741.01	971.80	1022.49	702.20	9094.09	20096.07
RN2-SA	581.44	550.59	521.68	513.89	929.00	608.02	616.83	937.50	617.95	971.80	971.32	702.20	8522.23	19010.12
RN2-DR-TS	627.51	550.59	534.66	775.23	960.79	681.09	671.30	982.77	756.91	1039.36	1116.92	754.63	9451.75	20676.62
RN2-DR-SA	590.32	550.59	521.68	722.48	960.79	659.60	671.30	937.50	633.19	971.80	971.32	745.08	8935.65	19577.04
PR1	617.19	601.16	550.04	743.83	812.45	504.86	653.03	912.92	557.96	802.89	1062.43	521.35	8340.11	18506.94
PR1-TS	656.10	633.96	577.31	800.89	885.59	590.10	750.99	962.70	640.45	985.78	1082.32	765.73	9331.91	20568.02
PR1-SA	617.19	601.16	550.04	749.92	857.62	578.07	706.79	940.92	623.59	942.08	1082.32	665.49	8915.18	19573.77
PR3	500.89	577.27	437.68	551.80	601.22	449.66	596.79	818.62	460.28	776.70	1062.43	461.69	7295.02	16200.92
PR3-TS	564.37	633.96	484.93	685.60	725.21	571.28	774.32	925.90	629.28	935.80	1082.32	732.64	8745.61	19528.51
PR3-SA	515.95	601.16	445.23	583.98	622.75	557.21	731.15	883.93	529.33	898.88	1082.32	650.58	8102.45	18379.35
RP	604.06	596.04	536.23	809.35	928.50	693.87	906.76	960.20	688.06	864.07	1043.23	682.89	*9313.25*	*19658.71*
RP-TS	648.93	634.29	585.13	837.95	952.76	734.95	950.50	1058.78	757.71	907.94	1102.37	859.68	10030.99	21254.63
RP-SA	604.06	596.04	536.23	819.90	928.50	723.67	929.28	999.02	723.06	907.94	1081.05	773.93	9622.69	20559.41
maxima	656.10	634.29	585.13	837.95	960.79	734.95	950.50	1058.78	757.71	1039.36	1116.92	859.68	10030.99	21254.63
available	1152.08	1319.15	975.13	1416.69	1628.88	1182.72	1615.60	1886.02	1381.33	1910.98	2204.97	1587.33	18260.88	35369.68

3.8.3.3 Random graphs with C = 108

- RP was the best constructive heuristic overall and for both classes.
- The best heuristic overall, and for both classes was RP-TS.
- All three heuristic types (RN2, PR, RP) achieved the best solution on some problem instances.
- On G1, G2, and G3, the RP heuristics dominate. These graphs have 90 edges.

Table 3.18. Results from experiments on random graphs with C = 108

Heuristic	G1	G2	G3	G4	G5	G6	G7	G8	G9	G10	G11	G12	Total
RN2	1046.47	1156.05	782.74	932.75	1313.00	1000.27	1335.03	1425.12	1022.30	1400.84	1523.53	1171.21	14109.30
RN2-TS	1063.78	1177.18	788.10	949.28	1351.21	1033.93	1367.33	1433.27	1022.30	1474.03	1628.08	1173.30	14461.80
RN2-SA	1046.47	1156.05	788.10	932.75	1313.00	1028.53	1335.03	1433.27	1022.30	1412.93	1523.53	1173.30	14165.26
RN2-DR-TS	1079.17	1260.04	846.91	1175.59	1415.23	1039.10	1375.54	1489.21	1041.19	1481.31	1571.52	1187.94	14962.75
RN2-DR-SA	1063.78	1234.22	846.91	1115.12	1369.22	1038.77	1335.03	1453.66	1031.92	1443.91	1525.04	1187.19	14644.77
PR1	1080.27	1284.32	884.30	1183.16	1289.30	1043.39	1205.85	1321.99	1091.22	1331.93	1491.54	1076.63	14283.89
PR1-TS	1089.81	1284.32	884.30	1193.06	1305.48	1095.13	1257.51	1428.53	1142.24	1409.05	1623.39	1197.60	14910.42
PR1-SA	1080.27	1284.32	884.30	1183.16	1289.30	1064.74	1257.51	1355.76	1118.84	1374.70	1519.35	1099.99	14512.24
PR3	1090.59	1284.32	884.30	1103.16	1232.80	1054.16	1012.80	1305.36	969.59	1276.62	1368.57	812.30	13394.57
PR3-TS	1090.59	1284.32	884.30	1203.33	1303.10	1059.50	1199.23	1423.61	1087.31	1497.19	1409.93	1040.31	14482.73
PR3-SA	1090.59	1284.32	884.30	1165.19	1288.54	1059.50	1199.23	1328.32	1080.44	1416.02	1409.93	1052.22	14258.60
RP	1091.75	1286.43	886.86	1202.32	1365.31	1070.18	1322.01	1470.51	1123.96	1379.93	1544.10	1120.27	*14863.64*
RP-TS	1091.88	1286.43	886.86	1213.65	1386.09	1087.37	1360.29	1486.86	1166.41	1438.85	1600.47	1222.12	15227.29
RP-SA	1091.75	1286.43	886.86	1202.32	1375.77	1070.18	1348.48	1470.51	1125.95	1422.31	1583.73	1171.65	15035.95
maxima	1091.88	1286.43	886.86	1213.65	1415.23	1095.13	1375.54	1489.21	1166.41	1497.19	1628.08	1222.12	15227.29
available	1121.81	1307.16	954.24	1408.32	1602.98	1199.33	1632.06	1881.29	1404.24	1917.71	2189.75	1627.33	18246.22

Heuristic	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	Total	Grand Total
RN2	817.26	811.46	731.88	838.73	1085.82	888.85	976.66	1325.87	954.81	1282.11	1349.28	1087.70	12150.44	26259.74
RN2-TS	835.35	811.46	764.44	898.69	1105.57	897.25	976.66	1424.67	967.76	1282.11	1567.51	1124.28	12655.74	27117.54
RN2-SA	817.26	811.46	731.88	838.73	1085.82	897.25	976.66	1325.87	967.76	1282.11	1379.80	1103.23	12217.83	26383.09
RN2-DR-TS	928.88	811.46	760.35	988.14	1255.20	985.27	1150.58	1425.43	1101.63	1394.78	1369.21	1181.48	13352.40	28315.15
RN2-DR-SA	861.41	811.46	731.88	935.78	1237.38	948.83	1150.58	1406.92	1007.06	1282.11	1369.21	1118.79	12861.40	27506.17
PR1	904.27	863.70	761.96	1086.68	1193.53	952.09	1194.28	1295.80	1072.72	1344.13	1505.78	989.12	13164.06	27447.95
PR1-TS	926.75	918.91	770.81	1134.88	1195.07	1016.15	1194.33	1378.85	1098.97	1370.97	1534.51	1142.49	13682.70	28593.12
PR1-SA	904.27	863.70	761.96	1086.68	1195.07	955.63	1194.33	1295.80	1072.72	1368.74	1530.33	1058.07	13287.30	27799.53
PR3	807.14	850.33	667.01	851.95	916.19	734.84	880.56	1115.68	736.25	1183.99	1401.72	719.25	10864.90	24259.47
PR3-TS	836.70	918.91	691.22	1005.36	1051.90	882.51	1167.65	1262.20	841.55	1326.39	1539.91	1118.87	12643.18	27125.90
PR3-SA	822.20	863.70	680.96	931.12	961.35	819.87	1062.83	1215.63	841.55	1314.07	1478.41	993.60	11985.30	26243.90
RP	892.80	878.70	737.20	1087.87	1211.61	1001.74	1276.58	1405.25	1051.52	1329.08	1569.14	1015.77	*13457.26*	*28320.90*
RP-TS	928.88	904.51	759.28	1134.88	1221.45	1014.61	1300.76	1440.28	1085.13	1391.48	1602.42	1179.06	13962.73	29190.01
RP-SA	892.80	878.70	746.39	1087.87	1211.61	1001.74	1279.91	1436.85	1074.21	1342.89	1599.93	1170.24	13723.13	28759.08
maxima	928.88	918.91	770.81	1134.88	1255.20	1016.15	1300.76	1440.28	1101.63	1394.78	1602.42	1181.48	13962.73	29190.01
available	1152.08	1319.15	975.13	1416.69	1628.88	1182.72	1615.60	1886.02	1381.33	1910.98	2204.97	1587.33	18260.88	35369.68

3.8.3.4 Random graphs with C = 144

- RP was the best constructive heuristic overall and for both classes.
- The best heuristic overall, and for both classes was RP-TS.
- All three heuristic types (RN2, PR, RP) achieved the best solution on some problem instances.
- On many of the early G problem instances (those with 90 edges and 110 edges), all the reward available on the graph was collected.

Table 3.19. Results from experiments on random graphs with C = 144

Heuristic	G1	G2	G3	G4	G5	G6	G7	G8	G9	G10	G11	G12	Total
RN2	1079.19	1184.97	893.08	1220.80	1394.27	1128.90	1437.98	1624.33	1103.73	1466.93	1833.02	1400.87	15768.07
RN2-TS	1079.19	1190.07	930.54	1358.83	1401.49	1187.70	1454.28	1652.87	1103.85	1466.93	1837.12	1436.60	16099.47
RN2-SA	1079.19	1190.07	900.37	1231.77	1401.49	1148.79	1454.28	1632.48	1103.85	1466.93	1837.12	1409.42	15855.75
RN2-DR-TS	**1121.81**	**1307.16**	**954.24**	1371.45	1555.60	1175.94	1523.81	1652.87	1301.12	1629.46	1897.03	1433.51	16924.00
RN2-DR-SA	**1121.81**	**1307.16**	**954.24**	1360.55	1548.38	1169.37	1456.41	1632.48	1166.10	1619.92	1888.52	1433.51	16658.45
PR1	1116.62	1300.22	949.05	1402.48	1597.10	1193.66	1558.34	1714.71	1326.25	1713.55	1852.07	1396.17	17120.22
PR1-TS	**1121.81**	**1307.16**	**954.24**	**1408.32**	**1602.98**	**1199.33**	1571.18	1723.57	1347.05	1744.78	1902.58	1437.23	17320.24
PR1-SA	**1121.81**	**1307.16**	**954.24**	**1408.32**	**1602.98**	**1199.33**	1558.34	1723.57	1326.25	1713.55	1891.20	1406.93	17213.67
PR3	1116.62	1300.22	949.05	1402.48	1597.10	1193.66	1532.49	1678.02	1297.63	1715.98	1863.56	1405.13	17051.95
PR3-TS	**1121.81**	**1307.16**	**954.24**	**1408.32**	**1602.98**	**1199.33**	1532.49	1731.31	1315.23	1735.85	1947.66	1414.53	17270.92
PR3-SA	**1121.81**	**1307.16**	**954.24**	**1408.32**	**1602.98**	**1199.33**	1532.49	1696.68	1311.74	1730.80	1904.71	1410.23	17180.51
RP	**1121.81**	**1307.16**	**954.24**	**1408.32**	**1602.98**	**1199.33**	1574.39	1775.52	1357.44	1736.39	1919.49	1455.24	*17412.32*
RP-TS	**1121.81**	**1307.16**	**954.24**	**1408.32**	**1602.98**	**1199.33**	1574.51	1793.12	1357.44	1755.16	1927.65	1458.46	17460.19
RP-SA	**1121.81**	**1307.16**	**954.24**	**1408.32**	**1602.98**	**1199.33**	1574.39	1782.45	1357.44	1755.16	1919.49	1458.46	17441.24
maxima	1121.81	1307.16	954.24	1408.32	1602.98	1199.33	1574.51	1793.12	1357.44	1755.16	1947.66	1458.46	17460.19
available	1121.81	1307.16	954.24	1408.32	1602.98	1199.33	1632.06	1881.29	1404.24	1917.71	2189.75	1627.33	18246.22

Heuristic	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	Total	Grand Total
RN2	1004.83	910.54	845.73	1067.09	1215.93	1042.96	1236.92	1434.75	1131.58	1672.47	1731.22	1342.00	14636.01	30404.08
RN2-TS	1046.92	948.45	845.73	1067.09	1235.68	1042.96	1347.35	1534.02	1184.63	1691.61	1751.15	1359.05	15054.64	31154.11
RN2-SA	1028.83	910.54	845.73	1067.09	1215.93	1042.96	1248.87	1452.11	1139.30	1672.47	1731.22	1347.98	14703.02	30558.77
RN2-DR-TS	1102.59	1169.33	937.98	1154.84	1361.61	1114.80	1419.49	1649.24	1187.61	1707.37	1770.15	1346.27	15921.27	32845.27
RN2-DR-SA	1095.74	1091.46	912.23	1154.84	1361.61	1114.80	1419.49	1649.24	1187.61	1672.48	1770.15	1346.27	15775.92	32434.37
PR1	1114.94	1161.52	937.98	1350.75	1508.73	1149.75	1460.78	1680.71	1274.38	1611.99	1863.55	1310.59	16425.68	33545.89
PR1-TS	1114.94	1166.03	937.98	1352.07	1527.22	1151.30	1496.00	1763.37	1304.18	1654.59	1898.77	1355.20	16721.65	34041.89
PR1-SA	1114.94	1161.52	937.98	1350.75	1508.73	1149.75	1465.88	1688.33	1276.01	1654.59	1863.55	1328.92	16500.95	33714.62
PR3	1114.94	1137.86	937.98	1350.75	1268.97	1149.75	1499.09	1462.40	1288.61	1651.65	1766.27	1293.08	15921.36	32973.30
PR3-TS	1114.94	1165.82	937.98	1352.07	1349.94	1151.30	1509.23	1544.61	1300.03	1686.85	1838.69	1341.48	16292.93	33563.86
PR3-SA	1114.94	1145.44	937.98	1350.75	1334.69	1149.75	1509.23	1544.61	1300.03	1669.87	1827.93	1320.88	16206.10	33386.61
RP	1113.82	1152.29	936.89	1346.13	1527.51	1151.30	1521.20	1738.68	1298.51	1672.29	1911.43	1387.69	*16757.73*	*34170.05*
RP-TS	1113.82	1163.08	937.31	1346.13	1533.92	1151.30	1523.30	1763.80	1302.74	1681.80	1926.00	1417.19	16860.37	34320.57
RP-SA	1113.82	1160.29	936.89	1346.13	1527.51	1151.30	1521.20	1747.29	1300.54	1681.80	1919.15	1417.19	16823.10	34264.34
maxima	1114.94	1169.33	937.98	1352.07	1533.92	1151.30	1523.30	1763.80	1304.18	1707.37	1926.00	1417.19	16860.37	34320.57
available	1152.08	1319.15	975.13	1416.69	1628.88	1182.72	1615.60	1886.02	1381.33	1910.98	2204.97	1587.33	18260.88	35369.68

3.8.3.5 Random graphs with C = 180

- RP was the best constructive heuristic overall and for both classes.
- The best heuristics overall, and for both classes were all three RP heuristics.
- RP found the best solution for all problem instances, and could not be improved by TS or SA.
- For many of the problem instances all the reward available on the graph was collected.
- The DR improvement procedure appears to make a considerable difference to the improvement of the RN2 heuristics.

Table 3.20. Results from experiments on random graphs with C = 180

Heuristic	G1	G2	G3	G4	G5	G6	G7	G8	G9	G10	G11	G12	Total
RN2	1101.42	1233.84	907.30	1375.73	1520.91	1149.84	1591.79	1735.89	1121.10	1702.24	2002.95	1504.67	16947.69
RN2-TS	1101.42	1251.28	923.42	1383.55	1530.52	1170.10	1608.09	1735.89	1121.10	1736.69	2007.06	1510.96	17080.08
RN2-SA	1101.42	1233.84	907.30	1375.73	1520.91	1156.19	1608.09	1735.89	1121.10	1712.99	2007.06	1510.96	16991.48
RN2-DR-TS	**1121.81**	**1307.16**	**954.24**	**1408.32**	**1602.98**	**1199.33**	1621.55	1801.76	1347.75	1859.73	2076.09	1577.25	17877.97
RN2-DR-SA	**1121.81**	**1307.16**	**954.24**	**1408.32**	**1602.98**	**1199.33**	1621.55	1774.19	1347.75	1840.91	2076.09	1562.28	17816.61
PR1	1116.62	1300.22	949.05	1402.48	1597.10	1193.66	1626.71	1872.36	1398.58	1912.14	2172.12	1621.06	18162.10
PR1-TS	**1121.81**	**1307.16**	**954.24**	**1408.32**	**1602.98**	**1199.33**	**1632.06**	**1881.29**	**1404.24**	**1917.71**	2172.12	**1627.33**	18228.60
PR1-SA	**1121.81**	**1307.16**	**954.24**	**1408.32**	**1602.98**	**1199.33**	**1632.06**	**1881.29**	**1404.24**	**1917.71**	2172.12	**1627.33**	18228.60
PR3	1116.62	1300.22	949.05	1402.48	1597.10	1193.66	1626.71	1872.36	1398.58	1912.14	2172.12	1621.06	18162.10
PR3-TS	**1121.81**	**1307.16**	**954.24**	**1408.32**	**1602.98**	**1199.33**	**1632.06**	**1881.29**	**1404.24**	**1917.71**	2172.12	**1627.33**	18228.60
PR3-SA	**1121.81**	**1307.16**	**954.24**	**1408.32**	**1602.98**	**1199.33**	**1632.06**	**1881.29**	**1404.24**	**1917.71**	2172.12	**1627.33**	18228.60
RP	**1121.81**	**1307.16**	**954.24**	**1408.32**	**1602.98**	**1199.33**	**1632.06**	**1881.29**	**1404.24**	**1917.71**	2177.66	**1627.33**	*18234.14*
RP-TS	**1121.81**	**1307.16**	**954.24**	**1408.32**	**1602.98**	**1199.33**	**1632.06**	**1881.29**	**1404.24**	**1917.71**	2177.66	**1627.33**	18234.14
RP-SA	**1121.81**	**1307.16**	**954.24**	**1408.32**	**1602.98**	**1199.33**	**1632.06**	**1881.29**	**1404.24**	**1917.71**	2177.66	**1627.33**	18234.14
maxima	1121.81	1307.16	954.24	1408.32	1602.98	1199.33	1632.06	1881.29	1404.24	1917.71	2177.66	1627.33	18234.14
available	1121.81	1307.16	954.24	1408.32	1602.98	1199.33	1632.06	1881.29	1404.24	1917.71	2189.75	1627.33	18246.22

Heuristic	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	Total	Grand Total
RN2	1075.01	962.14	866.43	1322.68	1471.82	1121.70	1390.19	1672.71	1267.13	1826.43	2028.37	1412.72	16417.33	33365.03
RN2-TS	1075.01	981.49	894.42	1356.03	1530.16	1150.58	1404.48	1672.96	1306.41	1826.70	2055.91	1447.85	16702.00	33782.08
RN2-SA	1075.01	962.14	866.43	1322.68	1471.82	1121.70	1404.48	1672.96	1267.13	1826.70	2038.70	1412.72	16442.48	33433.96
RN2-DR-TS	**1152.08**	**1319.15**	**975.13**	**1416.69**	1623.80	**1182.72**	1610.34	1880.08	1311.64	1826.70	2123.17	1504.75	17926.25	35804.22
RN2-DR-SA	**1152.08**	**1319.15**	**975.13**	**1416.69**	1623.80	**1182.72**	1610.34	1880.08	1311.64	1826.70	2123.17	1504.75	17926.25	35742.86
PR1	1146.89	1313.31	969.93	1411.66	1623.81	1176.73	1609.69	1879.13	1375.94	1894.77	2152.04	1566.24	18120.14	36282.24
PR1-TS	**1152.08**	**1319.15**	**975.13**	**1416.69**	**1628.88**	**1182.72**	**1615.60**	**1886.02**	**1381.33**	1894.77	2171.51	1568.34	18192.21	36420.82
PR1-SA	**1152.08**	**1319.15**	**975.13**	**1416.69**	**1628.88**	**1182.72**	**1615.60**	**1886.02**	**1381.33**	1894.77	2152.04	1566.24	18170.65	36399.25
PR3	1146.89	1313.31	969.93	1411.66	1623.81	1176.73	1609.69	1879.13	1375.94	1894.77	2165.31	1566.24	18133.41	36295.51
PR3-TS	**1152.08**	**1319.15**	**975.13**	**1416.69**	**1628.88**	**1182.72**	**1615.60**	**1886.02**	**1381.33**	1894.77	2165.31	1568.34	18186.01	36414.61
PR3-SA	**1152.08**	**1319.15**	**975.13**	**1416.69**	**1628.88**	**1182.72**	**1615.60**	**1886.02**	**1381.33**	1894.77	2165.31	1566.24	18183.91	36412.52
RP	**1152.08**	**1319.15**	**975.13**	**1416.69**	**1628.88**	**1182.72**	**1615.60**	**1886.02**	**1381.33**	1899.13	2179.78	1569.17	*18205.67*	*36439.82*
RP-TS	**1152.08**	**1319.15**	**975.13**	**1416.69**	**1628.88**	**1182.72**	**1615.60**	**1886.02**	**1381.33**	1899.13	2179.78	1569.17	18205.67	36439.82
RP-SA	**1152.08**	**1319.15**	**975.13**	**1416.69**	**1628.88**	**1182.72**	**1615.60**	**1886.02**	**1381.33**	1899.13	2179.78	1569.17	18205.67	36439.82
maxima	1152.08	1319.15	975.13	1416.69	1628.88	1182.72	1615.60	1886.02	1381.33	1899.13	2179.78	1569.17	18205.67	36439.82
available	1152.08	1319.15	975.13	1416.69	1628.88	1182.72	1615.60	1886.02	1381.33	1910.98	2204.97	1587.33	18260.88	35369.68

3.8.4 Discussion of phase 2 results

Points have already been made in the relevant sections about the individual results tables, so we concentrate here on observations about the similarities and differences between the tables.

3.8.4.1 Designed graphs

Constructive heuristics. For C = 36 and C = 72, RN2 was the best constructive heuristic for each class and overall. For C = 108, RP was the best overall and for classes L and R, but RN2 was the best for class C. For C = 144 and C = 180, RP was the best overall and for each class.

This suggests that RP is better than RN2 when the graph is both sparse and the budget is high. When only a small section of the graph can be traversed then RN2 seeks out better subtours.

Best heuristic overall. In all cases, the TS improvement was at least as good, and usually better than both the unimproved heuristics, and the SA improvement. RP dominated for higher budgets, and RN2 dominated for lower budgets. Tabu Search and Steepest Ascent were both quite sensitive to the initial solution.

Failure of RP. On several problems when C = 36, RP produced extremely bad solutions, even zero on one problem. This is probably due to deleting large amounts of the route with the final cycle deletions. It is a symptom of using RP with low cost budgets, and could have been anticipated by considering the nature of the heuristic. Fortunately, the TS and SA improvements were able to recover reasonable solutions.

Summary. It seems that the best approach is to use RN2 with TS (either with DR or without) when the cost budget is low (< 90), and to use RP with TS when the cost budget is high.

3.8.4.2 Random graphs

Constructive heuristics. Similarly to with the designed graphs, RN2 is the best constructive heuristic for C = 32, while RP is the best for the other budget levels. RP is better for lower budget levels than with the designed graphs, possibly because the random graphs are sparser.

Best heuristic overall. Again, the TS improvement dominated both unimproved solutions, and the SA improvement. RP-TS was the best heuristic for all budget levels, although often other heuristics achieved the same values.

Summary. The same as for the designed graphs.

Coda

▼ Summary

In this chapter we developed several original heuristics for the ASRP. PRUNE THEN ROUTE iteratively deletes arcs until the solution to the resulting CPP relaxation is a cost feasible solution to the original problem. ROUTE THEN PRUNE solves the CPP relaxation, and then deletes cycles from the tour until it is cost-feasible. RICHEST NEIGHBOUR uses a look-ahead greedy criterion to construct a path, with a check to ensure the path returns to the depot within the cost budget. We also developed several improvement procedures that were implemented as moves under the framework of Tabu Search.

Experimentation was performed on grid graph instances. Two methods of generating random grids were used, GRID GROW and GRID SELECT, and these methods were found to produce graphs with different characteristics. In particular, grids generated using the GRID GROW method have a larger number of isolated vertices, and those generated using the GRID SELECT method have a larger number of end edges.

We explored the influence of depot location by using three designed instances, and discovered that the position on the grid (corner, middle, or edge) did not have a significant effect, other than that attributable to regions of reward density. Future experiments might profitably devise some measure of reward distribution, perhaps based on clusters.

Extensive computational tests on random instances, allowed us to determine that for low-medium cost budgets, the RICHEST NEIGHBOUR heuristic performed best, while ROUTE THEN PRUNE was best for higher budgets.

This is the end of Part I, which was a focused study of the ASRP as a traditional Operations Research problem. The ASRP will be the test problem for later experimentation.

▼ Link

In Part II we introduce the Modular Local Search framework. We define its architecture and describe its programming implementation, and review common metaheuristics from the literature, exploring how they can be expressed in the MLS framework.

Part II
MLS Foundations

Overview of Part II
MLS Foundations

Part II introduces and defines the Modular Local Search framework.

Chapter 4 explains the architecture of MLS, defining all the components and discussing potential applications of these. This is followed by some illustrative examples of how popular metaheuristics would be modelled as MLS, without formally defining the logic for the modules, which occurs later. This chapter finishes with a discussion of the strengths and limitations of MLS.

Chapter 5 is a literature review for trajectory-based metaheuristics. The main metaheuristic paradigms are discussed extensively, since these provide the building blocks of MLS. Thoughts are given as to how these metaheuristics could be modelled as MLS techniques.

CHAPTER 4

Modular Local Search

This chapter provides a detailed description of the structure of MLS, as well as discussion of various ways in which the components and modules can be modelled and combined. To illustrate the concepts, a number of metaheuristics are described in terms of their MLS modules. A brief discussion of the strengths and limitations of MLS concludes.

4.1 Introduction

We introduce Modular Local Search (MLS), a framework for metaheuristics. MLS identifies a number of operators and stages that many metaheuristics have in common, and formalizes this structure as a modular heuristic system, whereby a particular metaheuristic can be implemented simply by "slotting in" the appropriate modules. An advantage of this approach is the ease with which modules that were originally inspired by particular metaheuristics can be mixed and matched to create hybrids. The nature of the MLS framework is that it makes explicit some previously implicit aspects of existing metaheuristics, thereby allowing these aspects to be manipulated, creating opportunities for new metaheuristic ideas and novel hybrids.

One of the most fundamental ways to classify metaheuristics is whether they are trajectory-based or population-based. **Trajectory-based** methods maintain a single solution and each iteration updates this single solution, usually by moving to a neighbour of the solution, but sometimes by the mechanism of a translocation to a different point in the solution space, for example by a reconstruction, a perturbation,

or revisiting past solutions. **Population-based** methods maintain a population of solutions, and perturb and combine these in various ways over a number of generations. Modular Local Search is designed to model *trajectory-based* methods; an MLS-based framework to support population-based methods is possible, but would require a number of different components and is beyond the scope of this thesis.

An important consideration is that there are many possible metaheuristic frameworks. Some previous efforts were briefly discussed in Chapter 1. There is no necessarily "right" framework; each may be useful for specific purposes. In the words of the statistician George Box [31], "all models are wrong, but some are useful." The motivation behind MLS is to create a heuristic system, such that "modules" derived from various metaheuristic concepts can be mixed and matched in a standardized way. The eventual goal of such a system is to allow heuristic guiding strategies to modify the algorithm as it progresses, based on the state of the search and the past successes or failures of attempted changes: essentially to enable learning and self-adaptation.

The MLS framework consists of both a conceptual structure, which is the focus of this chapter, and a programmatic implementation of that conceptual structure, which is outlined in Appendix B. The conceptual structure was designed from the start with the intention of being implemented as an object-oriented system, so there are some references to Classes, and we refer to modules "calling" other modules. These references are clarified in Appendix B. Appendix C describes the Modular Local Search Markup Language (MLSML). This is an XML-based language that specifies all the modules in an MLS heuristic. Assuming all the required modules have already been programmed, MLSML allows new complex hybrids to be expressed *declaratively*, rather than *programmatically*. The user simply specifies the modules that should be assigned to each component, and the program handles the interaction of these different concepts.

This chapter is structured as follows. A high-level description of the structure of MLS is given in Section 4.2. Sections 4.3-4.5 describe each of the MLS components in detail and Section 4.6 summarizes these into a list of all MLS components. Section 4.7 gives some illustrative examples of metaheuristics expressed as MLS. Finally, Section 4.8 discusses some of the uses of MLS, along with its strengths and weaknesses.

4.2 Structure of MLS

Three core concepts underpin the structure of MLS: the search scheme, the control system, and the memory structures. We define an **MLS component** to be a specific role in the MLS process, and an **MLS module** to be a specific function or value to provide functionality for that component. For example, one component is *generate-initial-solution*. This generic component is where the first solution is generated by some process, and then this solution is used as the starting point for the search. Particular modules that could be used in the *generate-initial-solution* component include *random generation* and *greedy construction*. One way to think of the distinction between components and modules is that components are the "slots" that need to be filled to define a heuristic, and modules are the specific parts that fill those slots. Another analogy is that the components define the *architecture* of the MLS framework; a blueprint for the bathroom of a house might call for a bath, that would be the

required “component”; specific “modules” that could be used include “claw-footed bath”, “free-standing modern bath” and “sealed and wall mounted bath”.

The **search scheme** is the set of components that control how the heuristic moves from one solution to the next. They influence the search through the **search iteration process**, a procedure that takes a starting solution as input and outputs a target solution, in a single iteration. The search scheme components together define the search topology, and how the topology is searched to find the target solution that the search moves to.

The **control system** consists of everything else required to execute a metaheuristic. It controls which solution should be passed as input to the search scheme, and makes changes to the search scheme, the control system itself, and the memory structures, as required throughout the search process. It is the “intelligence” of the heuristic algorithm.

The **memory structures** store any parameters, quantities or sets of attributes that are used by the search scheme and the control system.

We refer to MLS configurations. A **configuration** is a particular set of modules that are specified as belonging to the same heuristic. Modules are not necessarily all active at the same time; all the modules used in a heuristic are specified at the start, but they may be activated or deactivated in response to certain criteria.

4.3 The search scheme

The **search scheme** consists of a number of components that are present in all local search heuristics, either explicitly or implicitly. These components interact with the search through a number of steps of the **search iteration process**. Starting from the current solution, the search iteration process finds a neighbouring solution to be the target solution or, if no suitable solution can be found, returns the current solution as the target solution.

The search iteration process is described below:

Algorithm 4.1 procedure MLS SEARCH ITERATION PROCESS
    Input the current solution s
    Generate the move-list $\rightarrow M(s)$
    Reduce the move-list $\rightarrow M^R(s)$ *// Optional step*
    repeat
        Choose a trial solution, s''
        Check the trial solution admissibility
        if s'' is admissible **then** add it to the candidate list, A
    until sufficient trial solutions have been examined
    if A is empty **then** let the target solution be the current solution, $s' = s$
    else let the target solution be the candidate with the highest fitness
end

Figure 4.1 illustrates the search iteration process. There are three main points of control over the breadth of the search, which we define as the number of solutions examined. The current solution is used to generate a set of specific moves that can be performed to generate neighbour solutions. This set may not contain all the moves that are possible on the current solution, so offers the first point of control over the breadth of the search for this iteration. Next, an optional neighbourhood reduction process is performed to reduce the number of number of moves in the move-list; this process may involve some sort of fast, or "heuristic" evaluation of the move quality, and is the second point of control over the breadth of the search. Then the search process selects moves one at a time, constructing the resulting neighbour solution and evaluating the admissibility of this solution. If the neighbour is admissible it is added to the candidate list and this continues until either candidate list is full or the number of solutions examined has reached the limit. These thresholds are the third point of control over the breadth of the search. The final step is to choose the candidate with the highest *fitness* to be the target solution, i.e. the solution outputted by the search iteration process. If there were no admissible candidates found then the search notes this and returns the current solution. **Fitness** is usually the objective function value, but can be some other proxy value, as described subsequently.

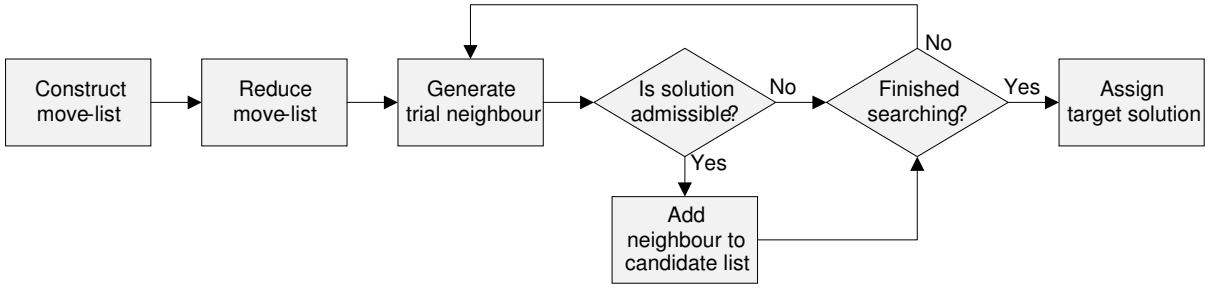


Figure 4.1: The search iteration process

The search iteration process is controlled by the components of the MLS search scheme:

- The neighbourhood scheme
- The neighbourhood reduction process
- The fitness function
- The admissibility conditions
- The search logic

Conceptually, the neighbourhood scheme and the admissibility conditions together define the set of solutions that could possibly be chosen, and the search logic dictates which of these solutions will be constructed and evaluated.

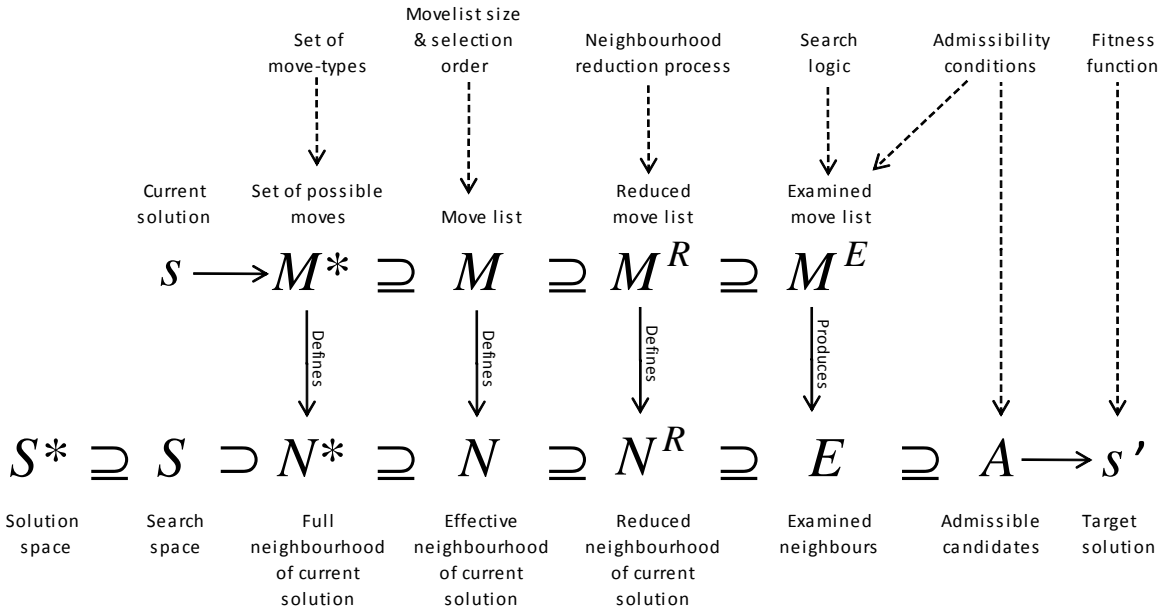


Figure 4.2: Relationships of solutions and moves in the MLS search iteration process

Figure 4.2 illustrates the relationships between the different sets of solutions and moves. The top row indicates which MLS search scheme components influence each set. Some of the above sets are conceptual only, and some are actually generated by the MLS algorithm. Only M, M^R, M^E, E and A are real sets calculated by the MLS algorithm during a search iteration, each generated from the previous:

$$s \to M(s) \to M^R(s) \to M^E(s) \to E(s) \to A(s) \to s'$$

The **solution space** S^* is the set of all possible solutions to the problem instance, while the **search space** S is the set of all solutions that can conceivably be reached by the MLS heuristic. These may be equivalent, but in some instances there may be solutions that cannot be reached by the MLS heuristic. The set of move-types defines the theoretical set of all possible moves on the current solution, where a move is an actual instance of a type of move (so swapping two cities in the TSP is a move-type, swapping cities B and F is a move). The move-list is the subset of moves that are actually stored in memory. This conceptually defines the effective neighbourhood of solutions; those which we could choose to generate, depending on what happens further in the search scheme process. The neighbourhood reduction process selects a subset of the move-list, from which we select those to perform to check for admissibility of the resulting solution. The reduced move-list defines the reduced neighbourhood of the current solution. The search logic and the admissibility conditions together determine which moves will actually be performed to create solutions that are tested for admissibility; these solutions form the set E, which is actually stored in memory. Of these, some are admissible, and are added to the candidate list. The final step is to select the best candidate, based on the evaluation of the fitness function.

The following sections describe each of the MLS search scheme components in more detail.

4.3.1 The neighbourhood scheme

The **neighbourhood scheme** defines which solutions are in the neighbourhood $N(s)$ of current solution s. A **neighbour** is any solution that may be reached by the application of one move on the current solution, so the neighbourhood is determined by the set of moves and move-types available. The neighbourhood scheme has three parts: the **set of move-types**, the **move-list size**, and the **move selection order**.

An important point to note is that although the neighbourhood scheme *defines* which solutions are in $N(s)$, these solutions are not actually constructed by the neighbourhood scheme. The neighbourhood scheme is focused on moves; a neighbour is a solution obtained by the application of a move to the current solution. The purpose of this is to save computational effort. For some problems the actual execution of a move is a significant computational effort – there is no need to do this for every solution in the neighbourhood if we are simply going to accept the first improving solution examined. We do, however, catalogue the *moves* that can be performed, without actually performing them. For example, when constructing the neighbourhood for a TSP solution, we might note that a move is to swap customers A and B, but we do not actually construct or evaluate the resulting route. Since the solutions are not constructed or evaluated, at this stage they can still be either feasible or infeasible. This set of moves is called the **move-list**.

The ability to restrict the neighbourhood offers more flexibility and an opportunity to make the heuristic more efficient. However, it can involve some difficult design considerations, depending on the problem and move structure. Choosing an unlimited maximum neighbourhood size avoids these difficulties and defaults to the standard option of considering the whole neighbourhood.

4.3.1.1 Set of move-types

We define a **move-type** to be the abstract definition of how to perform a move, and a **move** to be the specific realization of a move-type on the current solution. For the TSP, swapping the position of two cities in the route is a move-type; swapping the position of cities D and F is a move. For a given current solution, there will usually be many possible moves of each move-type.

Each move-type is either *active* or *inactive* at any given iteration. Inactive move-types are defined within the heuristic but are not available to the search scheme and are not evaluated. Move-types may be activated or deactivated by the control system between iterations.

4.3.1.2 Move-list size

The **move-list size** is the maximum number of moves that can be added to the move-list. It can either be *unlimited*, in which case all possible moves on the current solution are considered, or some specified number, in which case only a subset of moves are considered. The move-list size allows us to limit the computational effort required in listing the moves that can be performed. For some problems this can be a significant computational burden, and is often unnecessary, for example if the first admissible solution is to be chosen. Note that the cardinality of the move-list may be less than the move-list size, if there are fewer moves possible.

4.3.1.3 Move selection order

If the move-list size is unlimited, then the move selection order is unimportant, since all potential moves will be added to the move-list. However, if the move-list size is constrained then there is a choice to make for which possible moves to include. There are several options here:

Purely random. It is actually quite hard to select a purely random move, without listing all the possible moves first, which defeats the purpose of having a restricted move-list. The only way to do this is if there is some special structure to the moves so that we can randomly select the elements. For example, in the TSP if we have a move-type that is to swap the position of two cities within the route, then we can randomly generate a move by randomly choosing the first city (as a random number between 1 and the number of cities) and then randomly choose another city. This method also requires the addition of a step to check whether the generated move is already on the move-list. If there are multiple types of moves then there is a further complication. We need to know in advance how many of each type of move are available, and then the first step is to randomly choose the move-type, based on the proportion of the total moves available that each move-type represents, and then randomly choose the move from the moves of that move-type. A purely random move selection order requires some problem-specific consideration to implement, and may not be possible for all problems.

Random weighted by move-type. This method is essentially the same as the purely random method, except that instead of choosing the move-type of each next move by the proportion of total moves represented by that move-type, the weighting is specified by the user. These weightings control the tendency to prefer some moves over others, and can therefore be used to guide the search. To the best of our knowledge, this technique has not been considered in the literature. The weightings for each move-type are memory parameters that can be varied dynamically throughout the course of the heuristic.

Random fixed for move-type. Similar to the previous method, except that for each move-type a *fixed* proportion of the maximum neighbourhood size is selected, not necessarily the same for each move-type. Again, the proportions are memory parameters.

By implementation structure. There are many ways to store problem structures and solutions in the programming implementation of the heuristic, depending both on the problem and the programmer. Usually for these there will be a natural way to loop through the elements; for example if a route for the TSP with P cities is stored in a list, with the first element of the list being the first city visited, and so on, then a natural way to enumerate all the moves would be the following:

```
for i=1 to P do
    for j=i+1 to P do
        move = swap(route[i], route[j])
        add move to move-list
    end
end
```

Notice that this method produces a biased neighbourhood when the neighbourhood size is restricted; moves with cities near the start of the route will tend to be included in the neighbourhood more often.

When the move selection order is based on the implementation structure the order of the move-types to be examined is very important, since the all the moves of the first move-type will be selected before any of the second or subsequent types are selected, unless a fixed proportion for each move-type is specified, as described below.

By implementation structure, fixed for move-type. A fixed proportion of the maximum neighbourhood size is specified for each move-type and stored as memory parameters. The heuristic will select the appropriate number, as described above, for each move-type.

4.3.2 The neighbourhood reduction process

This is an entirely optional step in the search scheme, and most heuristics do not have this process. Examples of heuristics that do are given in Chapter 5. The **neighbourhood reduction process** applies some subroutine or rule to choose a subset of the moves in the move-list to be progress to be evaluated as solutions and checked for admissibility. This process will usually not involve explicitly evaluating the fitness function of each solution – the utility of this phase is in using more streamlined evaluation methods. There are not restrictions on the type of process, but there are two main types: *cardinality-based* and *evaluation-based*.

Cardinality-based reduction processes ensure that the reduced move-list (which defines the reduced neighbourhood) contains only a specified number of moves, which is a memory parameter. These can be selected randomly, or according to some selection routine, or based on a ranking according to some evaluation, like the ones discussed below.

Evaluation-based reduction processes include all those moves that meet some defined evaluation criterion – possibly comparison against a threshold, which is stored as a memory parameter.

One of the primary uses of a neighbourhood reduction process is for when the solution construction, or fitness function evaluation is quite a complicated or time-consuming task. If evaluating a solution is expensive, then each iteration of the search scheme can become quite expensive, if many solutions must be examined, especially if there is a high-ratio of poor neighbours to good neighbours. In this case a simpler proxy for the fitness function can be used to filter out moves that look promising, and those that don't.

The neighbourhood reduction process can also be used as part of the "intelligence" of the search process. One way of doing this would be to consider the solution elements that the move modifies, and check whether these solution elements (e.g. arcs in an arc routing problem) are on a list of previously identified "good" elements, or "bad" elements.

Another "intelligent" use is to increase or decrease the level of diversification occurring. If a fast indicator of the level of diversification a move introduces exists, then this can be used to specify that only high-diversification moves will be considered, without the need to explicitly construct and evaluate the solutions resulting from these moves.

One use of the neighbourhood reduction process can be to check for infeasibility, if this can be easily ascertained. The neighbourhood construction step simply considers possible moves, and is blind to whether or not they are feasible. If this is an important concern it can be considered as a neighbourhood reduction step. The alternative is to consider feasibility as part of the admissibility conditions, however if the feasibility of a solution can be checked or estimated without explicitly constructing it, then there may be some value to performing a check here.

If the neighbourhood reduction routine involves explicitly constructing the solution resulting from a move, and then evaluating the full fitness function, then there may be no advantage from using a neighbourhood reduction process over using the search logic and admissibility conditions.

Explicit neighbourhood reduction processes have not been considered often in the literature, however we consider them to have significant potential to efficiently guide searches to productive areas of the search space, especially in problems where solution construction or fitness function evaluation is computationally expensive.

4.3.3 The fitness function

The fitness function is used to determine the relative value of solutions. In its most basic version, the fitness function is simply the objective function, and most heuristics use this expression. However allowing the fitness function to potentially be different to the objective function gives the ability to modify it to create slightly different search schemes, with different search topologies. As with all MLS components, this modification can be combined with other modules to create hybrids.

An example of a heuristic which uses fitness function modification is Guided Local Search, which penalizes certain solution attributes (such as the inclusion of certain arcs in the TSP), making them less desirable and hence creating a modified search topology. Another use of the fitness function is to penalize infeasibility rather than making infeasible solutions inadmissible.

If the objective function is computationally expensive to evaluate, then a simplified evaluation function can be used for the fitness function in order to speed up the search. The fitness function can change at certain points throughout the search, perhaps returning to be the objective function in the last stages of an intensification phase.

More complicated fitness functions can involve the execution of some algorithm. For example a sophisticated MLS heuristic might use the evaluation of a simpler version of MLS (perhaps a basic ascent search) as the evaluation of the fitness of the neighbouring solution. In this case the value of the objective function for the local optimum reached by the evaluation of the simpler local search heuristic would be the fitness function.

4.3.4 The admissibility conditions

The **admissibility conditions** describe whether an examined solution is a **candidate** for selection as the target solution. If a solution is examined and found to be admissible then it is added to the **candidate list**.

The admissibility conditions examine solutions in relation to the current solution, to other solutions in memory such as the best-so-far, or according to certain rules and thresholds. They do not compare neighbours of the current solution with each other. So an admissibility condition cannot require that a solution be the best available - this type of condition is dealt with by the search logic.

The following are some examples of admissibility conditions in common local search heuristics:

- In *Ascent Search* only improving solutions are admissible;
- In *Tabu Search*, solutions that are not tabu, or that are tabu and satisfy the aspiration criteria, are admissible;
- In *Simulated Annealing*, solutions that are improving are admissible and solutions that are not improving are admissible with a certain acceptance probability.

A heuristic can have multiple admissibility conditions. If this is the case then *all* admissibility conditions must be satisfied. One common admissibility condition that is applied in addition to others is that the solution be *feasible*.

One possible use of admissibility conditions is to modify the data of the problem instance. For example, if certain solution attributes are known to be sub-optimal then these can be made non-admissible. This is similar to the neighbourhood-reduction process of filtering out moves that introduce these elements, although in this case the solution is actually generated, so using the admissibility conditions to achieve this effect could potentially be more expensive in terms of computational effort. Yet another approach would be to use the fitness function to penalize these solution attributes, rather than forbidding them completely.

4.3.5 The search logic

The search logic controls how moves are chosen from the reduced move-list, and examined by the admissibility conditions to determine if they are candidates. The search logic has three components:

- **The examinations maximum**. This is the number of solutions to evaluate to determine if they are admissible (can be unlimited).
- **The candidate list size**. This is the maximum number of candidates that can be added to the candidate list of admissible neighbours (can be unlimited).
- **The examination order**. This determines how moves are selected from the move-list to be examined by the admissibility conditions (has no effect on the search if the examinations maximum and the candidate list size are unlimited).

Note that both the examinations maximum and the candidate list size are simultaneously in force, so either can stop the search if it is reached. The search is also stopped if the move-list is exhausted without finding any admissible candidates.

After the search is finished for one of the three above reasons then the best solution in the candidate list, based on the fitness function evaluations, is returned. If the search does not find any candidates then the search scheme notes this and returns the current solution.

We discuss the implications and applications of the search logic and the restricted move-list in more detail in Section 4.8.1.

4.4 The control system

The **control system** of modular local search serves as its "intelligence". If the search scheme may be imagined as a particular point in "heuristic space", then the control system is the *meta*-heuristic that guides a path through heuristic space.

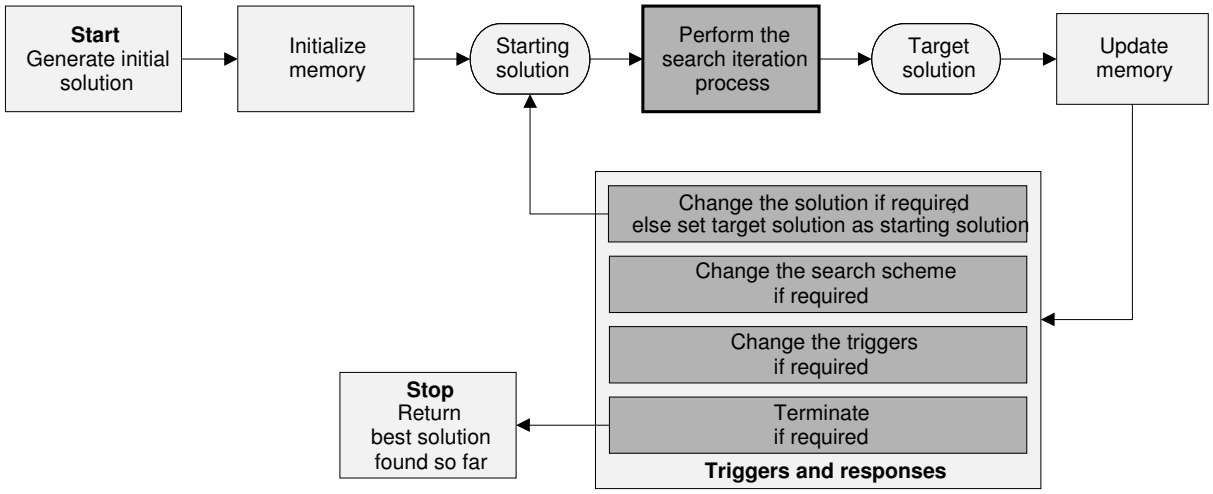


Figure 4.3: The MLS control system

The control system is composed of a number of components. Figure 4.3 illustrates the flow of information through these components. First an initial solution must be generated; at each iteration the search scheme acts on a starting solution and selects one of its neighbours, so a solution to begin this process is required. The next step is a memory initialization. The counters and other system-maintained memory structures are automatically initialized, so no module is necessary, however some heuristics may have specific initialization requirements. Next is a single search iteration, using the components of the search scheme, starting from the current solution and producing a target solution. The memory is then updated, based on the results of the search iteration.

The next stage checks to see if any of the triggers have been tripped, and if they have then the associated responses are performed. Multiple responses can be associated with each trigger, and responses generally perform one of four tasks: terminating the algorithm if a termination criterion is satisfied, changing the current solution, modifying the search scheme, or changing the triggers or responses themselves for the next iteration.

4.4.1 Generate-initial-solution

MLS is a local search methodology, it moves from solution to solution, so an initial solution is required to provide the starting point of this trajectory. The choice of initial solution can have a significant impact on the quality of the final solution for local search heuristics, and this is discussed in Chapter 5.

However, it is beyond the scope of our research to consider this aspect of heuristic design, and we treat the initial solution generation as a black-box function; any appropriate problem-specific method can be used as a module for this component.

The only exception to this is where the construction of a solution can be modelled as a local search heuristic. In this case the initial solution can be simply an empty solution, and the construction of the solution is treated as a phase of the MLS procedure. An example of this is the Greedy Randomized Adaptive Search Procedure (GRASP), where a solution construction phase is followed by a local search improvement phase. It is possible to model both of these phases as MLS, with a different configuration of modules for each phase.

4.4.2 Initialize-memory

The initialization of memory structures, such as iteration counters and lists of elite solutions, happens automatically, and does not require an MLS module. The **initialize-memory** component of the control system is for additional initialization tasks that may be required by some heuristics.

An example of a memory initialization is for the List-Based Threshold Accepting method of Tarantilis et al. [234]. In this heuristic a list of admissibility threshold values is generated at the start of the procedure, and then as the actual search progresses these are modified and replaced.

If an initialize-memory module is required for the execution of another MLS component, then this is specified by the requiring module, and a valid MLS heuristic will satisfy this requirements.

4.4.3 Update-memory

The **update-memory** component consists of multiple modules that update any of the memory structures that are in use in the heuristic. These update-memory modules are performed in the following order, based on the type of analysis.

4.4.3.1 Automatically updated memory structures

Some memory structures are updated automatically by the MLS framework without requiring a specific module. For example, all the iteration counters are automatically incremented, and all the search scheme quantities are stored in memory, such as the number of solutions examined, and the size of the move-list. These are discussed in more detail in the section on memory structures.

4.4.3.2 Specific lists or quantities

If a heuristic requires specific lists or quantities to be maintained then these can be managed through an appropriate update-memory module. The possibilities are endless, but some common examples are described below.

Tabu Search requires a list, or lists, of tabu solutions or solution elements to be maintained. This would be done in a special update-memory module that adds any newly tabu elements to the lists, updates the remaining tenure of all items, and removes any items whose tabu status has expired.

List-Based Threshold Accepting maintains a list of threshold values that are used to determine admissibility. The updating of this list would be performed by an update-memory module.

4.4.3.3 The best-so-far solution

One of the update-memory features that *requires* a module determines whether the best-so-far (BSF) solution is updated with the fitness function or the objective function. The overall goal of the heuristic is to find the solution with the best objective function value, so it would seem logical to use this evaluation method, however one of the advantages offered by using a simplified fitness function is that it can increase the speed of the search and this advantage is lost if every iteration the objective function must be evaluated against the objective function anyway.

The alternative, i.e., always using the fitness function, has its own complications. First is the obvious problem that BSF solutions, even globally optimal solutions, based on the objective function evaluation, may not have high fitness function values, and so these solutions may be passed over by the search if the objective function is not evaluated. A secondary problem is that the fitness function is liable to change from one iteration to the next, for example in guided local search, so that comparing the fitness function value of one solution with another to determine if the BSF should be updated can result in unfair comparisons.

The solution within the MLS framework is to make both methods possibilities and to leave the decision of which to utilize up to the heuristic designer, with possibly both being used at different phases of the heuristic. A MLS heuristic must have at least one of the following modules:

- **Update *BSF (objective function)***. This module evaluates the value of the objective function of the target solution against that of the current *BSF (objective function)* and if the target solution has a better objective function value then the target solution replaces the current *BSF (objective function)*. Note that the objective function value of the *BSF (objective function)* is stored in memory, rather than re-evaluated for each execution of this module.

- **Update *BSF (fitness function)***. This module evaluates the value of the fitness function of the target solution against that of the current *BSF (fitness function)* and if the target solution has a better fitness function value then the target solution replaces the current *BSF (fitness function)*. Note that the current *BSF (fitness function)* solution is re-evaluated using the current fitness function if this has changed since the last update, in order that both solutions are compared using the same measure, the current fitness function.

An MLS configuration can have either or both of these modules in operation at the same time. Although these modules are primarily *update-memory* modules, they can also be called as responses to certain triggers being tripped. This is discussed in more detail in Section 4.4.4.5.

If another MLS module requires a particular BSF solution to be present, then this is described in the specification for that module, and a valid MLS heuristic will satisfy this requirement. For example, a common Tabu Search aspiration criterion is that a trial solution always be admissible, regardless of its tabu status, if it is better than the best solution found so far. This comparison would usually be based

on the fitness function evaluation, rather than the objective function evaluation (although this is up to the designer), so the appropriate BSF solution would need to be available for comparison.

4.4.3.4 The state of the search

Update-memory modules are not restricted to simply storing values and entities that have already been calculated during the search iteration process, or incrementing counters; they can be quite complex and sophisticated procedures that analyze the current state of the search. Such modules cannot affect other modules directly themselves, but they can create "flags" in memory that may be detected by complimentary triggers, which are then able to perform any desired response. These responses will typically be used to either intensify or diversify the search.

The **state of the search** is an abstract concept that can nevertheless be quantified by an appropriate procedure, which is implemented in MLS through update-memory modules. The following paragraphs describe some aspects that could be investigated.

Results of current iteration. The outcomes of the current iteration, such as whether a target solution was found, how many moves and solutions were examined, etc.

Rate of improvement. The trajectory of solution values can be analysed, both in the immediate past and over the whole course of the heuristic. This can be used to identify when progress has stalled, or when it is progressing satisfactorily.

Quality of search space region. Tests can be performed to determine how promising the current region of the search space is. These could be based on the recent search history, or could involve other investigations into aspects of the search topology, such as the ratio of improving neighbours to non-improving neighbours, the prevalence of certain solution attributes in neighbours, or the "steepness" of the search space. An example of how this might be estimated is to randomly choose three neighbouring solutions, and then for each of these perform a basic ascent search with a simple move structure, and count the number of moves required to reach a local optimum.

Performance of heuristic modules. It is possible to track how well the search is doing, based on any number of metrics, and record this against the modules that are in use at the time. For example, at the end of a diversification phase the success of the diversification can be measured (perhaps based on the quality of the new search region), and recorded against the configuration used for the diversification. If there are multiple methods of diversification then these can gradually be trialled, and their success recorded. If the probability of selecting a particular diversification method is based on its past success then we have managed to implement a form of learning. More sophisticated approaches would also record other aspects of the search state at the same time, such as the prevalence of certain solution features in the neighbourhood. Sophisticated classification and prediction models could be employed to determine appropriate combinations of modules for certain patterns of within the search state, for example decision trees or neural networks. This type of update-memory module would probably not be executed every iteration, but could be available to call on demand as a response to certain triggers. Chapter 8 discusses advanced MLS applications such as learning in more detail.

4.4.4 Triggers and responses

Many of the components within the MLS framework are conceptually similar to procedures in existing heuristics from the literature, except that they have been made more explicit and formalized in MLS. The concept of triggers and responses is, to the best of our knowledge, quite original. They allow an MLS heuristic extremely flexible control over its own structure, and the state of the search. The trigger-response model is the key to the ability of MLS to be self-adaptive.

After every search iteration a test is performed to check whether a trigger is **tripped**, i.e. whether the **trigger logic** is satisfied. If the trigger is tripped then one or more **responses** that are associated with that trigger are performed. A response is procedure that performs some piece of analysis or changes another module or memory structure.

4.4.4.1 Parts of a trigger

A trigger has the following attributes:

- **Trigger logic**. This specifies the logic that is checked to determine if the trigger has been tripped. The trigger logic can only refer to and compare memory elements, both those specific to this trigger and any other generic memory elements. A trigger can consist of any desired calculation or subroutine, but must evaluate to either *true* or *false* (*tripped* or *not tripped*). However, the logic for a trigger should be as simple as possible, with most of the analysis tasks being performed during the update-memory phase.

- **Trigger memory elements**. These are parameters against which the trigger can be compared. Not all triggers need have memory elements, but they can and they are stored as part of the memory structures, so can be updated or modified by responses if desired. When a trigger is designed, one or more memory structures will usually be designed to service the trigger, along with any required update-memory modules to maintain the memory structures.

- **Active flag**. A trigger is either *active* or *inactive*. Triggers can be specified for a particular heuristic and then activated and deactivated throughout the course of the heuristic execution. Typically certain triggers will relate to certain MLS configurations, and when the configuration is changed one set of triggers will be deactivated and another set will be activated to take their place.

- **Responses**. A trigger can have one or more responses associated with it. Responses can be added or removed from a trigger throughout the course of the heuristic. Responses are performed as soon as the trigger is tripped, and they are performed in the order that they are specified.

4.4.4.2 Examples of triggers

Triggers are designed to react somehow to certain states of the search. The following list gives some examples of the possible conditions that could cause a trigger to be tripped:

- The total number of iterations for the heuristic has reached a threshold;

- A certain number of iterations have passed since some milestone event;
- There have been no improvements in the search for a certain (threshold) number of iterations;
- The allowed duration of the heuristic has expired;
- The search iteration process found no admissible candidates (an apparent local optimum);
- The ratio of admissible solutions examined to non-admissible solutions examined fell by a certain amount, e.g. by more than 20% over the past 5 iterations;
- The value of a memory element is less than a certain threshold (for example the temperature in Simulated Annealing);
- A certain number of iterations have passed since this trigger was last tripped *and* the search has not improved for a certain number of iterations.

A general design imperative is that triggers be **problem-agnostic**. What this means is that their logic should be based on comparison of values and thresholds, and cardinalities of lists, etc, so that the same triggers can be applied to multiple problem domains. This is not a strict *requirement* of the MLS triggers, but it enables portability of heuristic concepts across problem domains. A trigger might refer to the "quality" of a solution – this is a generic quantity that can be used for multiple problem domains, even though the method of evaluating this quality will be problem specific (and calculated with an update-memory module).

4.4.4.3 Types of response

There are seven different types of response:

- Terminate the heuristic;
- Learn something new by performing an update-memory analysis module (including evaluating the BSF);
- Change a memory element in the memory structures;
- Change the current solution by performing a change-current-solution module;
- Add or remove a search scheme module;
- Add or remove a response from a particular trigger;
- Activate or deactivate a trigger.

Note that apart from the termination response, all of these responses involve changing an MLS component. If a response attempts to modify an MLS component, then the appropriate parts of that component must be specified and present in the MLS heuristic. For example if a response modifies a memory element, this element must be defined. If a response adds an admissibility condition, this condition must be defined as a dependency.

A response has a single, optional, parameter, which specifies the MLS module or memory element on which it is acting. The implication of this is that a particular MLS heuristic implementation might have many versions of essentially the same response type, each acting on a specific module. This parameter is not modifiable like other MLS components; a response parameter cannot be changed, instead two separate responses of the same type would exist and would be swapped in and out of a trigger. The

benefit of this is that there are very few types of response that need to be considered, the only thing that changes is the module on which they are acting.

The following sections describe these response types in more detail. An important point is that these responses are performed in this order. One response could initiate an analysis to determine and update the relative performance metrics of different modules, and perhaps create a ranked list of diversification strategies. The next response could be to actually implement these strategies. Responses perform a single task, but multiple responses can be associated with each trigger, so the combination of these responses can create quite complex changes in the MLS configuration.

4.4.4.4 Response: termination

Every heuristic needs to have a **termination response** available through at least one trigger. The trigger that contains the termination response is commonly called the *termination criterion*. The termination response stops the search immediately and returns the best solution found so far. The termination response has no parameter.

4.4.4.5 Response: update-memory

The **update-memory response** performs one of the update-memory modules, which is specified in the parameter. This can be as complex as a sophisticated modelling exercise, or as simple as modifying a threshold. The end result of the update-memory module is a change in one or more elements of the memory structures.

The response can perform either an active update-memory module, which is performed anyway at the end of each iteration, or an inactive module.

This response is always performed before any of the other modification responses so that the updated memory elements can be used in subsequent responses and triggers.

When one of the search scheme parameters is modified, for example the candidate list size, or the temperature for Simulated Annealing, we may think of this as a "small move" in heuristic space; the heuristic is essentially the same but has different values for some parameters.

4.4.4.6 Response: change-current-solution

By default the target solution from the current iteration becomes the starting solution for the next iteration. If desired, however, this solution can be modified, or even substituted for a completely different solution. The **change-current-solution response** is performed by executing a particular type of module: a **change-current-solution module**; the module to execute is specified in the response parameter.

A *change-current-solution* module can take several forms:

- Running a construction heuristic to generate a completely new solution;
- Performing a local search-type "move" on the current target solution;

- Revisiting a previously-stored elite (or otherwise) solution from the memory structures;
- Performing some other procedure to generate a new solution.

A *change-current-solution* module might be as sophisticated as the execution of another entire MLS local search heuristic, with its own configuration. A similar outcome might have been possible by changing the current configuration, performing the search, and then changing back, but using a *change-current-solution* module may sometimes be a more elegant approach.

Note that *change-current-solution* modules are not executed automatically by the control system. They are only available to be executed by responses.

It can also be a completely new type of procedure. For example, path relinking uses the search history to create a new solution that is on a "path" between two previously visited solutions by varying particular solution elements. This type of "move" is best expressed as a *change-current-solution* module.

4.4.4.7 Response: add or remove module

This is two distinct types of response: **add module** and **remove module**. The modules modified in this response are those for the search scheme components or the update-memory component. This response creates what can be a major change in the configuration of the MLS heuristic; a "large move" in heuristic space, essentially creating what would appear to be a new heuristic or metaheuristic.

Basic examples would be changing the set of moves in the neighbourhood scheme. This is the metaheuristic approach from Variable Neighbourhood Search, and is used to diversify the search away from a local optimum. A more drastic series of responses would be to remove an admissibility condition and replace it with another admissibility condition.

The addition and removal of modules is performed by activating and deactivating these modules.

4.4.4.8 Response: add or remove a response

Responses correspond to specific actions to modify the MLS configuration, usually to either intensify the search or diversify the search. The appropriate response can depend on a number of factors, and can vary over time as more knowledge is gained. The ability to add or remove a response from a trigger allows variation in the way that this knowledge is applied, and provides the potential for a more intelligent search process. An alternative method of achieving the same goals would be to have multiple copies of a trigger, each with a different set of responses, and to activate a particular one at a time. For some purposes this may be the most appropriate method, however it is useful to have the option.

4.4.4.9 Response: activate or deactivate a trigger

Activating or deactivating particular triggers occurs when the MLS configuration changes. One method of switching between configurations is to have a trigger *A* with one set of responses that change the

MLS modules, activate another trigger *B* and deactivate this trigger *A*. When trigger *B* is tripped it would change the MLS modules back, reactivate *A* and deactivate *B*.

Note that if a trigger is activated or deactivated this takes effect as of the next iteration – so if a trigger that hasn't yet been checked is deactivated, the trigger is still checked and, if tripped, its responses are still performed this iteration, but it is not checked on the subsequent iteration.

4.5 The memory structures

The **memory structures** store parameters, lists, past solutions such as the best-so-far, and any other values or entities that need to be stored so that they can be used by the other MLS components and modules. We refer to a particular memory structure as a **memory element**.

Some memory elements are automatically maintained by the MLS framework, and are available to be used by any module. Others are updated by modules for the initialize-memory and update-memory components.

The memory structures described in this section are not an exhaustive list. Those described illustrate the potential of memory structures, however many more are possible, and often required by specific metaheuristics. MLS memory structures, when used in conjunction with appropriate update-memory modules, allows very flexible and powerful metaheuristic concepts. They also make the MLS framework very extensible; it is easy to create a new memory structure to keep track of some new idea, and then write update-memory modules, triggers and responses to utilize the new concept.

The **MLS memory structures** have multiple uses, storing things that need to be referenced:

- Parts of the search scheme control components: the move-list size, the examinations maximum, and the candidate list size.
- Additional parameters relating to specific search scheme modules or control system modules. For example the *temperature* in Simulated Annealing.
- Lists of solutions or solution attributes that are required by other MLS components. For example, tabu solution attributes in Tabu Search, and elite solutions that can be revisited in Path Relinking.
- Lists of other values that are needed by other MLS components. For example, multiple threshold values populated by a routine in List-Based Threshold Accepting.
- Automatically maintained and incremented counters. For example the iteration count, and the number of iterations since the best-so-far solution was improved.
- The best solution found so far, and the value of that solution.

The best solution so far and the automatic counters are maintained by default by the MLS framework. The other memory structures are specified by the various MLS modules, and the design of a new MLS module requires also designing how these memory structures are stored and updated.

4.5.1 Search parameters

There are two types of search parameters that are stored in the memory structures. The first are technically search scheme components, but their values are stored in memory rather than as modules since they are only single values. The move-list size, the examinations maximum and the candidate list size are stored in memory so that they are able to be modified the same as any other parameter by an update-memory module. The move selection order and examination order are components of the search scheme that are filled with modules, rather than memory parameters.

The other type of search parameter is quantities such as the *temperature* in Simulated Annealing. These are single-value quantities that are used by some module and need to be remembered and potentially updated by update-memory modules.

4.5.2 Lists

Many heuristics require lists of memory items to be retained. In general lists can consist of any particular type of item desired. The following are some examples in common use:

- Past elite solutions, such as in Path Relinking;
- Solution elements, such as those made tabu in Tabu Search;
- Values, such as the threshold values in List-Based Threshold Accepting.

Lists need not necessarily store single entities. For example, a duple entity could be defined that stores the name of a particular module, and some success metric for the last time it was utilized. This list could be used as part of a learning strategy. Consider that we have implemented a Tabu Search heuristic, and wish to determine which of 6 proposed tabu tenures should be used on the particular problem instance. We could specify a multi-phase MLS heuristic that runs the basic Tabu Search mechanism on a number of random starting solution for each tenure value, and stores the value of the best solution obtained after a set number of iterations in this memory structure. Then the MLS heuristic would choose the best tenure and continue for many more iterations for the main run of the heuristic. We are not suggesting that this approach would necessarily be a *good* metaheuristic – but it demonstrates the use of lists.

4.5.3 Automatic counters

The MLS framework automatically maintains a number of counters that are available to any module that needs to refer to them. No specific modules need to be specified to maintain these counters, they are included by default. The main use of the automatic counters is in trigger conditions. The following list is not exhaustive; any new counters that are deemed to be generally applicable and likely to be useful to multiple modules may be added to the system.

Automatically incremented counters:

- Total number of iterations
- Total number of iterations where a target solution was found
- Iterations since a target solution was found
- Iterations since a target solution was *not* found

- Total number of solutions examined
- Total number of admissible solutions added to candidate lists
- Number of iterations since the BSF was updated
- Number of times the BSF has been updated

For *each trigger* the following counters are automatically incremented:

- Iterations since the trigger was last tripped
- Iterations since the trigger was made active (0 if currently inactive)
- Iterations since the trigger was made inactive (0 if currently active)
- Number of times the trigger has been tripped
- Total number of active iterations
- Total number of inactive iterations

4.5.4 Search characteristics

The details of what happened in the current iteration of the search scheme are also automatically maintained, since these might be required by the triggers.

- Number of moves added to the move-list (note that this is not necessarily the same as the move-list size, if there were fewer moves available than the move-list size);
- Number of moves in the reduced move-list;
- Number of solutions examined for admissibility;
- Number of admissible candidates.

Other quantities required for specific modules are also maintained in the memory structures, but these must be maintained by user-created modules. For example, particular metrics relating to neighbourhood reduction processes.

4.5.5 Best solution so far

Some early heuristics did not actually require storing the best solution found so far, but all modern heuristics do, and it seems ridiculous not to, so this is an automatic part of the MLS framework. There are some design decisions, however. All solutions that are examined by the search logic and the admissibility conditions components of the search scheme have their fitness evaluated. The fitness function can be equivalent to, or different from, the problem objective function, and the real goal of the search is to find the best solution according to the objective function, so evaluating whether a solution is the best-so-far would make sense to be performed based on the objective function. However, this evaluation could be computationally very expensive, and avoiding this evaluation every iteration is one of the primary reasons why a different fitness function might be employed.

Whether to evaluate the best-so-far based on the fitness function or the objective function is a decision made by the **update-memory** modules, and is discussed in Section 4.4.3.3.

4.6 Summary of MLS components

We briefly summarize the types of components and modules in the MLS framework. Some modules are completely generic and can be reused regardless of the problem domain; others need to be redesigned for each new problem.

Note also that most modules have a status of either *active* or *inactive*. This allows an MLS heuristic to specify multiple options for a particular component, and then swap these in and out as desired.

4.6.1 Search scheme components

- **Neighbourhood scheme**.
    - **Set of move-types**. Whatever moves are appropriate for the problem. *Problem-dependent,* and needs to be created for each problem.
    - **Move-list size**. Can be unlimited or some fixed number. Is stored as a memory element, so can be manipulated by update-memory modules. *Problem-independent*.
    - **Move selection order**. The actual choices are problem-independent, but the logic used to do the selection is *problem-dependent* and needs to be created for each problem.
- **Neighbourhood reduction process**. *Problem-dependent*.
- **Fitness function**. *Problem-dependent*.
- **Admissibility conditions**. For most applications this will be *problem-independent*, however some specific admissibility condition modules could be designed that are *problem-dependent*.
- **Search logic**.
    - **Examinations maximum**. Can be unlimited or some fixed number. Is stored as a memory element, so can be manipulated by update-memory modules. *Problem-independent*.
    - **Candidate list size**. Can be unlimited or some fixed number. Is stored as a memory element, so can be manipulated by update-memory modules. *Problem-independent*..
    - **Examination order**. Since the moves that are being selected are simply in a list, this is *problem-independent*. The moves themselves are problem dependent, but the search logic simply sees “moves”.

4.6.2 Control system components

- **Generate-initial-solution**. *Problem-dependent*.
- **Initialize-memory**. Specific modules may or may not be *problem-dependent*.
- **Update-memory**. Specific modules may or may not be *problem-dependent*.
- **Change-current-solution**. Specific modules may or may not be *problem-dependent*. Change-current-solution modules are not executed automatically, they are executed when called by responses.
- **Triggers**. *Problem-independent*.
- **Responses**. *Problem-independent*, although the update-memory or change-current-solution modules that are called may be *problem-dependent*.

4.6.3 Memory structures

- **Best-so-far solution**. *Problem-independent*. The representation of the solution itself is problem dependent, but the BSF memory element simply holds a "solution object".
- **Parameters and thresholds**. These can take many forms but are *problem-independent*.
- **Lists**. These may be *problem-dependent* or *problem-independent*, depending on the application.
- **Search characteristics**. *Problem-independent*.

4.7 Examples of metaheuristics as MLS

We illustrate the MLS framework by describing how basic versions of four common metaheuristics could be expressed as MLS heuristics: random restart, Tabu Search, Simulated Annealing and Variable Neighbourhood Search. Descriptions of these metaheuristics can be found in Chapter 5. We also describe a new metaheuristic idea that MLS makes possible, which we call **Iterative Sampling Local Search**, which illustrates the power of some of the MLS components. We discuss these in the context of the Travelling Salesman Problem.

4.7.1 Random Restart as MLS

Random Restart repeatedly executes a basic Steepest Ascent search. Starting from a randomly generated solution, it moves to the best solution in the neighbourhood until a local optimum is reached, and then generates a new starting solution and repeats the ascent process.

The key characteristic of how random-restart is expressed in MLS is a change-current-solution module that is executed in response to a local optimum trigger.

The following modules specify how this basic version of random restart for the TSP could be expressed as MLS:

Search scheme modules:

- **Set of move-types**. Whatever move-types are appropriate for the problem. E.g. 2-exchange for the TSP.
- **Move-list size**. Unlimited.
- **Move selection order**. By implementation structure (it doesn't make a difference to the search since the size is unlimited, but this is fastest).
- **Neighbourhood reduction process**. None.
- **Fitness function**. Objective function, i.e. minimizing the total cost for the TSP.
- **Admissibility conditions**. If the trial solution has a better fitness than the current solution, then it is accepted.
- **Examinations maximum**. Unlimited.
- **Candidate list size**. Unlimited.
- **Examination order**. Random.

Control system modules:

- **Generate-initial-solution**. Randomly generate a solution, e.g. a random ordering of cities for the TSP.
- **Initialize-memory**. None, apart from the automatic initialization.
- **Update-memory**. The best so far is updated on the fitness function. There are no other update-memory modules performed every iteration, except for the automatic counters.
- **Change-current-solution**. A new random solution (random ordering of cities) is generated and set to be the starting solution for the next iteration. Note that this module is not performed automatically, but is available to be called by a response.
- **Triggers and responses**.
    - **Trigger-1 (*active*)**: the search found no admissible candidates (an apparent local optimum);
        - **Response**: Execute the change-current-solution module.
    - **Trigger-2 (*active*)**: the total iteration count (automatic memory element) reaches the *termination iteration threshold* (memory element);
        - **Response**: Terminate the heuristic.

Memory structures:

- **Best-so-far solution**. Fitness based.
- **Termination iteration threshold**. For the termination trigger.

4.7.2 Tabu Search as MLS

Tabu Search chooses the best move in the neighbourhood, whether it is improving or not. Since a neighbour is not required to be better than the current solution the search does not get stuck in local optima, however there is the risk of the search *cycling* among a few solutions. To prevent this, a tabu list is used, which stores attributes of recent moves or solutions. This tabu list prevents these attributes being repeated for a certain number of iterations, which guides the search into a different region of the search space. To prevent good solutions being passed over because of their tabu status, an aspiration criterion is commonly used that a neighbour is accepted if it is better than the best solution found so far.

The key MLS component through which Tabu Search may be expressed is the *admissibility conditions*. A neighbour is admissible if the move that led there did not modify any tabu solution elements *or* if it is strictly better than the BSF. To support this admissibility condition several more MLS modules are required. A *list* of tabu solution elements much be stored in the memory structures, and an update-memory module must maintain this list, adding new elements and removing expired elements. Basic Tabu Search keeps the same search scheme throughout the course of the heuristic; the metaheuristic elements are expressed through the memory structures rather than changes in the search scheme. The only trigger-response is the termination criteria, usually a fixed number of iterations.

The following modules specify how a version of basic Tabu Search for the TSP could be expressed as MLS:

Search scheme modules:

- **Set of move-types**. Whatever move-types are appropriate for the problem. E.g., 2-exchange for the TSP.
- **Move-list size**. Unlimited.
- **Move selection order**. By implementation structure (it doesn't make a difference to the search since the size is unlimited, but this is fastest).
- **Neighbourhood reduction process**. None.
- **Fitness function**. Objective function, i.e. minimizing the total cost for the TSP.
- **Admissibility conditions**. If the move contains no tabu attributes *or* if it is better than the BSF (note that this is a single admissibility condition). For the TSP we can make any edges that have just been added to the route tabu – so they cannot be removed until their tabu status has expired.
- **Examinations maximum**. Unlimited.
- **Candidate list size**. Unlimited.
- **Examination order**. By list order (makes no difference to the search since the search logic does not constrain the number of solutions examined, either through the examinations maximum or the candidate list size).

Control system modules:

- **Generate-initial-solution**. Any valid method for the TSP, e.g. a random ordering of cities.
- **Initialize-memory**. None, apart from the automatic initialization.
- **Update-memory**. The best so far is updated on the fitness function. Any solution elements that have been added to the target solution that are not in the starting solution are added to the tabu list, and given a tabu tenure equal to the value of the tabu tenure memory element. Any elements of the tabu list that have been on the list for a number of iterations equal to their tabu tenure are removed. For the TSP, the solution element would be arcs added to the route.
- **Change-current-solution**. None – the target solution becomes the starting solution for the next iteration.
- **Triggers and responses**.
    - **Trigger (*active*)**: the number of elapsed iterations reaches the termination iteration threshold memory element);
        - **Response**: Terminate the heuristic.

Memory structures:

- **Best-so-far solution**. Fitness based.
- **Termination iteration threshold**. For the termination trigger.
- **Tabu list**. Stores arcs, along with the remaining tabu tenure of each one.
- **Tabu tenure**. The number of iterations that new additions to the tabu list remain tabu.

4.7.3 Simulated Annealing as MLS

Simulated Annealing chooses the first admissible solution it examines. It always accepts improving solutions and accepts non-improving solutions with a probability that decreases over time. If the fitness value difference between the trial solution and the current solution is δ, then non-improving solutions ($\delta \geq 0$ for a minimization problem such as the TSP) are accepted with probability $e^{-\delta/T}$, where T is a parameter called the *temperature*. The temperature decreases by a fixed cooling rate a every n iterations. The heuristic terminates when the temperature effectively drops below some threshold T_{min}.

Within the MLS framework, Simulated Annealing is expressed primarily through an *admissibility condition*, which controls the acceptance of each trial solution based on a *temperature* memory element. This temperature is modified by a trigger-response that is tripped every n (memory element) iterations, and the response is to execute an update-memory module that reduces the *temperature* memory element by multiplication with the *cooling rate* memory element. A second trigger is tripped when the temperature falls below the *minimum temperature* memory element.

The following modules specify how a version of basic Simulated Annealing for the TSP could be expressed as MLS:

Search scheme modules:

- **Set of move-types**. Whatever move-types are appropriate for the problem. E.g. 2-exchange for the TSP.
- **Move-list size**. Unlimited.
- **Move selection order**. By implementation structure (it doesn't make a difference to the search since the size is unlimited, but this is fastest).
- **Neighbourhood reduction process**. None.
- **Fitness function**. Objective function, i.e. minimizing the total cost for the TSP.
- **Admissibility conditions**. If the trial solution has a better fitness than the current solution, then it is accepted. Otherwise it is accepted with probability $e^{-\delta / T}$, where δ is the amount of disimprovement in the fitness functions, and T is the temperature memory element.
- **Examinations maximum**. Unlimited.
- **Candidate list size**. 1, corresponding to a first-admissible search.
- **Examination order**. Random.

Control system modules:

- **Generate-initial-solution**. Any valid method for the TSP, e.g. a random ordering of cities.
- **Initialize-memory**. None, apart from the automatic initialization. Note that the memory elements *temperature*, *minimum temperature* and *cooling rate* are specified when the MLS heuristic is defined, rather than as initialize-memory modules.
- **Update-memory**. The best so far is updated on the fitness. There are no other update-memory modules performed every iteration, except for the automatic counters. However, there is an update-memory module that is called by a response that reduces the *temperature* memory element by multiplying it by the *cooling rate* memory element.

- **Change-current-solution**. None – the target solution becomes the starting solution for the next iteration.
- **Triggers and responses**.
    - **Trigger (*active*)**: The number of iterations since this trigger was last tripped (automatic memory element) reaches the *epoch-length iteration threshold* (memory element);
        - **Response**: Execute the update-memory module that reduces the temperature.
    - **Trigger (*active*)**: The *temperature* is lower than the *minimum temperature* (both memory elements);
        - **Response**: Terminate the heuristic.

Memory structures:

- **Best-so-far solution**. Fitness based.
- **Epoch-length iteration threshold**. For the temperature reduction trigger.
- **Temperature**. Used in the admissibility condition probability calculation. Set to a large value, e.g. 1000.
- **Minimum temperature**. Used in the termination criterion. Set to some value close to zero, e.g. 0.05.
- **Cooling rate**. Used in the update-memory module that reduces the temperature. A value between 0 and 1.

4.7.4 Variable Neighbourhood Search as MLS

Variable Neighbourhood Search performs a basic Ascent Search until it gets stuck in a local optimum, and then changes the move-set to enable the search to continue. The local optimum is only optimal with respect to the solution topology defined by the particular set of moves being employed. When this move set is modified, there may be new improving moves available.

There are many variations. To illustrate the way that multi-phase heuristics are expressed as MLS, we consider the case where there are two types of moves available; a simple type of move (2-exchange for the TSP) and a more computationally expensive type of move (3-exchange for the TSP). For most iterations the search uses the simpler 2-exchange move set, and when a local optimum is reached, a single iteration is performed with the more complex 3-exchange neighbourhood. This brief diversification phase will hopefully be sufficient to move the search into a new region of the search space. We further change the search during this diversification phase so that the best solution examined is chosen, regardless of whether it is improving or not. This ensures that the search is able to move away from the local optimum.

Within the MLS framework, Variable Neighbourhood Search is modeled as a multi-phase heuristic. In each phase a different configuration of modules is active, and the trigger-response mechanism is used to switch between them. We summarize the key features of the two configurations below:

Configuration 1 – local search phase

- Simple move-type, 2-exchange;
- Admissible if improving over current solution, best selected;

- Continues until a local optimum is reached.

Configuration 2 – diversification phase

- Complex move-type, 3-exchange;
- All solutions admissible, best selected;
- Executed for a single iteration.

The following modules specify how this version of Variable Neighbourhood Search for the TSP could be expressed as MLS:

Search scheme modules:

- **Set of move-types**. There are two move-types available. At the start of the heuristic the simpler 2-exchange move-type is *active* and the more complex 3-exchange move-type is *inactive*.
- **Move-list size**. Unlimited.
- **Move selection order**. By implementation structure (it doesn't make a difference to the search since the size is unlimited, but this is fastest).
- **Neighbourhood reduction process**. None.
- **Fitness function**. Objective function, i.e. minimizing the total cost for the TSP.
- **Admissibility conditions**.
    - **Condition-1 (*active*)**: If the trial solution has a better fitness than the current solution, then it is accepted.
    - **Condition-2 (*inactive*)**: All solutions are accepted.
- **Examinations maximum**. Unlimited.
- **Candidate list size**. Unlimited.
- **Examination order**. Random.

Control system modules:

- **Generate-initial-solution**. Any valid method for the TSP, e.g. a random ordering of cities.
- **Initialize-memory**. None, apart from the automatic initialization.
- **Update-memory**. The best so far is updated on the fitness function. There are no other update-memory modules performed every iteration, except for the automatic counters.
- **Change-current-solution**. None – the target solution becomes the starting solution for the next iteration.
- **Triggers and responses**.
    - **Trigger-1 (*active*)**: the search found no admissible candidates (an apparent local optimum);
        - **Response**: Deactivate the 2-exchange move-type.
        - **Response**: Activate the 3-exchange move-type.
        - **Response**: Deactivate admissibility condition Condition-1.
        - **Response**: Activate admissibility condition Condition-2.
        - **Response**: Deactivate this trigger (Trigger-1) for the next iteration.
        - **Response**: Activate Trigger-2 for the next iteration.

- **Trigger-2 (*inactive*)**: the number of elapsed iterations since Trigger-1 was last tripped (automatic memory element) reaches the *diversification phase iteration threshold* (memory element).
  - **Response**: Deactivate the 3-exchange move-type.
  - **Response**: Activate the 2-exchange move-type.
  - **Response**: Deactivate admissibility condition Condition-2.
  - **Response**: Activate admissibility condition Condition-1.
  - **Response**: Deactivate this trigger (Trigger-2) for the next iteration.
  - **Response**: Activate Trigger-1 for the next iteration.
- **Trigger-3 (*active*)**: the total iteration count (automatic memory element) reaches the *termination iteration threshold* (memory element);
  - **Response**: Terminate the heuristic.

Memory structures:

- **Best-so-far solution**. Fitness based.
- **Diversification phase iteration threshold**. Determines how many of the diversification phase iterations are performed. In this case, one.
- **Termination iteration threshold**. For the termination trigger.

Note that trigger-2 is only active for a single iteration before it is tripped. This is the length of the diversification phase that we have defined for this heuristic.

4.7.5 Iterative Sampling Local Search

The main motivation that drives metaheuristic design is the need to escape from local optima, which basic ascent searches quickly get stuck in. One way to do this, used by Tabu Search, is to choose the best solution from the neighbourhood, even if it is non-improving. So if there are no improving moves available, then the move that results in the least disimprovement in the objective function is selected. The problem with this method is that it leads to *cycling*, where a small set of solutions is repeated. Tabu Search counteracts this effect by making recent move attributes forbidden, so that if the search moves away from a "local optimum" it cannot immediately step back.

We propose an alternative method of preventing cycling, which we call **Iterative Sampling Local Search**. At each iteration, only a (possibly random) subset of the neighbours of the current solution are examined, and the best of these is selected as the target solution to move to. It is unlikely that the previous apparent local optimum will be present in the new neighbourhood subset, and if it is, it may not be locally optimal anymore. To the best of the authour's knowledge this metaheuristic idea has not been explored in the literature.

Within MLS there are three search scheme parameters that can be used to restrict the number of neighbours examined: the move-list size, the examinations maximum and the candidate list size. In this case we would achieve exactly the same effect by limiting the examinations maximum and the candidate list size, since all solutions are admissible, even if they are non-improving, and for the TSP all combinations of cities are feasible. In general, we achieve a faster, more efficient search by

restricting the move-list size, since the solutions do not need to be constructed and have their fitness evaluated first, which occurs with all solutions examined by the search logic. The difficulty with restricting the move-list size occurs when there is no easy way to generate random moves, for example with arc routing problems. In the case of the TSP, however, generating random 2-exchange moves is easy, so restricting the move-list is preferred over restricting the examinations maximum or the candidate list size.

The following modules specify how this version of iterated sampling local search for the TSP could be expressed as MLS:

Search scheme modules:

- **Set of move-types**. Whatever move-types are appropriate for the problem. E.g. 2-exchange for the TSP.
- **Move-list size**. Restricted to some number, which should be significantly less that the number of possible moves. For a 2-exchange move-type this can be calculated, and we might choose the move-list size to be 50% of the number of moves available.
- **Move selection order**. Random.
- **Neighbourhood reduction process**. None.
- **Fitness function**. Objective function, i.e. minimizing the total cost for the TSP.
- **Admissibility conditions**. All solutions are accepted.
- **Examinations maximum**. Unlimited.
- **Candidate list size**. Unlimited.
- **Examination order**. Random.

Control system modules:

- **Generate-initial-solution**. Randomly generate a solution, e.g. a random ordering of cities for the TSP.
- **Initialize-memory**. None, apart from the automatic initialization.
- **Update-memory**. The best so far is updated on the fitness function (which is actually the objective function). There are no other update-memory modules performed every iteration, except for the automatic counters.
- **Change-current-solution**. None – the target solution becomes the starting solution for the next iteration.
- **Triggers and responses**.
    - **Trigger (*active*)**: the total iteration count (automatic memory element) reaches the *termination iteration threshold* (memory element);
        - **Response**: Terminate the heuristic.

Memory structures:

- **Best-so-far solution**. Fitness based.
- **Termination iteration threshold**. For the termination trigger.

4.8 Discussion

One of the goals is to explicitly break each of the steps that occurs in various heuristics into discrete operations; the more finely packaged these steps, the more subtleties and variations are available to fine-tune hybrids of heuristics. This is one of the drivers behind such additional stages as the neighbourhood reduction process, and the multiple points of specifying the number of moves and solutions to examine.

4.8.1 Restricting the neighbourhood search

There are four points of control over the breadth of the neighbourhood examination that are built into the search scheme: the move-list size, the neighbourhood reduction process, the examinations maximum and the candidate list size. This allows fine control over the search process, and gives the ability to make trade-offs between search intensity and computational effort. The maximum neighbourhood size is only the first of these. The same algorithmic result can sometimes be obtained with MLS in a variety of ways. For example the following are equivalent (depending on the neighbourhood reduction process; a random reduction makes them equivalent):

- An unlimited move-list size, no neighbourhood reduction process, and an examinations maximum of 100.
- A maximum move-list size of 200, a neighbourhood reduction process that reduces the neighbourhood to 100, and unlimited examinations.

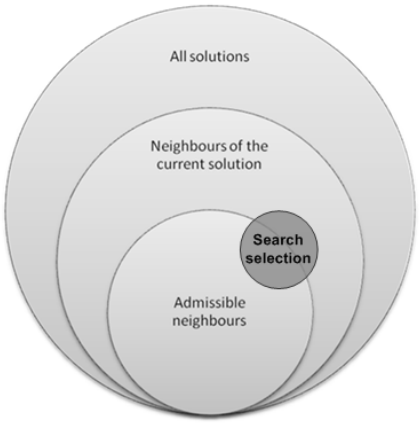


Figure 4.4: Partial solution hierarchy

Figure 4.4 shows a partial solution hierarchy. Neighbours of the current solution are a subset of all solutions, and only some of those are admissible. The search scheme, using the search logic, examines a subset of the neighbours, some admissible and some not admissible.

A key implication of the introduction of the examination of partial neighbourhoods, is that the term "local optimum" takes on a new meaning. We distinguish between a **true local optimum**, where there is no solution in the entire neighbourhood of the current solution that has a higher fitness, and an **apparent local optimum**, where there was no solution in those examined that had a higher fitness. All

true local optima are apparent local optima, but not all apparent local optima are necessarily true local optima.

This ability to control the execution of the search scheme with precision is one of the advantages of the Modular Local Search framework. Often in the literature the consideration of the search efficiency is ignored, and the whole neighbourhood is examined every iteration. Efforts to improve the efficiency of the search tend to concentrate on different types of neighbourhood. This removes one facet of control from the heuristic designer. It may either force neighbourhood size to be too small, by necessarily restricting the moves available, or force the iteration duration to be too long, by demanding explicit examination of too many neighbours.

We believe that this is not simply an "implementation concern". Deciding when to trade off between intensity of search and breadth of search is a key design consideration, and it may be a valid technique to introduce a diversification move-type that radically extends the neighbourhood, while limiting the number of these neighbours that can be examined, in order to make computation practical. Restricting the search space through measures like the maximum neighbourhood size, the reduction of the search space, the examinations maximum and the candidate list size is a valid tradeoff against the computational complexity of searching the neighbourhood at each iteration, and can lead to faster trajectories through the search space. It is worth reiterating that all of these quantities and restrictions are already present implicitly in existing local search heuristics, but they are usually set, by default, to *unlimited*.

The examination order consideration is also a contribution that is original, to the best of our knowledge, although it is a natural consequence of limiting the examination of the full neighbourhood. As soon as a limitation is placed on the *number* of solutions examined, the *order* of that examination becomes relevant. There are two points of control in the search scheme: the move selection order and the examination order. The obvious method is to select moves randomly, however for some problems there is no easy way to generate random moves, and explicitly enumerating all the moves so that one can be selected randomly from this list defeats the purpose of restricting the move-list size, especially for problems where the number of possible moves is prohibitively large.

4.8.2 Motivating metaheuristics – intensification and diversification

MLS is designed around being able to change heuristics, even part way through a run. The eventual goal of this research direction is semi-intelligent heuristics that can change themselves based on what they have learned. Why would we want to change the heuristic? Most reasons reduce to a need for either intensification or diversification. **Intensification** means to examine the current region of the search space more closely, and **diversification** means to move away from the current region of the search space more rapidly.

We have described that the control system, specifically the triggers and responses, is where the MLS configuration is changed, but have yet given little explanation of the reasons *why* such a change might be desirable. There are several reasons why it might be desirable to modify the configuration:

- The search is stuck in a local optimum → diversify;

- There has been no improvement for a long time → diversify;
- The search is cycling among the same, or similar, solution → diversify;
- The search needs to move more quickly → diversify;
- A potentially good region of the search space has been found→ intensify.

All of these reasons can be loosely grouped as requiring either intensification or diversification of the search. Diversification is more common, since it is used to improve the search, making it more appropriate for the current search state; leaving the search scheme as it is could be regarded as a trivial form of intensification.

It is interesting to note that just about every clever metaheuristic technique in the literature is an attempt to do one or more of these things: Tabu Search changes the tabu tenure to intensify the search in a good region or move out of a poor region more quickly, and has the tabu list at all in order to prevent cycling; Simulated Annealing has its annealing schedule to prevent being stuck in local optima and gradually lowers the "temperature" to intensify the search. What we are proposing is a framework which allows multiple techniques to be used, depending on which the most appropriate is at the time – all existing trajectory-based metaheuristics can then be seen as particular cases of an MLS approach.

4.8.3 Strengths of MLS

The MLS framework has many strengths that recommend it as a metaheuristic engine.

The main goal of the MLS system, and its major achievement, is that it allows extremely easy hybridization of multiple metaheuristic paradigms into a single algorithm. This occurs both by mixing different metaheuristics search scheme modules, but also by allowing complex multi-phase heuristics.

Because many of the operations within a local search heuristic which are usually implicit are made explicit in MLS, these can be deliberately modified. Aspects such as the examination order and candidate list size are usually ignored, however these can be the basis of subtle and sophisticated metaheuristic variations. A designer considering each component and module can discover new metaheuristic approaches that seem obvious when viewed as MLS, such as Iterative Sampling Local Search, discussed in Section 4.7.5.

One of the biggest benefits of this framework is that it allows the exploration of heuristic space in a systematic, rather than in an arbitrary, or ad-hoc, manner. The structure provided by the MLS components and modules provides a way to classify heuristics quite precisely, and develop measures for how similar various heuristics are, i.e. how close in heuristic space.

The complex memory structures and memory interactions provided by the update-memory modules, and the trigger-response mechanism, allow for heuristics to be self-adaptive and to make decisions on those adaptations based on the history of the search process – essentially creating a framework for learning. This suggests the possibility of higher-level control mechanisms. If heuristics are problem-solving tactics, then these higher-level heuristics (metaheuristics) may be thought of as strategies. In Chapter 8 we discuss several advanced applications of MLS, including some learning mechanisms.

The standardized architecture of MLS, and the fact that as much as possible is problem-independent, means that it is relatively straightforward to extend to new problems. We demonstrate this in later chapters where MLS is applied to several different problems. The control logic is almost completely reusable, and only the interfaces with the problem data need to be redesigned: things such as what is a solution, how is the objective function evaluated, what move-types are available, and how are these evaluated.

From an implementation perspective, the MLS framework also offers some advantages.

Along with the MLS markup language, the framework allows heuristics to be expressed and invented declaratively, rather than programmatically. A library of modules can programmed, and then these can be combined in an infinite number of ways to specify new, complex, heuristics *simply by listing the set of modules*.

When several different metaheuristics must be compared for the purposes of a "tournament", a notorious problem is determining how much of the difference in relative performances may be attributed to the superiority of the heuristic, and how much to the method of implementation. This is especially relevant for running times, and for heuristics that terminate after a certain time has elapsed. MLS provides a way to ensure that as much as possible of the heuristics' implementations is on a consistent basis, to make comparisons fairer.

The framework also enables significant savings on the time required to implement many different local search methods, since most of the code will be in common and reusable. Indeed, all of the control logic is reused, and simply the modules relating to the specific features of the new metaheuristic need to be designed.

4.8.4 Limitations of MLS

There are some limitations of the MLS framework.

It is very much designed as a practical system; it was inspired by reducing existing trajectory-based metaheuristics into a set of common components. For this reason there is not the theoretical structure that is claimed for alternative frameworks such as the Generalized Local Search Machines of Hoos and Stützle [146]. However, we believe that MLS is more suited for practical implementation of metaheuristics, and for hybridization and advanced control mechanisms.

The MLS framework is not suited for expressing *all* metaheuristics. It is limited to local search, i.e. trajectory-based methods. Population- and evolutionary-based methods have had considerable success in the literature, however these are outside the domain of the current form of MLS. We note, however, that many of the MLS concepts would have an applicability to population-based metaheuristics, and that it may be possible to extend the framework to a population-based version.

From an implementation perspective, an MLS version of a particular metaheuristic will probably not be as efficient computationally as a version of that heuristic programmed specifically, and optimized to take advantage of the specific heuristic structure. There is some overhead involved with maintaining the MLS structure, and the move-lists, etc. However, this difference in efficiency is not guaranteed, and

will depend on the skill of the coder of the MLS system and the coder of the specific metaheuristic. This possible difference is counterbalanced by the fact that the MLS system can be used repeatedly for many types of heuristic, so any optimization only needs to happen once. The overall development *and* execution time should be much lower with MLS because of this recycling of the main parts of the system.

Coda

▼ Summary

In this chapter we have defined the components of the Modular Local Search framework and have given some examples of how common metaheuristics can be expressed in this framework.

▼ Link

In the next chapter we catalogue metaheuristics from the literature, discussing variations and innovative features. We describe how these can be expressed in the MLS framework, and define a number of modules that may be used as building blocks in hybrids of these metaheuristic concepts.

CHAPTER 5

Metaheuristic Concepts

This chapter provides a detailed review of the main concepts on which most trajectory-based metaheuristics are based. The metaheuristics are grouped by their primary mode of operation: iteration, thresholding, memory, and other assorted concepts. Suggestions for how these concepts would be implemented under the MLS framework are presented and discussed.

5.1 Ascent Search

The most basic form of a local search heuristic is **Ascent Search** (**Descent Search** for minimization problems). This method, also known as the **Hill Climbing** heuristic, **First Improving Local Search**, **Depth-First Search** or simply **Local Search**, starts with a solution and moves to the first improving solution examined from the neighbourhood, continuing until it is stuck in a local optimum, i.e. there are no improving solutions in the neighbourhood, and then stops.

Steepest Ascent (also known as **Best Improving Local Search** or **Breadth-First Search**), is similar to Ascent Search except that the whole neighbourhood is examined and the *best* improving neighbour is selected. It can result in a shorter trajectory to a local optimum, but increases the processing time for each iteration.

Ascent Search has the tendency to get stuck in a local maximum of the underlying search topology. Depending on the problem, and the shape of the topology, it can still sometimes be reasonably effective. For example, if the neighbourhood scheme is such that every solution can be reached within one transforming move from the current solution, then the Steepest Ascent variation is guaranteed to arrive

at the global optimum within a single iteration. However, this iteration involves explicitly enumerating every solution, so it is usually not practical.

This basic iterative improvement procedure forms the core of every local search metaheuristic. They often add layers of sophistication in order to escape from local optima and continue the search, to diversify the search into other more promising areas of the search space, or to intensify the search around a promising area of the search space.

5.2 Iterated Search

Iterated Search procedures repeatedly apply a simple local search mechanism, choosing a different starting solution each time.

5.2.1 Repeated Local Search

Repeated Local Search is perhaps the simplest "metaheuristic", with the ability to escape from local optima and continue with the search. This is achieved by selecting a new initial solution and repeating the ascent, to a new local optimum, until some time limit or iteration count is reached.

If a population of solutions is available, then the simplest expression of repeated local search is **random restart local search**; simply randomly sampling another solution from the population. The ability to do this will depend on the problem. For example, a solution to the travelling salesman problem simply consists of an ordered list of cities visited, and any permutation of the cities will be a feasible solution. In this case constructing a random solution is trivial. In contrast, a solution to the Chinese postman problem consists of an ordered list of the arcs of a graph to be traversed, and not all sequences of arcs form valid tours; consecutive arcs must be adjacent. In this case constructing a random solution is considerably harder.

Local search heuristics start from a solution s and output a local optimum w, that is better than s. Many experimental results from the literature confirm that the quality of the initial solution strongly influences the quality of the local optimum. Therefore much effort has gone into ways of choosing the next starting solution.

There are two broad approaches to the choice of a new initial solution. Those methods which use a constructive heuristic to construct a new solution are commonly called **multi-start methods** (random restart is the simplest case of this). Other methods, known as **iterative local search** perform an operation to select a new initial solution by *perturbing* the local optimum reached by the previous iteration, in such a way that the search may continue.

5.2.1.1 Termination criteria

One design decision that is common to all the iterated search techniques is when to stop. In fact the need to include some method to terminate the heuristic is common to all metaheuristics that do not “get stuck” in local optima. Possible options include the following:

- A time limit on the computation. This is a natural termination condition, since a consideration of run-time is one of the motivating factors for using heuristics at all over algorithms (in the worst case explicit enumeration). A time limit also provides a way of balancing the number of iterations with the speed of the iterations, since either of these measures alone can be misleading as to the effort required. Time limits will, of course, depend on the speed of the machine running the heuristic, and the efficiency of the implementation.
- A limit on the number of iterations. The advantages of an iteration limit are that it allows comparison that is independent of the hardware or software implementation used to run the heuristic.
- A time or iteration limit after some significant event, for example the last improvement in the best solution.
- A local optimum is reached.

5.2.2 Iterated local search

Iterated local search (ILS) is a conceptually simple metaheuristic that iteratively applies local search to perturbations of successive local minima. The reasoning is that if the current solution is a local minimum then there are likely to be some portions of it that are optimized, and hopefully we will be able to keep these portions and continue to optimize the others, after escaping from the local minimum. From a search space perspective, we attempt to remain in a close region of the search space, to explore it further, rather than moving to a completely different region.

Lourenço et al. [176] describe four procedures that have to be implemented in an ILS heuristic: `GenerateInitialSolution`, `LocalSearch`, `Perturbation`, and `AcceptanceCriterion`, and define the following high-level architecture, which is widely reproduced within the literature:

Algorithm 5.1 metaheuristic ITERATED LOCAL SEARCH

s_0 = `GenerateInitialSolution`
s^* = `LocalSearch`(s_0)
repeat
  s' = `Perturbation`(s^*, *history*)
  $s^{*\prime}$ = `LocalSearch`(s')
  s^* = `AcceptanceCriterion`(s^*, $s^{*\prime}$, *history*)
until termination condition met
end

Within this basic framework, heuristics can either be very simple, or quite complicated. Lourenço et al. [176] give some tips for creating a basic version of ILS that usually leads to much better performance than random restart approaches: "(i) one can start with a random solution or one returned by some greedy construction heuristic; (ii) for most problems a local search algorithm is readily available; (iii) for the perturbation, and random move in a neighbourhood of higher order than the one used by the local search algorithm can be surprisingly effective; and (iv) a reasonable first guess for the acceptance

criterion is to force the cost to decrease, corresponding to a first-improvement descent in the set *S**." They consider the ILS approach to be a walk in the set of local optima, *S**.

5.2.2.1 Initial solution

The usual method to obtain an initial starting solution is to apply some constructive heuristic, perhaps a greedy method. A recent (2008) example of using a constructive heuristic can be found in Tang and Wang [233], who extend the Insert/Delete heuristic of Mittenthal and Noon [193] for the travelling salesman subset-tour problem with one additional constraint to the Capacitated Prize-Collecting Travelling Salesman Problem.

If there are no well-regarded constructive heuristics for a given problem domain then any random solution can suffice; Stützle [232] uses a random assignment of items to locations for the quadratic assignment problem, stating that no high performing construction heuristics are known. Even in recent research the random initial solution is chosen, an example from 2009 is Grosso et al. [129], who apply ILS to finding maximin latin hypercube designs[1], although they do give evidence that their approach could be improved by using a specialized heuristic to find the initial solution. Indeed, the premise of ILS is that starting each iteration from an already-promising solution is preferable to "any old solution", so this would seem to apply to the *initial* starting solution also.

Dong et al. [72] develop an ILS for the permutation flowshop problem with total flowtime criterion, and conduct an experimental analysis of the effect of different initial solution generation methods on 120 benchmark instances and 900 randomly generated instances. They used a number of different constructive heuristics from the literature, and also randomly generated instances. Figure 5.1 and Figure 5.2 are reproduced with permission from [72] and demonstrate that the constructive heuristics all performed much better than the randomly generated solutions, based on the average relative percentage deviation (ARPD) between the method and best solution known:

[1] The maximin LHD problem calls for arranging N points in a k-dimensional grid so that no pair of points share a coordinate and the distance of the closest pair of points is as large as possible.

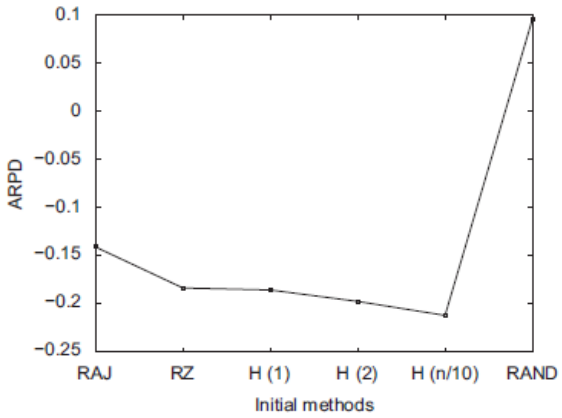


Figure 5.1: Performance of initial methods on benchmark instances [72]

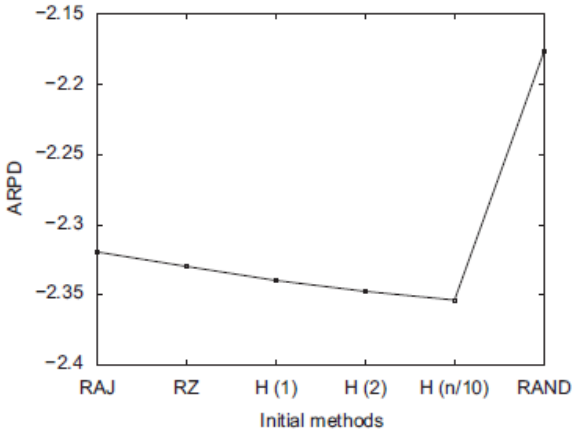


Figure 5.2: Performance of initial methods on random instances [72]

5.2.2.2 Local search component

The basic form of the local search component is simply ascent search, although there can be considerable differences in the performance of different neighbourhood schemes, and even hybrids with other metaheuristics. Quite sophisticated neighbourhoods can be constructed using expert knowledge of the problem domain. However, even quite basic neighbourhood schemes can provide good results.

Tang and Wang [233] utilize a sophisticated local search component which actually has three different search schemes. Every five iterations they run a basic form of the **Tabu Search** metaheuristic, which doesn't technically end in a local optimum, since Tabu Search just takes the best non-restricted move available, even if it worsens the objective value, so they let it run for 80 moves. Every 20 iterations they find a local optimum using a 3-opt neighbourhood, and the remainder of the time they use a fast 2-opt neighbourhood called *dynasearch*. The 3-opt neighbourhood is more computationally intensive, which is why it is only performed infrequently. This multi-phase approach is an interesting hybrid of the iterated local search, Variable Neighbourhood Search and Tabu Search heuristics.

5.2.2.3 Perturbation

The perturbation step transforms the current solution s into a new starting solution s'. It is used to escape from the local optimum resulting from the local search heuristic. A key consideration here is the extent to which the perturbation changes the current solution. If too great a change is made then there is a danger of losing the good features of the current solution, and moving into a completely different region of the search space. In this case the ILS heuristic may not have any advantage over a random restart method. However, if the perturbation is too small then the perturbed solution s' may be in what Lourenço et al. [176] call the same *basin of attraction* as s, i.e. they result in the same local optimum during the next local search phase. For illustration consider Figure 5.3. At the end of the local search phase we are sitting at the local maximum a. If the perturbation only moves the solution to b, then the next ascent search will end up at a again; a and b are said to be in the same basin of attraction. If,

however, the perturbation moves the solution to *c*, then the search will arrive at a new local maximum *d*, which in this instance is better than *a*, but in general this may not be the case.

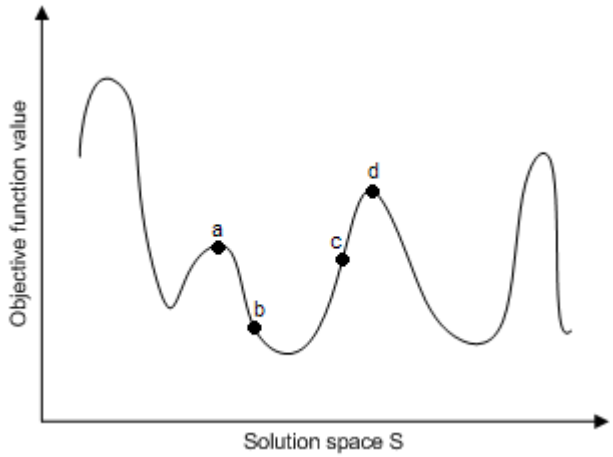


Figure 5.3: Pictorial representation of the perturbation step for iterated local search.

Lourenço et al. [176] point out that the local search should not be able to undo the perturbation, otherwise the same local optimum is likely to be repeated. They state that often a random move in a neighbourhood of higher order than the one used by the local search component can achieve good results.

Tang and Wang [233] utilize a feature called *backtracking*, which they borrow from Congram [55]. In their implementation of backtracking, whenever the current solution has not improved for a given number of successive iterations, it is pulled back to the best one found in history. This attempts to ensure that most of the search time is spent in promising regions of the search space. We can easily see that this concept can be extended to revisiting other promising regions of the search space; not necessarily the best solution found so far, but perhaps one of a set of elite solutions. This approach of the perturbation involving past solutions is explored in more detail in the discussion of **Path Relinking**.

The perturbation step is also sometimes known as a *kick*. This kick is usually performed by making the perturbation a small change on one or a few solution components, often at random. Tang and Wang [233] introduce a *guided kick*, which is essentially the evaluation of a set of possible small moves – adding a customer, removing a customer or swapping a customer – and then choosing the best of these moves. If none is improving then they perform a multi-customer swap and then restore feasibility if necessary. So the perturbation can sometimes be thought of as simply a modification of the neighbourhood for a move.

Lourenço et al. [176] refer to the *strength* of a perturbation as the number of solution components that are modified, for example in the TSP, it is the number of edges that are changed in the tour. They experimentally analyse perturbation strengths and show that for some problems, an appropriate perturbation strength is very small and seems to be independent of the instance size. However, for other problems the best permutation size is strongly dependent on the particular instance. They suggest using *adaptive perturbations*; changing the perturbation strength during the search.

Another approach described by Lourenço et al. [176] is more complex than simply making a move in a higher order neighbourhood. They describe a general procedure of subtly modifying the problem instance, for example via the parameters defining the various costs, then running the local search on this modified problem to obtain a new solution. This is the approach used by Baxter [18], which may be the earliest application of an iterated local search approach, although he simply called it *local optima avoidance*.

Dong et al. [72] also define a parameterized perturbation, which is a number of pair-wise swaps of the solution components. The number of swaps to perform is their perturbation strength. They perform an experimental analysis of various permutation strengths, and determine that the optimal level for their flowshop problem is between 4 and 7, based on the average relative percentage deviation (ARPD) between the method and best solution known, as shown below in Figure 5.4 and Figure 5.5, which are reproduced with permission from [72]:

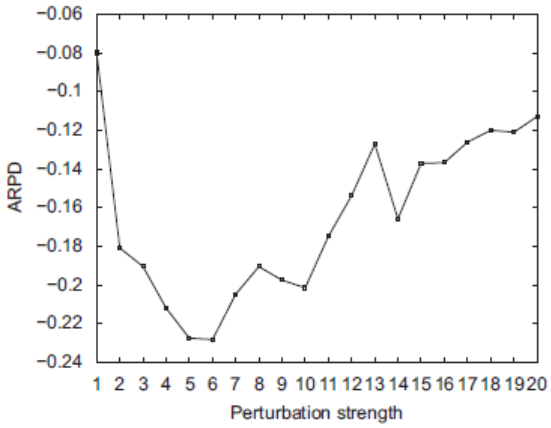


Figure 5.4: Performance of perturbation strengths on benchmark instances [72]

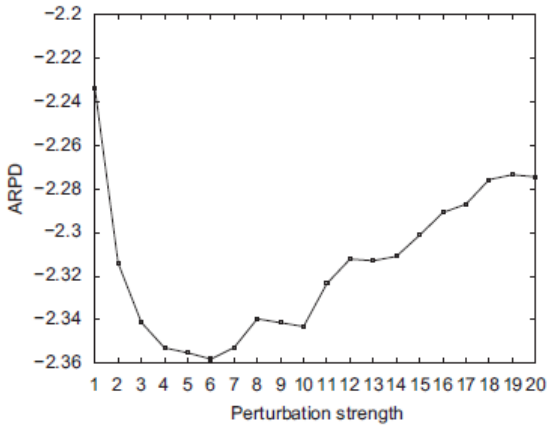


Figure 5.5: Performance of perturbation strengths on random instances [72]

5.2.2.4 Acceptance Criterion

Iterated local search may be thought of as a walk in the space of local optima [176], with the neighbourhood of this space being defined by the local search components executed at each step and the perturbation mechanism. The `AcceptanceCriterion` procedure then determines whether that "move" in local optima space will occur, or whether the search will consider another locally optimal "neighbour" of the current solution. This procedure can be used to control the balance of intensification and diversification of the search process.

A strong intensification effect is obtained by requiring that the neighbouring local optimum be better than the current local optimum, this would be analogous to a *first-improving* ascent search. We couldn't find an example of this in the literature, but it is easy to extend this analogy and consider a *best-improving* ascent search in the space of local optima, which would correspond to making all the available perturbations, and applying the embedded local search component to each one, then selecting

the best "neighbouring" local optimum. This type of search would be very computationally intensive, however.

Lourenço et al. [176] summarised a finding of Stützle [231], who analysed the run-time behaviour of ILS heuristics for the TSP, and concluded that the "better" acceptance criterion led to a type of stagnation behaviour for long run times, as expected for a strong intensification search.

To obtain a strong diversification effect, the neighbouring local optimum could always be accepted, regardless of whether it is better or worse than the current solution. The acceptance criterion in ILS is equivalent to the admissibility criteria in the MLS framework, and therefore any of the methods discussed in later sections for using these criteria are possible. For example, in Martin et al. [184,185] a hybrid of ILS and Simulated Annealing is used, where the acceptance criterion is based on the Simulated Annealing idea: $s*'$ is always accepted if it is better than $s*$, otherwise if is accepted with probability $\exp\{C(s*) - C(s*')) / T\}$, where C is the objective function value (based on a minimization objective) and T is a parameter called the annealing temperature.

Lourenço et al. [176], in their review of ILS for the 2003 *Handbook of Metaheuristics*, state that:

> "Most of the acceptance criteria applied so far in ILS algorithms are either fully Markovian or make use of the search history in a very limited way. We expect that there will be many more ILS applications in the future making strong use of the search history; in particular, alternating between intensification and diversification is likely to be an essential feature in these applications."

This has been true in that ILS perturbations are incorporated as key aspect of many modern hybrids that arise. A general trend seems to be that "pure" forms of the various metaheuristic families are starting to be become less prevalent in the literature; they are being replaced by sophisticated hybrids that take the features of many metaheuristics and mix them into adaptive, multi-phase approaches.

5.2.2.5 MLS interpretation

As with many metaheuristic concepts, Iterated Local Search may be interpreted in more than one way. The most natural interpretation is to consider a default MLS search scheme as the ILS `LocalSearch` component. We would then implement the perturbation using MLS triggers and responses. After the *local optimum trigger* is tripped, a *perturbation response* module is performed. This occurs within the MLS control system, leaving the search scheme as is. The perturbation response module acts as a `ChangeCurrentSolution` module, using whatever perturbation logic is specified by the heuristic. This module would also include the acceptance criterion logic to determine whether to accept the perturbed solution or not.

An alternative interpretation is to follow Lourenço et al. [176] and consider the ILS heuristic as a walk in local optimum space. In this case the search scheme would be quite non-standard, but there wouldn't be the trigger and responses described above. A "move" in this artificial solution space would be a compound process of performing a perturbation and then executing an embedded local search heuristic

(which could be an MLS heuristic itself). The ILS acceptance criterion would be expressed as part of the MLS admissibility criteria, and the search logic would be first improving.

Yet another interpretation that uses the MLS trigger-response model moves the responsibility for the perturbation from the control system `ChangeCurrentSolution module` to the search scheme (the inner local search process). When the *local optimum trigger* is tripped, the local optimum stays as the current solution for the next iteration, however the neighbourhood scheme and possibly the admissibility criteria are modified. The new neighbourhood is whatever move the perturbation consists of, and the admissibility criteria include the ILS acceptance criterion. In addition the triggers and responses are modified so that the previous search scheme is restored after one iteration (the perturbation phase). This type approach, where the MLS heuristic is shifted into a completely different configuration for a short diversification phase, is discussed in more detail in the section on Variable Neighbourhood Search (see Section 5.5.2).

The decision of which interpretation to implement would be based on the expected complexity of the components, and especially on whether any other hybrid modules need to be included.

5.2.3 Multi-start and GRASP

Multi-Start (MS) heuristics have the same basic concept as Iterated Local Search, which is the repeated application of a basic local search heuristic, starting from a different solution each time. Whereas ILS perturbs the local optimum reached at the end of each run to find the starting solution for the next run, MS generally begins each iteration from an independently chosen solution. In this sense MS can be seen as a special case of ILS, where the perturbation is the generation of a completely new solution. **Random Restart** is the simplest expression of this.

Martí [186], in his 2003 review, describes the basic structure of a multi-start heuristic, which we reformulate in Algorithm 5.2:

Algorithm 5.2 metaheuristic MULTI-START
    repeat
        s = `GenerateSolution`
        s^* = `LocalSearch`(s)
    until termination condition met
end

MS is conceptually a very simply metaheuristic. It is worth mentioning that the `LocalSearch` procedure has a trivial case of not changing s at all. In this case the multi-start method is simply sampling from the population of solutions, and evaluating the objective function.

When each starting solution is truly independent of those that have gone before and the search history, MS becomes a Monte Carlo method, and early papers studied the convergence properties, for example Solis and Wets [228].

Tsubakitani and Evans [237] develop a multi-start heuristic which they call **jump search**. They claim to have developed the first non-random multi-start procedure. Jump search constructs a number of "jump points" – potential starting solutions with mutually exclusive neighbourhoods. Local search is applied to the jump point with the best solution value, and so on down the list until the time limit is reached, or all the jump points are exhausted.

Because of its simplicity, multi-start is often enhanced with features of other metaheuristics, especially in the form of *adaptive memory*. Whereas iterated local search switches to a new solution by perturbing some solution components of the current local optimum, adaptive MS techniques often use other aspects of the search history or knowledge of the problem domain to construct the next starting solution.

5.2.3.1 Classification scheme for multi-start

Martí [186] proposes a classification of MS heuristics based on three elements. Each can be considered either "present" or "not present", but can also represent a whole range between these two extremes:

Memory. This is used to identify elements that are common to "good" previous solutions, using a definition of "good" that can include the objective function value, but also other factors such as diversity. Memory is a very powerful technique that was first fully developed under the Tabu Search framework, but which has become a pervasive concept. The other extreme of this, memory avoidance, can also have some value in terms of diversity.

Randomization. This refers to the degree of randomness in the generation method for the starting solutions. The approach can either be fully random, fully systematic or some combination. Systematically generated solutions are those that are constructed deterministically. Randomness can be an easy way of achieving diversification, but with no control over the solution it is not assured. Glover, in many papers (eg [120]), is an advocate of systematic heuristic methods that intelligently exploit knowledge of the problem and search history, over those that "resort to randomness".

Degree of rebuild. This indicates the solution elements that remain fixed from one iteration to the next. Martí [186] states that most applications build the solution at each generation from scratch, but more recent implementations have fixed some solution elements for a certain number of iterations. This aspect is where multi-start begins to look very similar to iterated local search, where only certain solution components are perturbed. The difference is mainly conceptual; in ILS the default position is to keep most of the solution elements the same, and change only the few necessary to perturb the solution sufficiently, in MS the default position is to construct a new solution, keeping only those few components that are deemed to valuable to lose. Martí [186] draws the analogy with Tabu Search, which also uses memory to target solution elements based on impact, frequency and recency, and with Path Relinking, which creates new solutions based on attributes of previous elite solutions.

5.2.3.2 Basic GRASP

One of the most widely applied multi-start heuristics is the Greedy Randomized Adaptive Search Procedure (GRASP), introduced by Feo and Resende [92,93]. Many studies have confirmed that the quality of the local optimum returned by a local search routine is strongly dependent on the quality of

the initial starting solution, so the attractiveness of GRASP heuristics is that they combine strong initial solutions with the improvement power of local search, and they are easy to hybridize with other features.

A common method of finding reasonably high-quality solutions for combinatorial optimization problems is to apply a greedy construction heuristic. Greedy methods start with an empty solution, and add solution components based on some ranking, until a complete feasible solution is obtained. Usually, greedy construction methods can generate one, or possibly several, solutions for a given problem; since the ranking of solution components is deterministic, it proceeds through the same selection order each time. GRASP heuristics overcome this limitation by introducing *randomness* into the selection of solution components. Instead of adding the best-ranked component, it selects randomly from a *restricted candidate list* of solution components. These solution components are ranked according to the incremental change in the objective function resulting from their inclusion (the greedy aspect), and this restricted candidate list is updated and the incremental benefits reevaluated after each selection (the adaptive aspect).

Resende and Ribeiro [219] give a basic structure of the greedy randomized adaptive construction process (the `GenerateSolution` function of the multi-start procedure defined above), which we reformulate in Algorithm 5.3:

Algorithm 5.3 procedure GREEDY RANDOMIZED ADAPTIVE CONSTRUCTION
    Solution ← ∅
    repeat
        Evaluate the incremental costs/benefits of the candidate elements
        Build the restricted candidate list (RCL)
        Select an element in the RCL at random
        Randomly select an element from the RCL and add it to *Solution*
    until *Solution* is a complete solution
end

GRASP heuristics are extremely easy to implement. Apart from the termination criterion, the only parameters that need to be tuned relate to the restricted candidate list of solution elements. According to Hoos and Stützle [146] there are two different mechanisms for defining the RCL: by *cardinality restriction* or by *value restriction*.

In the case of cardinality restriction, the RCL consists of the best k solution elements, based on their incremental costs. If $k = 1$ then the construction procedure is not random, and defaults to the special case which is the basic greedy construction heuristic.

In the case of the value restriction we consider, without loss of generality, a minimization problem where $c(e)$ is the incremental cost of incorporating element e into the solution, and we let c^{min} and c^{max} be the smallest and largest incremental costs. Then a solution element e is included in the RCL if, and only if, $c(e) \leq c^{min} + \alpha(c^{max} - c^{min})$. As with k in the case of the cardinality restriction, the smaller the

parameter α, the greedier the heuristic. For α = 1, the algorithm is equal to random choice of all elements, α = 0 corresponds to greedy construction.

We can classify the basic GRASP method using the classification scheme described above:

- **Memory**. Basic versions of GRASP have no memory, each new starting solution is constructed independently of those that have gone before.
- **Randomization**. GRASP does have a randomization component, and this can be strengthened or relaxed using the candidate list parameter, either k or α.
- **Degree of rebuild**. Basic GRASP has a full rebuild at each iteration.

Each of these areas gives opportunities to extend or hybridize GRASP, and this has been done in the literature. GRASP heuristics have been successfully applied to a large number of problems, due to their ease of implementation, and their tendency to produce high-quality solutions quite quickly. See Festa and Resende [95] for an annotated bibliography of GRASP up to 2001. Many GRASP applications focus on problem specific aspects, but there are some enhancements to the basic GRASP that should be highlighted.

5.2.3.3 Enhanced GRASP heuristics

Prais and Ribeiro [212,213] explore variation of the RCL parameter α. In their review, Resende and Ribeiro [219] summarise the experimental results of [212] into four different variation schemes for α:

- α self tuned during the heuristic run (this is known as **Reactive GRASP**);
- α randomly chosen from a uniform discrete probability distribution;
- α randomly chosen from a decreasing non-uniform discrete probability distribution; and
- fixed values of α, close to the purely greedy choice.

The experiments were conducted on four different optimization problems. The reactive GRASP most often found the best solutions, followed by random choice from the uniform distribution. Fixed values of α performed the worst.

Reactive GRASP selects α at each iteration from a discrete set of possible values. The scheme below was introduced by Prais and Ribeiro [213], and is quoted from Resende and Ribeiro [219]:

> "Let $\Psi = \{\alpha_1, \ldots, \alpha_m\}$ be the set of possible values for α. The probabilities associated with the choice of each value are all initially made equal to $p_i = 1/m$, $i = 1,\ldots,m$. Furthermore, let z^* be the incumbent solution and let A_i be the average value of all solutions found using $\alpha = \alpha_i$, $i = 1,\ldots,m$. The selection probabilities are periodically reevaluated by taking $p_i = q_i / \sum_{j=1}^{m} q_j$, with $q_i = z^*/A_i$ for $i = 1,\ldots,m$. The value of q_i will be larger for values of $\alpha = \alpha_i$ leading to the best solutions on average. Larger values of q_i correspond to more suitable values for the parameter α. The probabilities associated with these more appropriate values will then increase when they are reevaluated."

Prais and Ribeiro [213] update the selection probabilities after a fixed number of iterations, which they call the parameter *block-iterations* (they used *block-iterations* = 100). In terms of the classification scheme of Martí [186], reactive GRASP introduces a degree of *memory*, by using selection probabilities that change dynamically over the course of the heuristic, and it allows the level of *randomness* to be tuned to the specific problem at hand. The *degree of rebuild* is still complete.

Scaparra and Church [225] extend the reactive GRASP approach by making the parameter *block-iterations* variable. In the initial runs they use a large value for *block-iterations* in order to have a better estimate of the "goodness" of each α choice. As the algorithm proceeds they systematically reduce the value of *block-iterations* in order to intensify the use of good α values. They use two schemes for this: setting *block-iterations* to 75 and reducing it by 25 each time the probabilities are updated, and setting it to 100 and halving its value after each update, to a minimum value of 25. Their results suggest that the variable parameter performs better and is more efficient computationally than the fixed block-sizes, and the method of halving was superior to the fixed reductions.

Bresina [32] introduced the concept of **bias functions**, in the context of an iterative sampling search method called **heuristic-biased stochastic sampling**. These bias functions were extended to the restricted candidate list (RCL) of GRASP heuristics in Ribeiro and Hansen [27], and we paraphrase below the description found in the review of Resende and Ribeiro [219]. In the basic GRASP, each of the elements in the RCL has an equal chance of being selected for inclusion in the solution being constructed, however any probability distribution can be used to bias the selection towards particular candidates. Bresina [32] suggests a family of bias functions, based on the rank $r(\sigma)$ assigned to each candidate element σ:

- random bias: $\texttt{bias}(r) = 1$;
- linear bias: $\texttt{bias}(r) = 1/r$;
- log bias: $\texttt{bias}(r) = \log^{-1}(r + 1)$
- exponential bias: $\texttt{bias}(r) = e^{-r}$; and
- polynomial bias of order n: $\texttt{bias}(r) = r^{-n}$.

Let $r(\sigma)$ denote the rank of element σ and let $\texttt{bias}(r(\sigma))$ be one of the bias functions defined above. Once these values have been evaluated for all elements of the RCL, the probability $\pi(\sigma)$ of selecting element σ is

$$\pi(\sigma) = \frac{\texttt{bias}(r(\sigma))}{\sum_{\sigma' \in RCL} \texttt{bias}(r(\sigma'))}$$

As with all metaheuristics, many hybrids can be created by mixing the features of GRASP with those of other techniques.

5.2.3.4 MLS interpretation

Multi-start heuristics can be modeled in MLS very similarly to iterated local search. The basic multi-start heuristic, where a black-box construction technique is applied independently before each

application of the local search process, lends itself more strongly to the trigger-response MLS model, rather than the change-of-neighbourhood method. So after a *local optimum* trigger is tripped, the response is to execute a construction technique to set the new starting solution.

GRASP presents an interesting case. In general constructive methods are out of the scope of our interest; the MLS framework simply treats these as black-box functions. However, we have explored the details of the greedy randomized adaptive construction component of GRASP because this procedure itself can be considered a local search, and is able to be interpreted as an MLS heuristic.

Under MLS we can consider GRASP to be a two-phase local search heuristic. The first local search phase performs the construction, and the second phase performs the "traditional" local search improvement. Each of these phases has an MLS search scheme with a quite different configuration, which are switched via the trigger-response model. To model GRASP with MLS we modify the definition of a "solution" slightly to be any set of solution elements, including an empty set, and draw a distinction between a feasible solution and an infeasible (incomplete) solution.

We first consider the construction phase. The neighbourhood is defined by the single movetype of adding a solution element to the current solution, and the neighbourhood size is unconstrained. The restricted candidate list is modeled by a *neighbourhood reduction* procedure. All the solutions in the neighbourhood are evaluated and ranked based on their incremental improvement in the fitness function. Only those which meet the RCL conditions are accepted, using either cardinality-based or value-based restriction.

The search logic for the GRASP construction phase is to accept the first admissible solution examined from the reduced neighbourhood (*search size* = 1). The admissibility conditions are probability-based; for basic GRASP each solution has a probability equal to the inverse of the cardinality of the reduced neighbourhood, i.e. if there are m solutions in the reduced neighbourhood then each is accepted with probability $1/m$. For more advanced variations the probability can vary, for example as in reactive GRASP, or with bias functions. The search logic is to examine all the solutions until one is accepted, in random order. The search logic is set to sample with replacement.

There will be a number of triggers in effect. The most basic one waits until a feasible solution is constructed. When this occurs the response is to change the configuration of the MLS heuristic to the local search phase. Other potential triggers are waiting for particular iteration counts to be passed, at which time some of the parameters may be changed, as for reactive GRASP.

After the *feasible solution phase-change* trigger is tripped the neighbourhood scheme changes to the type of moves that the local search requires, along with the other MLS modules that the search scheme requires. The *feasible solution phase-change* trigger is deactivated and the *local optimum* trigger is activated. After it is tripped, the configurations are reversed again and the appropriate triggers are reactivated or deactivated.

5.3 Thresholding

Thresholding is one of the earliest metaheuristic ideas, where the search does not get stuck in local optima because non-improving are accepted if they meet certain conditions, which get stricter over the course of the heuristic. In general these heuristics are expressed in the MLS framework by manipulating the admissibility criteria, with the trigger-response procedure modifying the threshold.

5.3.1 Simulated Annealing

Simulated Annealing (SA) is one of the earliest, and most well-studied metaheuristics. Kirkpatrick et al. [160] and Černý [42] independently introduced Simulated Annealing as an application to combinatorial optimization of the Metropolis algorithm from statistical mechanics (see Metropolis et al. [188] for the original algorithm). The name derives from the analogy with the manner in which liquids or metals recrystalize in the process of annealing while cooling. If the initial temperature of the system is too low, or the cooling is done too quickly, the system may become quenched forming defects, which correspond to local optima in combinatorial optimization.

Simulated Annealing proceeds according to a standard local search approach, with the key difference that it offers a way to escape from local optima. At each iteration, a neighbouring solution is always accepted if it *improves* the fitness function, and accepted with a certain probability otherwise. Allowing worsening moves means that the algorithm can escape from local optima. This probability decreases as the search progresses, making worsening moves less likely near the end of the search. This has the effect of a broad search across the search space at the beginning of the process, and a more intense search as the heuristic identifies the most promising region to focus on.

The probability of accepting a non-improving move is based on the Boltzmann distribution of statistical mechanics. At each iteration, where the current solution is s, a neighbouring solution s' is selected at random. Let $\delta = z(s') - z(s)$ be the improvement in the objective function obtained by moving to the neighbouring solution. If $\delta > 0$ then the new solution is accepted; if $\delta \leq 0$ then the new solution is accepted with probability $e^{-\delta/T}$, where T is a parameter called the *temperature*. This acceptance function implies that small decreases in the objective function (for maximization problems) are more likely to be accepted than large ones. The temperature is initially set to an appropriately high value T_0; after a fixed number of iterations the temperature is decreased. Simulated Annealing stops when the temperature reaches a value close to zero, and no improving solution has been found – the system is "frozen". Therefore, although Simulated Annealing offers a way to escape local optima in the early stages, its eventual termination point is in a local optimum.

Algorithm 5.4 metaheuristic SIMULATED ANNEALING

```
s = GenerateInitialSolution
Select an initial temperature T = T0
Set temperature change counter t = 0
repeat
    Set iteration counter n = 0
    repeat
        Choose a random neighbour s' of s
        Calculate δ = z(s') – z(s)
        if δ > 0 then s = s'
        else if Uniform(0,1) < e^(-δ/T) then s = s'
        n = n + 1
    until n = N(t)
    t = t + 1
    Reduce temperature T = T(t)
until termination condition met
end
```

Eglese [80] describes the generic choices that must be made when designing a Simulated Annealing algorithm. The following aspects together constitute the *cooling schedule*:

- the initial value of the temperature parameter, T_0;
- a temperature function, $T(t)$, to determine how the temperature is to be changed;
- the number of iterations, N, to be performed at each temperature (the *epoch length*); and
- a stopping criterion to terminate the algorithm.

In their seminal paper, Kirkpatrick et al. [160] base their cooling schedule on the analogy with physical annealing. The initial temperature is set high enough to accept almost all possible moves, and a geometric temperature function is used: $T(t + 1) = a.T(t)$, where a is constant and $0 \leq a \leq 1$. Eglese [80] states that typical values of a used in practice lie between 0.8 and 0.99. Clearly, in this case the temperature slows its rate of cooling as it approaches zero. Other cooling schedules are possible, and are commonly considered, for example Lundy and Mees [177] propose a scheme where there is only a single iteration at each temperature, and $T(t + 1) = T(t) / (1 + B.T(t))$, where B is a constant. This represents a slower cooling than with fixed values of a and N. Bölte et al. [40] describe the only known annealing schedule that guarantees optimality: $T(t) = C / \log(t)$, where C is a constant. However this schedule requires run times that are "too long for most applications".

Pirlot [210] gives two commonly-used stopping criteria: if the value of the best solution has not improved by at least $q\%$ in the last k sets of N steps, or if the number of accepted moves is less than $q\%$ of N for the last k sets of N steps. Other researchers, such as Connolly [56], prefer to fix the number of iterations à priori to obtain algorithms with a deterministic run time.

The most common approach is for N to be a constant, perhaps proportional to the size of the problem instance or the neighbourhood, as described by Eglese [80].

5.3.1.1 Varations on cooling schedules

The performance of Simulated Annealing is known to be sensitive to the values of the control parameters used for the cooling schedule. Kouvelis and Chiang [161] undertook a computational study of various parameter setting for their SA algorithm for single row layout problems in flexible manufacturing systems, and concluded that the solutions are "highly dependent on the initial configuration", which is specific to each problem.

Wang and Wu [247] use response surface methodology to discover good parameters for the cooling schedule, combining this with a Steepest Ascent search using each parameter set as a "solution". Similarly, Bölte et al. [40] use genetic programming[1] to evolve good annealing schedules for a range of problem instances for the quadratic assignment problem. They found that temperatures that oscillated between two values were usually better than those that were monotone decreasing, and that the shape of this oscillation did not matter; they achieved good results with different shapes, such as sine and rectangular. These works were also some of the earliest examples of using one heuristic technique to optimize another, although they are used more as preprocessing research than as part of the heuristic itself.

A two-staged Simulated Annealing algorithm is one in which the nature of the search changes once the temperature has reached a certain threshold. This allows a faster heuristic to search at high temperatures, and then a more thorough configuration to continue at lower temperatures. Varanelli and Cohoon [242] discuss such a two-staged heuristic and propose a method for determining the change-over temperature; they note that if this threshold is too low then the heuristic can get trapped prematurely in an inferior solution, while if it is too high then computing time can be wasted by accepting too many unnecessary worsening moves.

Lin et al. [175] study adaptive variations of both Simulated Annealing and Threshold Accepting (which is introduced in the following section). They propose adaptive features for both the neighbourhood exploration and the temperature/threshold values. The adaptive neighbourhood is reminiscent of the ideas in reactive GRASP, where solutions that share features with previously examined good solutions are selected with a higher probability. The adaptive temperature/threshold is based on an analysis of how often the best-so-far solution value was improved at the previous temperature. They allow the temperature to increase and decrease, to intensify and diversify the search, according to a number of rules. Note that this type of adaptive behaviour is ideally suited to be expressed as MLS.

Azizi and Zolfaghari [12] also develop an adaptive SA, where the temperature control function cools suddenly with an improving move and slowly heats up (gets less strict) if needed:

[1] Genetic programming is a method that maintains a population of entities, in this case algorithms. By generations of breeding the "best" algorithms together, the quality of the population improves.

$T_t = T_{\min} + \lambda \ln(1 + r_t)$, where $T_{\min}$ is the minimum value the temperature can take, λ is a coefficient that controls the rate of temperature rise, and r_t is the number of consecutive non-improving moves at iteration t. The initial value of r_t is zero, and the initial temperature is $T_{\min}$. The minimum temperature prevents the probability function from becoming invalid when r_t is zero. The rationale of this approach is that there is a good chance of improving moves early in the search, so there is little need for a high temperature to push the search away from local optima.

5.3.1.2 Variations on the Simulated Annealing structure

Ishibuchi et al. [149] alter the basic Simulated Annealing heuristic. At each iteration, instead of randomly selecting a single solution from the neighbourhood and accepting it according to the acceptance rules, they randomly sample K solutions from the neighbourhood and test the best of these for acceptance. Note that this is a generalization of Simulated Annealing, since the basic algorithm is a special case with $K = 1$. They also propose a variation of this where K solutions are randomly sampled from the population, and test these one at a time. The first solution that improves the objective function is accepted, unless no solutions improve the objective function, in which case the best of the sample is tested for acceptance using the acceptance rules. This second variation may be considered as lying conceptually between the regular Simulated Annealing approach and the first variation. In terms of MLS, this would be considered changing the **search logic** through the **examinations maximum**.

Katayama and Narihisa [157] propose an interesting multi-phase SA algorithm, which they call **reannealing**. Essentially the Simulated Annealing progresses as usual, but with a fast neighbourhood examination scheme (they use a 1-opt neighbourhood for the unconstrained binary quadratic programming problem). At the end of the process the procedure is repeated, using a lower initial temperature and using the local optimum from the previous SA process as the starting solution for the new one. This reannealing process has two extra parameters in addition to those of the inner SA algorithm: the number of times to perform the annealing process, and the reduction rate to apply to the initial of the current process to get the initial temperature of the next process.

Ohlmann et al. [199] introduce a new heuristic called **compressed annealing** for a class of problems where the formation of a neighbourhood structure is impeded by a set of constraints, for example the travelling salesman problem with time windows (see Ohlmann and Thomas [200]). Neighbourhoods are recovered by relaxing the complicating constraints into the objective function within a penalty term. They refer to the penalty multiplier as “pressure”, hence the name “compressed annealing”.

5.3.2 Threshold Accepting

Dueck and Scheuer [77] propose the **Threshold Accepting** (TA) algorithm, which is a deterministic variation of the Simulated Annealing principle. Improving moves are always accepted, and non-improving moves are accepted only if they are don’t worsen the objective function value more than some threshold, which is typically set to be a deterministic, non-decreasing (for a maximization problem) step function based on the iteration count. The authors state that the “trivial” threshold schedule is “essentially best” and suggest that the performance of TA is basically insensitive to the threshold schedule. Their schedule is linear in the iteration count i, with $T_i = T_0 (1 - i/M)$, where M is

the limit on the number of iterations. Threshold Accepting is also known as **deterministic annealing**, for example by Gendreau et al. [111].

Dueck and Scheuer [77] present significant improvements over a Simulated Annealing heuristic for the travelling salesman problem, and also a faster runtime, due to the fact that TA does not need to compute random numbers or probabilities. Moscato and Fontanari [196] independently developed a Threshold Accepting heuristic, and reported less of an advantage over SA, but suggested that the probabilistic acceptance function is not a crucial element of the Simulated Annealing approach. Lin et al. [175] compare experimentally both standard and adaptive versions of Simulated Annealing and Threshold Accepting, and find that the TA versions perform better than their SA counterparts.

Dueck [75,76] further develops the Threshold Accepting approach by introducing two new metaheuristics, which he calls the **Great Deluge** algorithm and **Record-to-Record Travel**. These heuristics are based on new monotone threshold schedules. Great deluge starts with an initial water level, which may be interpreted as a minimum allowable solution value (in a maximization problem). The first allowable neighbour found prompts a move to that neighbour and an increase in the water level. This increase amount is the only tunable parameter in the great deluge algorithm, known as "up". One popular choice for "up" is that it should be somewhat smaller than one percent of the average gap between the value of the current solution and the water level. Note that the threshold in Great Deluge is judged against the actual solution value, rather than the amount of disimprovement, as in TA. The Record-to-Record Travel heuristic also uses a different quantity to compare against its threshold. It maintains a value "record", which is the value of the best solution found so far. It then allows solutions which are better than "record" minus some allowed deviation, which is, again, the only tunable parameter.

Hu et al. [147] disagree with the assertion of Dueck and Scheuer [77] that the threshold schedule is unimportant and develop a Threshold Accepting variation, which they call **Old Bachelor Acceptance**, where the threshold schedule is self-tuning and non-monotone. Their approach derives from an experimental examination, where they exhaustively examined all possible threshold schedules for some small problems they devised for this purpose. They found that many of the optimal schedules they found were non-monotone (able to increase as well as decrease), and they cite the work of other researchers who have reported effective non-monotone schedules for Simulated Annealing. In old bachelor acceptance the threshold changes dynamically, up or down, based on the perceived likelihood of being near a local maximum. They observe that if the current solution is better than most of its neighbours, then it will be hard to move to a neighbouring solution; standard TA will repeatedly generate a trial solution and fail to accept it. The authors describe the principle of *dwindling expectations*: after each failure, the criterion for “acceptability” is relaxed by slightly increasing the threshold T. After sufficiently many consecutive failures the threshold becomes large enough for the heuristic to escape. The converse of dwindling expectations they call *ambition*, whereby after each acceptance of the trial solution the threshold is lowered. They consider a number of different increase and decrease functions; the basic version is simply to use a constant value, but they also consider quite complicated functions that depend on the neighbourhood size, the number of iterations remaining, and the current threshold value. Old Bachelor Acceptance is an interesting and flexible generalization of

Threshold Accepting. A recent study by Ricca and Simeone [220] applies metaheuristics to a political districting problem, and finds that old bachelor acceptance outperforms both Simulated Annealing and two versions of Tabu Search.

Tarantilis et al. [235] propose a slightly different variation of an adaptive TA, which they call the **Backtracking Adaptive Threshold Accepting** algorithm. This heuristic is structured around an outer loop, which controls the modification of the threshold, and the inner loop that does a local search based on the acceptability conditions of that threshold. Basically if at least one acceptable solution was found at a given threshold, then the threshold is reduced, otherwise it is raised. The process in the outer loop is stopped when the maximum number of outer loop iterations has been reached, or when no feasible inner loop move can be found, even though the threshold has just been backtracked (raised). The threshold is reduced by multiplying it by a reduction rate r: $T(t + 1) = r\ T(t)$, where t in this case refers to iterations of the outer loop. A key aspect of this heuristic is that when the threshold is backtracked it is raised to a value *lower* than the value before the previous reduction; in this way the threshold tends towards zero. Specifically, $T(t + 1) = T(t - 1) - (1 - b)(T(t - 1) - T(t))$, where b is the percentage of threshold backtracking.

Tarantilis et al. [234] introduce the **List-Based Threshold Accepting Method**, which they describe on a VRP problem with the goal of minimizing costs, which we reframe as a maximization problem in the description below. This two-phase TA variation starts with an initial feasible solution and then generates a "threshold list". Starting from the initial solution s, a neighbour s' is generated and their objective function values are compared as $T_{new} = (z(s) - z(s')) / z(s)$. If T_{new} is positive and lower than the maximum element of the list, T_{max}, then T_{new} is inserted in the list. This is repeated until the list is filled with M threshold values. This initialization phase is then followed by the local search phase. At each iteration a neighbour solution s' is generated from the current solution s, and a new threshold value T_{new} is calculated using the formula above. If $T_{new} \leq T_{max}$ then the proposed solution s' is accepted and s is set to s'. T_{new} then replaces T_{max} on the list, and T_{max} becomes one of the other values on the list. In this way the list becomes more strict over time. This list-based method has only a single parameter, the list size M. Lee et al. [166] extend this List-Based Threshold Accepting method by selecting at each iteration the best of a sample of admissible neighbours, rather than the first admissible neighbour examined. This approach echoes the earlier method of Ishibuchi et al. for SA, which was discussed above, although they do not cite this paper.

5.3.3 MLS interpretation

The thresholding metaheuristics are expressed in the MLS framework through the *admissibility conditions* of the search scheme. A solution is an admissible candidate if the fitness function passes the specific condition.

The basic Simulated Annealing algorithm has an admissibility condition that a neighbour solution is accepted as an admissible candidate if the fitness function value is better than that of the current solution, or otherwise accepted with the Metropolis probability, which has a parameter *temperature*. There is no neighbourhood reduction phase, and the search logic is to select the best 1 of 1 (first admissible). The reduction of the temperature parameter is controlled through the trigger-response

model; when a certain *iteration count* trigger is tripped, the temperature parameter is reduced, according to the appropriate cooling schedule, which is part of the response. A second trigger is waiting for the termination condition, to terminate the heuristic; this can be an iteration count, a trigger count (number of temperature reductions), or a certain temperature being reached.

Most of the variations of Simulated Annealing simply utilise different logic for modifying the temperature, both when to modify it and how to modify it. All of these variations are expressed with different logic within the trigger and response modules. The Threshold Accepting variants have essentially the same structure as for Simulated Annealing, except with different admissibility conditions, and different response modules to modify those conditions.

Based on the literature review above, we define the following MLS modules.

Memory structures. These memory structures exist in the heuristic's memory. Not all heuristic implementations will utilize all memory elements, but they are present if needed. If an MLS module requires a particular memory structure, then this is listed. The following memory structures are those which can be specified by the user, and modified by the trigger-responses.

- Threshold for admissibility. Note that this also includes the temperature for SA.
- Cooling rate

The following memory structures are automatically generated and updated by the MLS control process, without having to be specified by the user.

- Iteration count
- Iterations since last tripping of each trigger
- Iterations since BSF was improved
- Size of neighbourhood in last iteration
- Size of restricted neighbourhood in last iteration
- Number of solutions examined in last iteration
- Number of admissible candidates in last iteration

We are able to define the MLS logic for several of the admissibility conditions, to demonstrate how MLS modules are defined. Algorithm 5.5 defines the admissibility condition for the Metropolis condition, which is the fundamental module that defines Simulated Annealing.

Algorithm 5.5 MLS admissibility condition METROPOLIS CONDITION

// The fundamental module of Simulated Annealing. This admissibility condition
// accepts the trial solution if it has a better fitness than the current solution, or if not,
// then with probability based on the size of the difference, using a "temperature"
// parameter.

Prerequisites: A memory parameter must be defined for the annealing temperature.
Input: $f(s)$, s', T *// The fitness of the current solution, the trial solution and the annealing temperature*

Calculate the fitness of the trial solution, $f(s')$
$\delta \leftarrow f(s') - f(s)$
if $\delta > 0$ **then**
    return *admissible*
else if `Uniform`$(0,1) < e^{-\delta / T}$ **then**
    return *admissible*
else
    return *inadmissible*
end
end

Algorithm 5.6 defines the admissibility condition for Threshold Accepting.

Algorithm 5.6 MLS admissibility condition BASIC THRESHOLD ACCEPTING

// The fundamental module of Threshold Accepting. This admissibility condition
// accepts the trial solution if the difference in fitness values between then current
// solution and the trial solution is less than the threshold.

Prerequisites: A memory parameter must be defined for the threshold.
Input: $f(s)$, s', T *// The fitness of the current solution, the trial solution and the threshold*

Calculate the fitness of the trial solution, $f(s')$
$\delta \leftarrow f(s) - f(s')$
if $\delta < T$ **then**
    return *admissible*
else
    return *inadmissible*
end
end

Algorithm 5.7 gives the admissibility condition for the Great Deluge algorithm.

Algorithm 5.7 MLS admissibility condition GREAT DELUGE

// The fundamental module of the Great Deluge algorithm. This admissibility condition
// accepts the trial solution if it is absolutely above the threshold, which is dynamically
// updated by other modules.

Prerequisites: A memory parameter must be defined for the threshold.
Input: s', T *// The trial solution and the threshold*

Calculate the fitness of the trial solution, $f(s')$
if $f(s') > T$ **then**
    return *admissible*
else
    return *inadmissible*
end
end

Algorithm 5.8 gives the admissibility condition for Record-to-Record Travel.

Algorithm 5.8 MLS admissibility condition RECORD-TO-RECORD TRAVEL

// The fundamental module of Record-to-Record Travel. This admissibility condition
// accepts the trial solution if the difference in fitness between the best-so-far solution
// and the trial solution is less than the threshold.

Prerequisites: A memory parameter must be defined for the threshold.
Input: $f(s^*)$, s', T *// The fitness of the BSF solution, the trial solution and the annealing temperature*

Calculate the fitness of the trial solution, $f(s')$
$\delta \leftarrow f(s^*) - f(s')$
if $\delta < T$ **then**
    return *admissible*
else
    return *inadmissible*
end
end

5.4 Adaptive Memory and Tabu Search

Tabu Search (TS) is a metaheuristic developed by Glover that uses a list of tabu solution elements to guide the search away from recently visited solutions, and hence away from local optima. The core concept is relatively basic, however Tabu Search has grown into a framework that encompasses many aspects of memory, often referred to as **Adaptive Memory Programming**. When discussing Tabu

Search it is necessary to highlight the enormous contribution and body of work of Fred Glover, and several of his regular research partners. It is impossible to overstate the importance and influence of their work, both on the Tabu Search family of metaheuristics, and on the field of metaheuristics in general. The 1997 book *Tabu Search* by Glover and Laguna [120] is possibly the single most comprehensive source of metaheuristic concepts and original ideas that exists. Thirteen years after its publication many of the concepts introduced are now fundamental pillars of the metaheuristic design toolbox, however new papers are still being published based on ideas in this work that haven't yet been picked up by the research community and are still fueling further progress. This work also discusses implementation concerns of adaptive memory concepts, such as data structures and efficiency, etc. Such considerations are outside the scope of this review, which focuses on *conceptual* features.

In this section on Tabu Search and adaptive memory, it is impractical to list every use or variation of the adaptive memory framework, therefore we briefly discuss the main features, and conceptual paradigms – drawing heavily from Glover and Laguna [120]. Our focus in this section is to catalogue the ideas that would be valuable in an MLS framework, rather than to give a full history of the development of every nuance of these ideas. Other citations are used where necessary to further develop ideas, however this section does not attempt to provide a comprehensive Tabu Search bibliography.

Glover and Laguna [120] describe the distinction between Tabu Search and Adaptive Memory Programming:

> "Tabu Search has been presented with two faces in the literature – one simpler and one more advanced. The simpler method incorporates a restricted portion of the TS design, and is sometimes used in preliminary analyses to test the performance of a limited subset of its components – usually involving only short term memory. The more advanced method embodies a broader framework that includes longer term memory, with associated intensification and diversification strategies. This second approach, due to its focus on exploiting a collection of strategic memory components, is sometimes referred to as *Adaptive Memory Programming* (AMP)."

This section begins with a description of the basic Tabu Search heuristic, and then we catalogue and discuss the other memory-based strategies.

5.4.1 Basic Tabu Search

In addition to the broader Tabu Search/Adaptive Memory Programming framework being so central to modern metaheuristic design, the Tabu Search metaheuristic itself has proven to be one of the most successful local search approaches, with countless applications; even quite simple implementations often perform competitively.

Tabu Search was introduced by Glover [117], in the same paper that he coined the term "metaheuristic". Tabu Search chooses the best available move, whether it is improving or not. This allows the heuristic to escape from local maxima, but creates the risk of cycling among solutions; to illustrate this concept consider that the current solution is a local optimum. There is no improving

move in the neighbourhood of the current solution (by definition), so the move which worsens the objective function the least is chosen. Depending on the move-types available, at this point there may be at least one improving move available, which is the move back to the previous solution, the local optimum. Without any further guidance, the search would step back and forth between these two solutions, which is called cycling. In general, cycling occurs whenever a sequence of solutions is necessarily repeated continuously.

In order to prevent this, a tabu list is used, which stores attributes of recent moves or solutions. The attributes remain on the list for a set number of iterations, the tabu tenure. These tabu restrictions tend to let the heuristic escape from local maxima because once the heuristic has left them it cannot return for a set number of iterations, unless some aspiration criterion is met. An aspiration criterion allows a tabu move to be admissible; a common criterion is that if the resulting solution is (strictly) better than the best obtained so far, then it is accepted. The tabu list is a form of *recency-based* memory, and may be considered *short-term memory*.

5.4.1.1 Tabu lists and tabu tenure

Standard tabu lists are usually implemented as circular lists of fixed length. The addition of a new tabu element to the list causes the oldest element to "fall off". In this sense tabu tenure and tabu list size are equivalent. However, fixed-length tabus cannot always prevent cycling; the tabu list prevents certain moves or solutions being revisited only for a certain period of time – if the search has not managed to escape the basin of attraction of the local optimum, then cycling can occur. In this case various techniques have been applied to vary the tabu tenure. Glover [118] (among others) proposed varying the tabu list length during the search, and Gendreau et al. [110] use a random tabu tenure within a small range. The possibility of a *dynamic* tenure, i.e. allowing different tabu tenures for different attributes, according to a rule, or randomly, is at odds with the concept of the tabu "list". For this reason, we prefer to refer to *tabu tenure*, rather than *tabu list size*.

Glover and Laguna [120] distinguish between random dynamic tenures and systematic dynamic tenures. *Random dynamic tenures* select a tenure between a minimum and a maximum, following a uniform distribution, either changing the tenure every iteration, or after a constant number of iterations. The simplest *systematic dynamic tenure* consists of a pre-defined sequence of tenures between a minimum and a maximum. The advantage is that it allows patterns that are unlikely to occur randomly, such as alternately increasing and decreasing values.

The systematic variation of the tabu tenure provides an easy-to-control method of intensification and diversification. A longer tabu tenure tends to increase the diversification, by forcing the search away from recently examined solutions, conversely a shorter tabu tenure lessens this effect, resulting in a more intense search.

5.4.1.2 Aspiration criteria

An *aspiration criterion* is a safety check to make sure that the tabu restrictions do not exceed their purpose, and prevent promising solutions from being considered. By far the most common aspiration

criterion in practice is to accept a move, which would otherwise be inadmissible because it is tabu, if the resulting solution value is better than the best solution found so far.

Glover and Laguna [120] suggest possible aspiration criteria in more detail:

- **Aspiration by default**. If all available moves are classified tabu, and are not rendered admissible by some other aspiration criterion, then a "least tabu" move is selected. For example if all moves are tabu, then those which are only have a remaining tabu tenure of one iteration could be considered, and the best of these chosen.
- **Aspiration by global objective**. A move aspiration is satisfied if the move yields a solution better than the best obtained so far. This is a common standard aspiration criterion.
- **Aspiration by regional objective**. A move aspiration is satisfied if the move yields a solution better than the best found in the region of the current solution.
- **Aspiration by search direction**. A move aspiration is satisfied if the move yields a solution that allows the direction of the search (improving or nonimproving) to continue.
- **Aspiration by influence**. A move aspiration for a low influence move is satisfied if a high-influence move has been performed since establishing the tabu status for the low influence move.

The concept of *influence* measures the degree of change in solution structure or feasibility, and may be thought of as a "move distance". Generally, moves of high influence tend to diversify the search more, modifying the solution in a greater way. Glover and Laguna [120] stress the importance of a balance between move quality and move influence; influential moves do not always result in solutions of high quality, but at different phases of the search may be desirable.

5.4.1.3 MLS interpretation

Although Tabu Search chooses the "best" solution at each iteration, the definition of best is subject to a type of *fitness function* interpretation, as described by Glover and Laguna [120]:

> "The meaning of *best* in TS applications is customarily not limited to an objective function evaluation. Even where the objective function evaluation may appear on the surface to be the only reasonable criterion to determine the best move, the non-tabu move that yields a maximum improvement or least deterioration is not always the one that should be chosen. Rather, as we have noted, the definition of best should consider factors such as move influence, determined by the search history and the problem context."

Influence – can be incorporated into fitness function, to help diversify the search. Update-memory procedures can update the tabu list, not just naively, but by revoking tabu status if desired, perhaps at a change of phase.

5.4.2 Candidate list strategies

Candidate list strategies are a method of reducing the number of solutions examined in a given iteration. This is necessary where the neighbourhood is large or when its elements are expensive to evaluate. However, they are not simply limitations on the number of solutions examined, but incorporate careful metaheuristic ideas to guide the search.

Glover and Laguna [120] describe in detail several classes of candidate list strategy, which we summarize below. They emphasize that "the effectiveness of a candidate list strategy should not be measured in terms of the reduction of the computational effort in a single iteration. Instead a preferable measure of performance for a given candidate list is the quality of the best solution found given a specified amount of computer time."

The motivation behind candidate list strategies influences one of the central features of MLS, although MLS has *multiple* points of control for limiting the number of solutions examined, and formalizes the role of each component. The relevant MLS components are the *move-list size*, the *neighbourhood reduction process*, the *examinations maximum* and the *candidate list size*. Each of the strategies is described, and then the MLS interpretation given.

5.4.2.1 Aspiration plus

The **aspiration plus** strategy establishes a threshold for the quality of a move, based on the search history. The procedure examines moves one at a time until a move satisfying this threshold is found, and then additional moves are examined, equal in number to the value of *plus*. This strategy may be thought of as a countdown after a solution better than the threshold is found. The best solution overall is then selected.

In order to ensure that neither too many nor too few moves are considered, the rule is qualified to require that at least *min* moves and at most *max* moves are examined. So if we let *first* be the number of moves examined when the aspiration criterion is first satisfied, then if *min* and *max* are not specified, then this strategy would consider *first* + *plus* moves. However, if *first* + *plus* < *min*, then *min* moves are examined, and if *first* + *plus* > *max* then *max* moves are examined.

This strategy is expressed by using a combination of MLS components. Note that the strategy says nothing about the admissibility of solutions; the threshold is not used to define the solutions that are admissible, but is simply used as a method of dynamically setting the number of moves examined. The strategy is designed to reduce the number of moves that are *evaluated* as solutions, so we can freely choose more to be in the move-list that are required, so long as only the appropriate number are actually evaluated.

The neighbourhood reduction process is the main component for this strategy. It performs a full evaluation of the moves in the move-list, comparing with the threshold, which is a memory element. When *first* + *plus* moves have been examined (and at least *min*) then the best move is returned as the reduced move-list. The search logic would then only have a single solution to examine. This implementation shifts most of the examination work away from the search logic to the neighbourhood

reduction process. A more general extension of this method would be to return all the *first + plus* examined moves as the reduced move-list. An advantage of this is that instead of evaluating the full solution, a "heuristic" method of evaluating the solutions could be used, perhaps a faster method than the full solution evaluation. The search logic would then complete the task of examining these solutions according to any admissibility conditions.

5.4.2.2 Elite candidate list

The **elite candidate list** strategy first builds a *master list* of moves by examining all, or a large number of, moves and selecting the k best (where k is a parameter). At each iteration the current best move from the master list is chosen to be executed, continuing until such a move falls below a given quality threshold, or a given number of iterations have elapsed. Then a new master list is constructed and the process repeats.

This strategy is unusual in that it stores actual *moves*. For example consider a small TSP route:

A – B – C – D – E – F – A

Two 2-exchange moves are to swap B and D, and to swap D and F. When evaluating these moves we obtain the potential routes

A – D – C – B – E – F – A and A – B – C – F – E – D – A

The elite candidate list strategy states that we use both of these moves on the master list. Let us suppose that we then select the B-D swap as the best move. The D-F swap is the next move on the master list, however after the previous move it results in the following:

A – D – C – B – E – F – A → A – F – C – B – E – D – A

This is quite a different outcome than the move was originally chosen for! Glover and Laguna [120] state that this strategy is motivated by the assumption that a good move, if not performed at the present iteration, will still be a good move for some further iterations. This may sometimes be the case but is not guaranteed, so careful monitoring of the list and re-evaluation of the solutions is necessary. Since re-evaluation of the solutions must occur at each iteration to choose the current best, this strategy seems to us not to achieve much, although is an interesting approach.

In terms of MLS this strategy would be implemented by maintaining a memory structure that contains the identifiers of the moves in the "master list". This list would initially be constructed as an *initialize-memory* module, and then updated when required with an *update-memory* module. An interesting aspect is that the actual problem move-types are implemented in these update-memory modules, rather than as part of the search scheme. The neighbourhood scheme move-type would be the selection of a predetermined move from the list in memory, so that the eventual search scheme move-list is simply the stored master list from memory. The trigger-response mechanism is used to control when the master list is replenished.

5.4.2.3 Successive filter strategy

Moves can often be broken into component operations, for example an exchange move might consist of an "add" component and a "drop" component. The **successive filter** strategy limits the number of moves evaluated by filtering only those that consist of good components. Glover and Laguna [120] give an example where there are 100 add possibilities and 100 drop possibilities, giving the number of add/drop combinations as 10,000 moves. However, by restricting attention to the 8 best add and drop moves, considered independently, the number of combinations to examine is only 64. This reduces the total number of evaluations from 10,000 to 264 (64 combined moves and 100 each of the basic moves).

Glover and Laguna [120] motivate this heuristic with a discussion of the TSP. They note that good solutions are often primarily composed of edges that are among the 20 to 40 shortest edges meeting one of their endpoints. Some studies have attempted to limit consideration entirely to tours constructed from these edges. The successive filter strategy offers a more flexible alternative by simply specifying that *one* of these edges should be incorporated as part of a move.

Successive filter strategies offer a perfect use-case for the MLS neighbourhood reduction process. A reduced move-list is produced from the move-list by deconstructing each of the moves into its component parts, and then evaluating the performance of these component moves. The best components are noted and only those full moves that are composed of the best components are included in the reduced move-list. All other parts of the MLS heuristic perform as usual, so this strategy would be an effective method to combine with other heuristic ideas. A major limitation is that the move-types must be amenable to decomposition into their component parts, so the feasibility of this strategy is quite problem dependent.

5.4.2.4 Sequential fan candidate list

The **sequential fan candidate list** strategy, described by Glover and Laguna [120], is a potentially very powerful metaheuristic concept that has not received much attention from the metaheuristic community. The basic idea is to generate p best alternative moves at a give step, and to create a "fan" of solution streams, one for each alternative. The several best available moves for each stream are again examined, and only the best p moves overall (where many or no moves may be contributed by a given stream) provide the p new streams at the next step.

One way of considering this is as a population-based method, and as such it is highly exploitable by parallel processing. From our perspective, the more interesting form of the strategy is as a *look-ahead* strategy. In this case a limit is placed on the number of iterations that the streams are created past the current iteration, then the best outcome from all these streams is used to identify a *best current move*, which is the first move in the stream that leads to the best outcome. Upon executing this move, the neighbouring solution becomes the new current solution, which is the source of p new streams and the process repeats. This look-ahead strategy is very computationally intensive, but can be very powerful, especially when used in combination with other methods for limiting the scope of the stream searches and making them quicker, or more heuristic. Of course the parameters such as the iteration limit and the number of streams, p, can be modified dynamically throughout the course of the heuristic.

This strategy highlights the flexibility of the MLS framework, especially the object-oriented implementation of it that we use, and which is discussed in Appendix B. Within MLS this strategy is best implemented by use of a complex *fitness function*. We noted in Chapter 4 that the fitness function may be significantly more complex than the objective function, even the result of the processing of another algorithm. In this case the fitness function is actually the evaluation of another (simpler) MLS heuristic! There is significant potential with this strategy to create a complex web of MLS heuristics that each have varying levels of complexity, and perhaps interact with shared memory structures. Even quite simple implementations would seem to have considerable power, although they will be computationally expensive. This factor makes them ideal to combine with the techniques of MLS that limit and streamline the number of solutions examined, possibly utilizing some of the other candidate list strategies described in this section.

5.4.3 Other adaptive memory concepts

There are many other examples of Adaptive Memory concepts. Glover and Laguna [120] provide an in depth discussion of many of these, for example, some of the following:

- **Strategic Oscillation**, which varies some threshold up and down so that the search may repeatedly arrive at solutions along the boundary from different levels. Strategic Oscillation can occur with any desired aspect of the search; it is often used to slightly relax feasibility conditions.
- **Path Relinking** is a method that takes advantage of remembered elite solutions. A "path" between two solutions can be found (for example the current solution and a previously visited solution. The motivation is that there are likely to be other promising solutions along this path. Path Relinking can be used as a metaheuristic in its own right, but is often combined as an intensification (or diversification) mechanism in other metaheuristics.

One example of adaptive memory is from Tang and Wang [233], who develop a speed-up mechanism for their enhanced dynasearch 2-opt neighbourhood by remembering the best h solutions found in the search history of their iterated local search heuristic (this list updates dynamically, so that the lowest-ranked item falls off when a new item is added). This list is then used to weight solution components (arcs in their case), which they use to map the search space to a reduced search space, upon which their algorithm proceeds much faster.

Ghosh and Sierksma [116] present Complete Local Search with Memory, which maintains several lists of promising and examined solutions in memory. See [17,143,201,222,243,254] for further examples of Adaptive Memory Programming.

5.5 Other trajectory methods

This section describes the other main trajectory-based metaheuristic paradigms.

5.5.1 Guided Local Search

Guided Local Search (GLS) was developed by Voudouris and Tsang [245], and has achieved some popularity as a metaheuristic concept. See [25,79,88,89] for examples of use.

The main mechanism of GLS is with a modified *fitness function*. Local search proceeds until a local optimum is reached. At this time the solution features of the local optimum are penalised by adding penalty terms to the fitness function, so that the local optimum is no longer locally optimal, with respect to the new fitness function. In this way the search is *guided* to other areas of the search space by modifying the search topology.

To be implemented in MLS, a new fitness function module would be required that considers the *penalties* associated with each solution component. In addition, memory parameters to hold these penalties would need to be specified, and an update-memory module to calculate and update the penalties after each iteration after a local optimum is reached would be needed.

5.5.2 Variable Neighbourhood Search

Variable Neighbourhood Search (VNS) is an extremely popular metaheuristic concept, originally introduced by Mladenović and Hansen [194]. See Hansen and Mladenović [135] for a good introduction and summary. See [57,68,100,134,136,141] for a selection of other papers using VNS, including many variations and extensions. VNS is also very suitable for hybridizing with other methods, for example [105,214].

The basic concept is that solutions that are locally optimal with respect to one neighbourhood structure are not necessarily locally optimal with respect to another. By varying the neighbourhood structures, then search is able to continue.

Hansen and Mladenović [135] describe several of the basic VNS heuristic structures, which we summarise below.

- **Basic Variable Neighbourhood Descent** has a set of neighbourhood structures N_k, for $k = 1,...,k_{max}$. The search starts with the first neighbourhood structure ($k = 1$) and selects a neighbour at random. If this solution is better than the current solution then the search moves there, otherwise k is incremented and a random point is selected from the next neighbourhood structure. If k reaches k_{max} then it is reset to 1, and if an admissible neighbour is found at any point the next selection starts from $k = 1$ again. The search continues until some stopping criterion is met.

- **Basic Variable Neighbourhood Search** proceeds the same as Basic Variable Neighbourhood Descent, except that the random solution chosen from the kth neighbourhood then has a local search method applied to it to arrive at a local optimum, and this local optimum is compared

with the current solution to determine if the search should move there or if k should be incremented.

In general, any technique which modifies the neighbourhood structure can be thought of as Variable Neighbourhood Search. In the experiments in later chapters we utilize a version where a specific neighbourhood structure is swapped into service as required as a diversification phase.

To express the above variations of VNS as MLS configurations, the primary mechanism would be the trigger-response model. The candidate list size would be set to 1, to accept the first admissible solution, with an *improving* admissibility condition, and the examinations maximum would also be set to 1 (only one solution examined). A trigger would be waiting for an apparent local optimum (the single solution examined was not acceptable), in which case the neighbourhood structure (the set of move-types) would be changed. Another trigger would be tripped if an admissible solution *was* found, in order to reset the neighbourhood structure back to $k = 1$.

5.5.3 Hyperheuristics

Hyperheuristics are an interesting metaheuristic paradigm that shares many similarities with MLS, both in mechanism and in motivation. Burke et al. [35] define hyperheuristics as the procedure of "using (meta-) heuristics to choose (meta-) heuristics to solve the problem in hand."

Burke et al. [36] present a good summary of the main ideas, and summarise the hyperheuristic concept with the following steps:

Algorithm 5.9 procedure HYPERHEURISTIC

Step 1. Start with a set H for heuristic ingredients, each of which is applicable to a problem state and transforms it to a new problem state. Examples of such ingredients in bin-packing are a single top-level iteration of 'Exact Fit' or a single top-level iteration of 'largest first, first fit';

Step 2. Let the initial problem state be S_0

Step 3. If the problem state is S_i then find the ingredient that is in some sense most suitable for transforming that state. Apply it, to get a new state of the problem S_{i+1}

Step 4. If the problem is solved, stop. Otherwise go to Step 3.

end

One of the motivations for hyperheuristics is to assist in the design of metaheuristics that are suited for a problem, when no problem-specific knowledge is available. Researchers typically tailor metaheuristics closely to problem domains to achieve high levels of performance; the goal of hyperheuristics is to be able to apply a more general technique that adapts itself to the problem.

These goals are very similar to those of MLS. In fact, it could be argued that MLS is actually an implementation and refinement of a **hyperheuristic** framework. "Heuristics choosing heuristics" is precisely what is made possible by MLS, and is demonstrated in Chapter 8 with some of the advanced applications.

In Section 8.3 we introduce a similar concept: Adaptive Diversification Local Search, which has a mechanism for selecting “the ingredient that is in some sense most suitable for transforming the search state”, and we introduce a learning mechanism that makes this judgement get progressively more “suitable”. In Section 8.2 we introduce the MLS Design Problem, which uses one higher-level MLS metaheuristic configuration to guide the design and development of a lower-level heuristic to solve the actual ASRP problem.

In its earliest stages hyperheuristics were mainly ways of combining low-level "heuristic ingredients" that consisted on constructive or perturbative elements. However, much of the recent hyperheuristic literature seems to be converging with the approach of MLS. Bai et al. [14] apply both metaheuristics (GRASP, Simulated Annealing, and Multi-Start) to a produce allocation problem, along with several varieties of hyperheuristics and found that the hyperheuristics found the best solution on most instances. The hyperheuristics they used combined elements of Tabu Search and Simulated Annealing in various types of hybrid, for example using both a tabu list to control which moves are considered, and a simulated annealing probability for acceptance of the resulting solution. Note that these metaheuristic control elements are acting to "evolve" a particular heuristic for the underlying allocation problem, but the evolved heuristic is composed of low-level heuristic elements. In contrast, MLS operates at a higher level of abstraction, where the underlying heuristic being assembled is itself a full version of MLS. MLS actually allows any number of these levels of abstraction, so in the extreme case we could have heuristics designing heuristics that design heuristics that solve the problem. It is not clear that there would be any benefit of this type of layering; in principle it creates an extremely flexible and powerful system, but in practice the computational requirements would be so extreme as to prevent it functioning effectively.

Hyperheuristics are an interesting approach that seem to have many of the similar goals to MLS, but approach the task somewhat differently; using metaheuristic principles to design combinations of low-level heuristics, whereas MLS attempts to design other high-level metaheuristics. They are a subset of the options available in MLS, and can be expressed using appropriate triggers and responses, and carefully designed memory structures and memory-update modules.

Coda

▼ Summary

In this chapter we have explored some of the common metaheuristic paradigms that exist for trajectory-based methods, and have given descriptions of how these could be formulated as MLS heuristics. This concludes Part II.

▼ Link

Part III applies MLS to the ASRP; multiple heuristics are developed and extensively tested.

Part III
Experimentation and Analysis

Overview of Part III
Experimentation and Analysis

Part III explores the uses of MLS, using the ASRP as a test problem.

Chapter 6 defines a number of MLS heuristics, based on the standard metaheuristic paradigms of Steepest Ascent, Simulated Annealing, Tabu Search, and Variable Neighbourhood Search. We explicitly formulate all the modules required for these heuristics, and these become the foundation of the MLS "toolbox". Extensive computational tournaments provide a large dataset of results and these are modelled against a set of problem characteristics. Several hybrids are developed to demonstrate the hybridization capability of MLS.

Chapter 7 contains a side investigation into methods to develop interesting sets of problem instances. A number of heuristics are developed and tested on a large number of problem instances. A brief study of using MLS to *design* problem instances that exhibit desired properties shows promise.

Chapter 8 contains a brief discussion of ways to use MLS to develop advanced metaheuristics. Two examples are demonstrated: using MLS to design other MLS heuristics, as a "meta" strategy, and using the memory structures of MLS to adaptively modify the structure of the heuristic, with elements of *learning*.

Chapter 9 concludes with a discussion of directions for future research.

CHAPTER 6

Applying MLS to the ASRP

In this chapter we perform extensive computational experiments with Modular Local Search (MLS) heuristics on the Arc Subset Routing Problem (ASRP). A number of standard metaheuristics are formulated and tested, and some basic hybrids are demonstrated. Some attention is given to new methods of predicting which of two heuristics will perform better on a given problem instance, based on an analysis of the problem characteristics of that instance.

6.1 Introduction

Recall that in Chapter 4 we introduced the MLS framework for metaheuristics, and in Chapter 5 we described in detail how many of the existing metaheuristic paradigms could be expressed as MLS heuristics. In Chapter 3 we defined the ASRP and performed a preliminary investigation of this problem, using traditional techniques and constructive heuristics. There are three objectives in this chapter.

The first objective is to demonstrate the use of the MLS framework to express some common metaheuristics: Steepest Ascent, Simulated Annealing, Tabu Search, and Variable Neighbourhood Search. We develop variations on these heuristics and compare them to each other in computational experiments. The focus here is to illustrate in practice how MLS can be used to construct heuristics.

The second objective is to extend these "regular" heuristics and to construct some basic hybrids of these. Again, the focus is to demonstrate the ease with which this can be performed with MLS, rather than attempting to find the best possible hybrids. We consider two types of hybrid. The first type

combines the primary search scheme characteristics of two source heuristics; we construct a hybrid of Simulated Annealing and Tabu Search that uses the admissibility conditions of both simultaneously to determine whether a given neighbour is admissible. The second type are *multi-phase* hybrids that change their structure in a significant way in response to certain events during the search. These types of hybrid are typically quite complicated to implement, however the MLS structure is relatively simple; all that is required is the specification of appropriate triggers and responses to completely change the structure of the heuristic. We propose hybrids of Simulated Annealing with Variable Neighbourhood Search, and Tabu Search with Variable Neighbourhood Search.

The final objective concerns the analysis of the experimental results. In the application of metaheuristics to real-world problems a major factor is choosing *which* metaheuristic to use to solve a given problem. This decision is easy if there is a particular metaheuristic that is known to dominate; if a particular metaheuristic outperforms all other metaheuristics on most problem instances then this technique can be safely chosen. However, the situation is more complex if there are two or more metaheuristics that perform well, each on different problem instances. In this case it is natural to ask whether it is possible to analyze the problem instances beforehand and choose the metaheuristic that is most suited for that type of problem instance.

To develop such a system would seem to be a significant research project in its own right. In this chapter we attempt to provide some of the foundation for such a system by demonstrating the validity of such an approach. Our hypothesis is that it is possible to use problem instance characteristics that are calculated à priori to predict which of two heuristics will perform better.

6.2 Problem instance design

We build on the methods developed in Chapter 3, utilizing grid graphs. The purpose of the problem instance design is to develop a large set of problem instances for the experimentation in this chapter. It is desirable to create a set of problem instances with quite diverse problem instance characteristics.

6.2.1 Graph generation methods

For the reasons presented in Chapter 3.4.1, all experimentation on the Arc Subset Routing Problem is performed on grid graphs, with arcs of unit cost. For our purposes the key advantage of grid graphs, as discussed below in Section 6.2.4, is that they allow the calculation of many different graph characteristics.

Graphs are generated with a specified density, which is a property unique to grid graphs. Let A^c be the set of arcs in the complete grid, and $A \subseteq A^c$ be the subset of arcs in the problem instance graph. The **graph density** $D = |A|/|A^c|$ is the proportion of arcs from the corresponding complete grid that are included in the graph. An $m \times n$ complete grid has $|A^c| = m(n-1) + n(m-1) = 2mn - m - n$ arcs. For example, a 4×4 complete grid has 24 arcs; a graph with 50% density would have 12 arcs. Of course, all graphs must be connected.

In the preliminary investigation of the ASRP we used grid sizes of 10×10. For subsequent experiments, graphs based on size 15×15 grids are used. Graphs are randomly generated using three different generation schemes, based on the GRIDGROW and GRIDSELECT methods of Chapter 3.

6.2.1.1 The GridDeselect generation method

In the preliminary ASRP investigation the GRIDSELECT graph generation method was introduced. This method randomly selects the appropriate number of arcs from the complete grid and then checks to see if the graph is connected. If the graph is disconnected it is discarded and the process repeats until a connected graph is obtained. When the density is low the GRIDSELECT generation method can be quite computationally expensive, as the probability of obtaining a connected graph decreases as the density decreases. The advantage of the GRIDSELECT method is that tends to create a homogeneous graph; the arcs are spread evenly across the grid, rather than being more concentrated in one region.

We introduce the **GRIDDESELECT** graph generation method, which was also used by Johnston and Chukova [151]. This method starts with a complete grid and randomly deletes arcs until the desired density is achieved, ensuring at each step that an arc is not deleted if its deletion would disconnect the graph.

Algorithm 6.1 procedure GRIDDESELECT
// This procedure is a method for generating a connected grid-based subgraph of
// a desired density by starting with a complete grid and iteratively removing arcs

Input: A^c, D *// The set of arcs in the complete grid and the desired density*
Output: A *// The set of arcs in the generated subgraph*

$A \leftarrow A^c$
repeat
Randomly choose an arc $a \in A$
Check if $A \setminus a$ results in a connected graph
if $A \setminus a$ results in a connected graph **then** delete a from A
until $|A| = D \times |A^c|$
end

This method still results in a homogeneous graph, but is much more computationally efficient that the GRIDSELECT method.

6.2.1.2 The GridGrow-*k*-Seeds generation method

The preliminary investigation introduced the GRIDGROW graph generation method. This method starts with a single random arc from the complete grid, and then iteratively adds arcs until the desired density is achieved, ensuring at *each step* that the graph is connected. So an arc is only added to the graph if it is adjacent to an already-included arc.

We develop this concept further by introducing the concept of *seeds*. The **GRIDGROW-*k*-SEEDS** generation method is a hybrid of the GRIDSELECT and GRIDGROW methods from the preliminary investigation. The initial step randomly includes k arcs from the complete graph; these are the *seeds*. At each iteration an arc is randomly selected from the complete grid and included in the graph if it is adjacent to an already-included arc; this is the similarity to the GRIDGROW method. However, at the end of this process, when the desired density is achieved, the resulting graph may not be connected. If this is the case the graph is discarded and the process repeats until a connected graph is obtained. This is the similarity to the GRIDSELECT method.

Algorithm 6.2 procedure GRIDGROW-*k*-SEEDS
// This procedure is a method for generating a connected grid-based subgraph of
// a desired density by with k seed arcs from the complete grid and iteratively adding
// adjacent arcs. If the resulting graph is disconnected then the procedure repeats.

Input: A^c, D *// The set of arcs in the complete grid and the desired density*
Output: A *// The set of arcs in the generated subgraph*

repeat
  $A \leftarrow \varnothing$
  Randomly choose k seeds arcs a_1, a_2,..., a_k from A^c
  Add arcs a_1, a_2,..., a_k to A
  repeat
    Randomly choose an arc a from $A^c \setminus A$
    if a is adjacent to another arc in A **then** add a to A
  until $|A| = D \times |A^c|$
  Check whether A results in a connected graph
until A results in a connected graph
end

It turns out that not many iterations are needed to achieve a connected graph; most of the time this can be achieved on the first attempt. Note that if k =1 then this method is the basic GRIDGROW method. The advantage of the GRIDGROW-*k*-SEEDS methods is that they can result in more interesting, non-homogeneous, graph structures; as noted in Chapter 3, the graph tends to be more dense around the seeds.

6.2.2 Reward distribution

In the preliminary investigation of Chapter 3, rewards were randomly assigned to arcs, uniformly chosen from some range U(a, b). The arcs were assigned randomly to a *reward class* based on specified proportions; each reward class has a different range from which the rewards are generated. However the rewards were spread randomly about the graph.

In this chapter we introduce a new method that allows some regions of the graph to have higher rewards than others by again utilizing the concept of seeds. Let $R \subseteq A$ be the set of arcs that have had their reward assigned. Randomly choose k arcs from A to be the reward seed arcs. Seed reward values are assigned to these arcs then they are added to R; for example, if there are three seed arcs these might have initial rewards of 5, 10 and 20. An unassigned arc a is randomly chosen from the adjacency of R, such that $a \in A \backslash R$ is adjacent to an arc $w \in R$. The adjacent arc w from R is called the **reward-parent** of a. If a is adjacent to more than one arc in R then one of these arcs is randomly chosen to be the reward-parent. The reward of a is then chosen to be within a certain range either side of the reward of w, with an adjustment to prevent rewards from going negative:

$$\max(\delta, r_w) - \delta \leq r_a \leq \max(\delta, r_w) + \delta$$

where δ is the maximum deviation between an arc and its reward-parent. This process repeats until all arcs are assigned. Depending on the value of δ, this method tends to create regions that have rewards in similar ranges, and the randomness allows variation, resulting in a rich variety of reward distributions for different graphs.

6.2.3 Depot location

The depot node is where the route must start and end. The location of the depot node can have a potentially large effect on the type of solutions that are possible, especially for a low budget. For the experiments in this chapter we assign the depot to a random node. Other possibilities include, but are not limited to the following:

- **Least central** – so that the maximum shortest path between any two nodes is maximised.
- **Most central** – so that the maximum shortest path between any two nodes is minimised.
- **Least rich** – for each node a "richness" measure is calculated. This richness measure is found by, for each arc, dividing the reward by the shortest distance to a node of that arc, and then adding for all arcs. The node with the lowest richness measure is selected.
- **Most rich** – the same as for least rich, except that the node with the highest richness measure is selected.

6.2.4 Problem instance generation settings

For the computational experiments in this chapter a set of 1440 problem instances was generated by generating 30 instances for each combination of the following generation settings:

- **4** × graph generation and reward assignment methods:
    - GRIDDESELECT, rewards assigned randomly from (5, 15)
    - GRIDGROW-1-SEED, rewards assigned randomly from (5, 15)
    - GRIDGROW-3-SEEDS, rewards assigned randomly from (5, 15)
    - GRIDGROW-3-SEEDS, rewards assigned by seed: 3 seeds (5,8,10), with $\delta = 3$.
- **3** × graph density levels: (25%, 50%, 75% of the complete grid)
- **4** × budget levels: (50%, 75%, 100%, 125% of total cost)
- **1** × depot setting method: random

These settings were chosen to provide a good mix of problem instances.

6.2.5 Problem characteristics

Problem characteristics are metrics that can be calculated for a given problem instance. These are calculated independently of the generation settings; two graphs generated in quite different ways could nevertheless have similar characteristics, and conversely two graphs generated with the same settings could have quite different characteristics. We extend the metrics introduced in Chapter 3 and define the following set of problem characteristics, many of which are specific to graphs based on grid graphs. There are characteristics for arc distribution, reward distribution and depot location.

- GRID_ROWS. The number of rows of vertices down the graph. Although we are using a 15×15 grid as a template, the actual graph might not extend all the way across, and so the *effective* height of the grid may be less than 15 (if the resulting graph was drawn on a piece of paper and the number of rows counted, this would be the "effective" height, which may be less than the underlying complete grid template).
- GRID_COLUMNS. The number of columns of vertices across the graph. Although we are using a 15×15 grid as a template, the actual graph might not extend all the way across, and so the *effective* width of the grid may be less than 15.
- GRID_ARCS. The number of arcs in the complete grid corresponding to the *effective* grid defined by GRID_ROWS and GRID_COLUMNS, rather than the 15×15 template grid.
- NODES. The number of nodes in the graph. Although the complete grid has $15^2 = 225$ nodes, the actual number in the graph is limited to only those that are incident on an included arc.
- ARCS. The number of arcs included in the graph.
- DENSITY. This measure is of the density of the graph based on the *effective* grid described above: ARCS / GRID_ARCS. The DENSITY may be greater than the density used in the graph generation method.
- BUDGET. The cost budget for the problem instance. In our case this corresponds to the maximum number of arc traversals, since arcs have unit cost.
- AVG_NODE_DEGREE. The mean degree of included nodes, where the degree of a node is the number of incident arcs, which may be 1, 2, 3 or 4.
- DEPOT_DEGREE. The degree of the depot node.
- DEPOT_ROWS_FROM_SIDE. Since the graph is laid out on a grid, every node may be thought of as having a "distance" to each side, not based on the path to a "side" node, but the number of grid rows or columns. This metric is the minimum of the four distances.
- PENDANT_ARCS. The number of arcs that are incident on a node of degree one. These are arcs that are only connected to the rest of the graph at one end.
- STRAIGHTLINE_ARCS. The number of arcs that are connected to only two other arcs, one at either end. Note that "straightline" arcs are not necessarily laid out in a straight lines on the grid.
- AVG_ARC_ADJACENCY. Each arc has an *adjacency*, the number of other arcs it is adjacent to. This metric is the mean of these across all arcs.
- AVG_SHORTEST_PATH. Between every pair of nodes the shortest path may be calculated (for example by using the Floyd-Warshall algorithm, described by Evans and Minieka [87]). This

metric is the mean of these shortest paths, and gives a measure of how well-connected the graph is.

- AVG_DEPOT_SHORTEST_PATH. The mean of the shortest paths between the depot node and every other node. Provides a measure of how centrally the depot node is located.
- BUDGET_TO_COST_RATIO. The ratio of the budget to the total cost of the graph, which in our case is the same as the number of arcs since they have unit cost: BUDGET / ARCS.
- TOTAL_REWARD. The sum of the rewards of all arcs.
- MAX_REWARD. The maximum reward of any arc.
- MIN_REWARD. The minimum reward of any arc.
- AVG_REWARD. The mean of the rewards of all arcs.
- STD_DEV_REWARD. The standard deviation of the rewards of all arcs.
- AVG_NODE_INCIDENCE_REWARD. The incidence reward for a node is the sum of the rewards of all incident arcs. This metric is the mean of these across all nodes.

Note that many of these characteristics will be strongly correlated, since they are designed to capture slightly different aspects of the same underlying structural features. This is not a problem, as the modelling procedures take that potential correlation into account.

6.3 MLS metaheuristics

We define a number of Modular Local Search heuristics. Our goal in this chapter is not to develop particularly new or effective heuristics, but to demonstrate the basic mechanism and usage of MLS, so the heuristics used are quite simple. We define variations of four common metaheuristic families: Steepest Ascent, Simulated Annealing, Tabu Search, and Variable Neighbourhood Search. We propose a number of variations of each of these, varying some of the key parameters. Detailed introductions to these heuristics are given in Chapter 5; the descriptions below focus on the specific MLS implementations of these that are used for experimentation in this chapter.

We consider 45 MLS instances in total. The motivation for these counts is given in Sections 6.3.4-6.3.8; it depends on the number and type of the parameters being varied:

- **1** × Richest Neighbour
- **4** × Steepest Ascent
- **25** × Simulated Annealing
- **6** × Tabu Search
- **9** × Variable Neighbourhood Search

In Section 6.5 we then combine the modules that make up these heuristics into hybrids of various types.

6.3.1 Construction heuristic

The same construction heuristic is used to generate the initial solution for all the MLS metaheuristics. This allows us to remove one degree of freedom; although the construction heuristic is very important to the final solution quality, it isn't part of the generic MLS framework, and much research has already

been performed that investigates construction heuristics; we are more interested in the metaheuristic strategies that come after this.

The construction heuristic used is the RICHEST NEIGHBOUR heuristic from Chapter 3 (Algorithm 3.3), with a lookahead of 1. In the computational experiments that follow we also present the results from this heuristic, to be used as the base level of performance from which the metaheuristics improve.

6.3.2 Move types

A *solution* for the ASRP is a route through a number of arcs, represented as an ordered sequence of nodes, beginning and ending at the depot node. The *objective function* is the total reward of the arcs included in the route.

For the purposes of this experiment we utilize two sets of the move-types defined in Chapter 3: the *basic* move-types ADD, DROP, SHORTCUT and DETOUR, and the *extended* move-types NADD, NDROP, NSHORTCUT and NDETOUR.

6.3.3 MLS defaults

In the descriptions of the metaheuristics that follow an **MLS configuration** is a template that defines which modules and parameters are included in the heuristic. An **MLS instance** is a specific instance of that configuration, with the parameter values set. In the configuration, if there are multiple values that a parameter may take, these are represented in curly braces, e.g. {1, 2, 3}.

The configurations only list deviations from the *defaults*, rather than listing every module explicitly each time. For the purposes of this chapter's experimentation the defaults are listed below:

- **Move-list size**: 5000
- **Move selection order**: random
- **Neighbourhood reduction process**: none
- **Admissibility condition**: FEASIBLE (Algorithm 6.3)
- **Fitness function**: OBJECTIVE (Algorithm 6.5)
- **Examinations maximum**: unlimited
- **Examination order**: random
- **Generate-initial-solution**: RICHEST NEIGHBOUR(1) (Algorithm 3.3)
- **Initialize-memory**: none
- **Update-memory**: UPDATE BEST-SO-FAR (OBJECTIVE) (Algorithm 6.6)
- **Change-current-solution**: none
- **Memory structures**: best-so-far solution (objective)

The move-list size was set to 5000 after an early round of experimentation that caused some of the heuristics to take prohibitively long to execute.

In particular, note that for these experiments we do not allow infeasible solutions, the FEASIBLE admissibility condition (Algorithm 6.3) always applies. Recall that multiple admissibility conditions may be active at any time, and they must *all* be satisfied for the solution to be admissible.

Specific modules are defined for each heuristic, but the following modules are common to many of the heuristics, or are in principle generic. Recall the notation conventions that s represents a solution (usually the current solution), s' represents a trial neighbour solution, s'' represents the target solution, i.e. the best admissible neighbour, s^* is the best-so-far solution, $f(s)$ is the fitness function value of s, $z(s)$ is the objective function value of s, $c(s)$ is the cost of s, A is the set of admissible neighbours and B is the cost budget.

Modules that are specific to the ASRP have the scope explicitly defined; all other modules are generic and may be applied to any problem domain.

6.3.3.1 Admissibility condition modules

Algorithm 6.3 presents an admissibility condition that is domain-specific to the ASRP. This admissibility condition is satisfied if the solution is feasible, which in the context of the ASRP means that the total cost does not exceed the budget.

Algorithm 6.3 MLS admissibility condition FEASIBLE (ASRP)
Scope: ASRP problems
Input: s', B *// The trial solution and the cost budget*

Calculate the cost of the trial solution, $c(s')$
if $c(s') \leq B$ **then**
return *admissible*
else
return *inadmissible*
end
end

Algorithm 6.4 gives the improving admissibility condition, which is satisfied if the trial solution has a strictly better fitness function value than the current solution.

Algorithm 6.4 MLS admissibility condition IMPROVING
Input: $f(s)$, s' *// The fitness of the current solution and the trial solution*

Calculate the fitness of the trial solution, $f(s')$
if $f(s') > f(s)$ **then**
return *admissible*
else
return *inadmissible*
end
end

6.3.3.2 Fitness function modules

Algorithm 6.5 gives the OBJECTIVE fitness function, which is the default unless specifically over-ridden.

Algorithm 6.5 MLS fitness function OBJECTIVE
Input: *s* *// The solution being evaluated*

return $z(s)$
end

6.3.3.3 Update-memory modules

Algorithm 6.6 gives the default update-memory function, which is to update the best-so-far solution if the target solution is better.

Algorithm 6.6 MLS update-memory UPDATE BEST-SO-FAR (OBJECTIVE)
Input: *s″, s** *// The target solution and the BSF solution*

if $z(s'') > z(s^*)$ **then**
$s^* \leftarrow s''$ *// Update the best-so-far*
end
end

6.3.3.4 Trigger modules

Algorithm 6.7 gives the local optimum trigger, which is tripped if there were no admissible neighbours found in the previous iteration of the search scheme. Note that this is only checking for an *apparent* local optimum; there may have been other admissible solutions that were technically in the neighbourhood of the current solution, but were not examined due to the search logic.

Algorithm 6.7 MLS trigger LOCAL OPTIMUM
Input: *A* *// The set of admissible candidates*

if $A = \varnothing$ **then**
return *tripped*
else
return *not tripped*
end
end

Algorithm 6.8 checks whether a certain number of iterations have passed since a specified trigger has been tripped. Every trigger has an automatic counter memory parameter that is incremented every iteration and is reset to zero when tripped. This trigger takes a parameter which is the trigger whose counter is being checked; this can be, but is not necessarily, this trigger itself. If this trigger is included in the configuration it has a requirement that the threshold memory parameter also be set.

Algorithm 6.8 MLS trigger ITERATIONS SINCE LAST TRIGGER (*trig*)
Prerequisite: A memory parameter must be defined for the iteration threshold.
Input: t_{trig}, θ *// The number of iterations since trigger "trig" was last tripped and the iteration threshold for this trigger*

if $t_{trig} \geq \theta$ **then**
return *tripped*
else
return *not tripped*
end
end

Algorithm 6.9 checks whether the total number of iterations since the metaheuristic started has reached some threshold. Usually used to terminate the heuristic, or in more complicated scenarios to enter a new phase.

Algorithm 6.9 MLS trigger TOTAL ITERATIONS
Prerequisite: A memory parameter must be defined for the iteration threshold.
Input: t, θ *// The number of iterations and the iteration threshold for this trigger*

if $t \geq \theta$ **then**
return *tripped*
else
return *not tripped*
end
end

Algorithm 6.10 checks whether the specified trigger has been tripped a certain number of times. Every trigger has an automatic counter memory parameter that is incremented each time the trigger is tripped. This trigger takes a parameter which is the trigger whose counter is being checked; this can be, but is not necessarily, this trigger itself. If this trigger is included in the configuration it has a requirement that the threshold memory parameter also be set.

Algorithm 6.10 MLS trigger TRIGGER TRIP COUNT (*trig*)
Prerequisite: A memory parameter must be defined for the trip count threshold.
Input: K_{trig}, θ *// The number of times that trigger "trig" has been tripped and the trip count threshold for this trigger*

if $K_{trig} \geq \theta$ **then**
return *tripped*
else
return *not tripped*
end
end

6.3.3.5 Response modules

All MLS configurations must include Algorithm 6.11, which is the termination response. This response can belong to a variety of different triggers.

Algorithm 6.11 MLS response TERMINATE
Terminate the heuristic, returning the best solution found so far
end

Algorithm 6.12 is the response to deactivate a particular trigger, which is specified as an input parameter. An inactive trigger is not checked at the end of the search scheme iteration until it is reactivated.

Algorithm 6.12 MLS response DEACTIVATE TRIGGER (*trig*)
Set trigger *trig* to *inactive*
end

Algorithm 6.13 is the response to activate a particular trigger, which is specified as an input parameter.

Algorithm 6.13 MLS response ACTIVATE TRIGGER (*trig*)
Set trigger *trig* to *active*
end

Algorithm 6.14 is the response to deactivate a particular admissibility condition, which is specified as an input parameter.

Algorithm 6.14 MLS response DEACTIVATE ADMISSIBILITY CONDITION (*c*)
Set admissibility condition *c* to *inactive*
end

Algorithm 6.15 is the response to activate a particular admissibility condition, which is specified as an input parameter.

Algorithm 6.15 MLS response ACTIVATE ADMISSIBILITY CONDITION (*c*)
Set admissibility condition *c* to *active*
end

6.3.4 Richest Neighbour

This metaheuristic is a trivial container for the solution obtained by the construction heuristic. It simply immediately terminates, returning the constructed solution. Algorithm 6.16 gives the MLS configuration. Since the move-list size is set to zero, no moves are selected, no solutions are examined, and therefore an apparent local optimum has been reached so the procedure terminates.

The configuration is called RICHEST NEIGHBOUR since this is the solution obtained by the construction heuristic RICHEST NEIGHBOUR(1). In the experimentation this instance is called "RN1".

Algorithm 6.16 MLS configuration RICHEST NEIGHBOUR
Generate-initial-solution: RICHEST NEIGHBOUR(1) (Algorithm 3.3)
Move-list size: 0
Triggers and responses:
Trigger: LOCAL OPTIMUM (Algorithm 6.7) - *active*
Response: TERMINATE (Algorithm 6.11)
end

6.3.5 Steepest Ascent

Steepest Ascent is one of the simplest local search heuristics; it examines all of the solutions in the neighbourhood and selects the best improving solution, iterating until there are no improving solutions in the neighbourhood, at which time the current (best) solution is returned.

Steepest Ascent is expressed with the configuration in Algorithm 6.17.

Algorithm 6.17 MLS configuration STEEPEST ASCENT
Move-types: {*basic*, *extended*}
Admissibility conditions:
IMPROVING (Algorithm 6.4)
FEASIBLE (Algorithm 5.1)
Candidate list size: *unlimited*
Triggers and responses:
Trigger: LOCAL OPTIMUM (Algorithm 6.7) - *active*
Response: TERMINATE (Algorithm 6.11)
Memory parameters:
Lookahead: {4, 8, 12}
end

There are no additional modules required for Steepest Ascent, other than those already defined.

6.3.5.1 Steepest Ascent instances for experimentation

Four instances of Steepest Ascent are considered, as shown in Table 6.1:

Table 6.1: Configuration settings for the Steepest Ascent MLS instances

Name	Move-type set	Lookahead
StpAscBasic	basic	1
StpAscExt4	extended	4
StpAscExt8	extended	8
StpAscExt12	extended	12

6.3.6 Simulated Annealing

Simulated Annealing is one of the oldest metaheuristics. The first admissible candidate examined is chosen as the target, improving solutions are always accepted, and non-improving solutions are accepted with a certain probability, which decreases as the heuristic progresses.

Simulated Annealing is expressed with the configuration in Algorithm 6.18.

Algorithm 6.18 MLS configuration SIMULATED ANNEALING
Move-types: *basic*
Admissibility conditions:
ANNEALING PROBABILITY (Algorithm 6.19)
FEASIBLE (Algorithm 6.3)
Candidate list size: 1
Triggers and responses:
Trigger-1: ITERATIONS SINCE LAST TRIGGER (*trigger-1*) (Algorithm 6.8) - *active*
Response: REDUCE ANNEALING TEMPERATURE (Algorithm 6.20)
Trigger-2: TEMPERATURE THRESHOLD (Algorithm 6.21) – *active*
Response: TERMINATE (Algorithm 6.11)
Memory parameters:
Annealing temperature: {100, 250, 500, 1000, 1500}
Cooling rate: {0.99, 0.95, 0.80, 0.60, 0.35}
Iterations before temperature reduction: {5, 10, 20, 50, 80}
Temperature threshold: {0.001, 0.05, 0.2, 1.0, 10}
end

Note that the memory parameter *iterations before temperature reduction* is the threshold used in *Trigger-1*.

Simulated Annealing also has a number of specialist modules. The main module is the ANNEALING PROBABILITY admissibility condition defined in Algorithm 6.19. This module has the prerequisite that the *annealing temperature* and *cooling rate* memory parameters are defined.

Algorithm 6.19 MLS admissibility condition ANNEALING PROBABILITY

Prerequisites: Memory parameters must be defined for the annealing temperature and the cooling rate.

Input: $f(s)$, s', T *// The fitness of the current solution, the trial solution and the annealing temperature*

Calculate the fitness of the trial solution, $f(s')$

$\delta \leftarrow f(s') - f(s)$

if $\delta > 0$ **then**

return *admissible*

else if `Uniform`$(0,1) < e^{-\delta / T}$ **then**

return *admissible*

else

return *inadmissible*

end

end

Algorithm 6.20 is a specialist Simulated Annealing response, it reduces the annealing temperature. This response represents the "annealing" concept in the Simulated Annealing heuristic; it makes non-improving solutions less likely to be accepted as admissible. This response is technically an **update-memory** module, and the response is to execute the update-memory module, but they are computationally equivalent.

Algorithm 6.20 MLS response REDUCE ANNEALING TEMPERATURE

Input: T, a *// The annealing temperature and the cooling rate*

$T \leftarrow a.T$

end

Algorithm 6.21 describes the specialist Simulated Annealing trigger that is used to detect when the heuristic should be terminated. When the annealing temperature has been reduced below a certain threshold this trigger is tripped.

Algorithm 6.21 MLS trigger TEMPERATURE THRESHOLD

Prerequisite: The memory parameters for the annealing temperature and the minimum temperature threshold must be defined

Input: T, T_{min} *// The current annealing temperature and the temperature threshold*

if $T \leq T_{min}$ **then**
  return *tripped*
else
  return *not tripped*
end
end

6.3.6.1 Simulated Annealing instances for experimentation

The version of the Simulated Annealing metaheuristic used here has four parameters, each with five values:

- Annealing temperature: {100, 250, 500, 1000, 1500}
- Cooling rate: {0.99, 0.95, 0.80, 0.60, 0.35}
- Iterations before temperature reduction: {5, 10, 20, 50, 80}
- Temperature threshold: {0.001, 0.05, 0.2, 1.0, 10}

A full factorial design would call for $5^4 = 625$ different Simulated Annealing heuristics. Instead, a Taguchi experimental design for four factors, each at five levels, was used. The Taguchi design allowed an even spread of the parameter values to be chosen, while keeping the number of variations to a minimum (see Wu and Hamada [251] for a description of Taguchi experimental designs). The Simulated Annealing instances are listed in Table 6.2.

Table 6.2: Configuration settings for the Simulated Annealing MLS instances

Name	Temp	Rate	Iterations	Threshold
SA1	1500	0.99	80	0.001
SA2	1500	0.95	50	0.05
SA3	1500	0.80	20	0.2
SA4	1500	0.60	10	1
SA5	1500	0.35	5	10
SA6	1000	0.99	50	0.2
SA7	1000	0.95	20	1
SA8	1000	0.80	10	10
SA9	1000	0.60	5	0.001
SA10	1000	0.35	80	0.05
SA11	500	0.99	20	10
SA12	500	0.95	10	0.001
SA13	500	0.80	5	0.05
SA14	500	0.60	80	0.2
SA15	500	0.35	50	1
SA16	250	0.99	10	0.05
SA17	250	0.95	5	0.2
SA18	250	0.80	80	1
SA19	250	0.60	50	10
SA20	250	0.35	20	0.001
SA21	100	0.99	5	1
SA22	100	0.95	80	10
SA23	100	0.80	50	0.001
SA24	100	0.60	20	0.05
SA25	100	0.35	10	0.2

6.3.7 Tabu Search

Tabu Search selects the best neighbouring solution from the neighbourhood whether it is improving or not. This allows it to escape from local optima, but can cause cycling. It mitigates this by maintaining a *tabu list* of solution elements that may not be changed for a certain number of iterations, the *tabu tenure*. In the case of the ASRP we make arcs tabu when they are added or removed from the route; they may not be added or removed from the route again until the tabu tenure has elapsed. The exception is when the neighbouring solution meets the *aspiration criterion* that it is (strictly) better than the best-so-far solution.

Tabu Search is expressed with the configuration given in Algorithm 6.22.

Algorithm 6.22 MLS configuration TABU SEARCH
Move-types: *basic*
Admissibility conditions:
TABU ARCS WITH ASPIRATION (Algorithm 6.23)
FEASIBLE (Algorithm 6.3)
Candidate list size: *unlimited*
Memory update: UPDATE TABU ARCS (Algorithm 6.24)
Triggers and responses:
Trigger: TOTAL ITERATIONS (Algorithm 6.9) - *active*
Response: TERMINATE (Algorithm 6.11)
Memory parameters:
Tabu tenure: {3, 5, 7, 10, 15, 20}
Iteration threshold for termination: 1000
end

Tabu Search has a number of specialist modules. The main module is the admissibility condition that checks whether the neighbouring solution involves changing any tabu arcs, as defined in Algorithm 6.23. For notational convenience we treat a solution s as a set of arcs.

Algorithm 6.23 MLS admissibility condition TABU ARCS WITH ASPIRATION
Scope: ASRP problems
Prerequisites: The tabu tenure memory parameter, and the tabu list memory element must be defined
Input: s^*, s, s', T *// The best-so-far solution, the current solution, the trial solution and the tabu list*

$\Delta \leftarrow \{s \cup s'\} \setminus \{ s \cap s'\}$ *// Find all the arcs that have changed from s to s'*
if $\Delta \cap T = \varnothing$ **then** *// No tabu arcs are changed*
return *admissible*
else if $f(s') > f(s^*)$ **then** *// Tabu arcs are changed but the aspiration criterion is met*
return *admissible*
else
return *inadmissible*
end
end

After each iteration the tabu list needs to be updated; any arcs that have changed from the current solution to the target solution need to be added, and their remaining tenure set to the tabu tenure memory parameter, arcs already on the list need to have their remaining tenure decremented, and then arcs that have zero remaining tenure must be removed from the list. In MLS the tabu list is actually a list of *tabu items*. A **tabu item** is a memory element that is a container for an *arc* and an *integer*

representing the remaining tabu tenure. For convenience, we refer to arcs being on the tabu list, and to setting their tenure; programmatically this is managed with the tabu items. Algorithm 6.24 describes the procedure by which the tabu list is updated.

Algorithm 6.24 MLS update-memory UPDATE TABU ARCS

Scope: ASRP problems

Prerequisites: The tabu tenure memory parameter, and the tabu list memory element must be defined. The tabu arcs with aspiration admissibility condition must be active.

Input: s, s'', T, τ, r_a *// The current solution, the best solution, the tabu list, the tabu tenure, and the remaining tenure for all arcs* $a \in T$

$\Delta \leftarrow \{s \cup s''\} \setminus \{ s \cap s''\}$ *// Find all the arcs that have changed between from s and s"*

$r_a \leftarrow r_a - 1, \forall\ a \in T$ *// Decrement the tenure of all the arcs currently on the list*

$T \leftarrow T \setminus \{a: r_a = 0\}$ *// Remove any arcs with no remaining tenure from the list*

$r_a \leftarrow \tau, \forall\ a \in \Delta$ *// Set the tenure of any changed arcs to the tabu tenure parameter*

$T \leftarrow T \cup \Delta$ *// Add the changed arcs to the tabu list*

end

6.3.7.1 Tabu Search instances for experimentation

Six instances of Tabu Search are considered, each the same except for the tabu tenure. These are listed in Table 6.1.

Table 6.3: Configuration settings for the Tabu Search MLS instances

Name	Tabu tenure
Tabu3	3
Tabu5	5
Tabu7	7
Tabu10	10
Tabu15	15
Tabu20	20

6.3.8 Variable Neighbourhood Search

Variable Neighbourhood Search proceeds like Steepest Ascent until a local optimum is reached, at which point the set of move-types is changed so that the current solution may no longer be a local optimum with respect to the new search topology and the search may continue. Our variation of this metaheuristic uses the change of neighbourhood structure as a temporary diversification phase; the set of move-types is changed for a certain number of iterations and then changed back. For the ASRP implementation we utilize two sets of move-types: the *basic* move-types (ADD, DROP, SHORTCUT and DETOUR) and the *extended* move-types (NADD, NDROP, NSHORTCUT and NDETOUR).

Variable Neighbourhood Search is expressed with the configuration in Algorithm 6.25.

Algorithm 6.25 MLS configuration VARIABLE NEIGHBOURHOOD SEARCH

Move-types: *basic*, *extended*

Admissibility conditions:

IMPROVING (Algorithm 6.4)

FEASIBLE (Algorithm 6.3)

Candidate list size: *unlimited*

Triggers and responses:

Trigger-1: LOCAL OPTIMUM (Algorithm 6.7) - *active*

Response: SWITCH TO EXTENDED MOVE-TYPES (Algorithm 6.27)

Response: DEACTIVATE TRIGGER (*trigger-1*) (Algorithm 6.12)

Response: ACTIVATE TRIGGER (*trigger-2*) (Algorithm 6.13)

Trigger-2: ITERATIONS SINCE LAST TRIGGER (*trigger-2*) (Algorithm 6.8) - *inactive*

Response: SWITCH TO BASIC MOVE-TYPES (Algorithm 6.26)

Response: DEACTIVATE TRIGGER (*trigger-2*) (Algorithm 6.12)

Response: ACTIVATE TRIGGER (*trigger-1*) (Algorithm 6.13)

Trigger-3: TOTAL ITERATIONS (Algorithm 6.9) - *active*

Response: TERMINATE (Algorithm 6.11)

Memory parameters:

Lookahead: {4, 8, 12}

Iterations in diversification phase: {1, 3, 5}

end

Note that the memory parameter *iterations in diversification phase* is the threshold used in *Trigger-2*.

This implementation of Variable Neighbourhood Search requires an additional two response modules, Algorithm 6.26, which swaps the move-types from the *extended* set to the *basic* set, and Algorithm 6.27 that performs the opposite operation.

Algorithm 6.26 MLS response SWITCH TO BASIC MOVE-TYPES

// Activates the basic move-types and deactivates the extended move-types.

// The basic set consists of {ADD, DROP, SHORTCUT, DETOUR};

// the extended set consists of {nADD, nDROP, nSHORTCUT, nDETOUR}.

Scope: ASRP problems

Set the *extended* move-types *inactive*

Set the *basic* move-types *active*

end

Algorithm 6.27 MLS response SWITCH TO EXTENDED MOVE-TYPES

// Activates the extended move-types and deactivates the basic move-types.

// The basic set consists of {ADD, DROP, SHORTCUT, DETOUR};

// the extended set consists of {NADD, NDROP, NSHORTCUT, NDETOUR}.

Scope: ASRP problems

Set the *basic* move-types *inactive*

Set the *extended* move-types *active*

end

6.3.8.1 Variable Neighbourhood Search instances for experimentation

Nine instances of Variable Neighbourhood Search are considered, crossing three levels for the two parameters, as shown in Table 6.4.

Table 6.4: Configuration settings for the Variable Neighbourhood Search MLS instances

Name	Lookahead	Diversification iterations
VNS4_1	4	1
VNS4_3	4	3
VNS4_5	4	5
VNS8_1	8	1
VNS8_3	8	3
VNS8_5	8	5
VNS12_1	12	1
VNS12_3	12	3
VNS12_5	12	5

6.4 Experimentation and analysis

The experimentation phase was conducted as a number of tournaments. The number of instances (1440) and heuristics (45) precluded running every heuristic on every problem instance, although this was the original intention. In total, 43,765 experiments were performed, requiring a combined duration of 4,569 computer hours, or 27 computer weeks. In actuality the duration was less than this because the experiments were spread across a number of machines, however it still required several months of computation. It was decided that this subset of experiments constituted a large enough population to enable meaningful analysis.

Unfortunately, because the experiments were run on a number of different computers, direct comparison of the exact running times is not possible, however all of the machines were approximately similar in terms of processor speed and RAM, so some qualitative analysis can be performed; all machines were Intel Core2-Duo processors running Windows XP, with 4GB of RAM. In particular, all of the heuristics of each type were performed on the same machine (e.g. all the Tabu Search heuristics were performed on the same machine), so these may be compared with each other more directly. Note

that the problem instances were randomized before being assigned to the heuristics, in order to get a good mix of problem types.

We begin the analysis with a general comparison of the performance of the heuristics, and then analyze each heuristic family in more detail, including some attempts to model the relationship between problem characteristics and heuristic performance, attacking the problem in several ways.

6.4.1 Predicting relative performance of heuristics

When solving real world combinatorial optimization problems, one of the questions that an Operations Research practitioner must answer is that of which solution methodology to use. If the problem is of such a size or complexity that exact algorithms are impractical then a metaheuristic must be chosen. Many of the "case studies" in the literature seem to choose a particular heuristic with no justification. From a certain perspective this is understandable; practitioners do not always have the time to compare multiple methods, and it is not always necessary to find the "best" method; sometimes *any* heuristic will give solutions that are "good enough".

However a methodology for systematically determining which types of heuristics are suited for which problems would be a useful addition to the field, especially combined with a standardized way to describe metaheuristics. We do not attempt to develop such a methodology, but we do provide some validation of the feasibility of such an approach.

Our goal, in the analysis of the computational results of this chapter, is to determine whether it is possible to train a predictive model such that on a separate test set of problem instances it is able to predict which of two heuristics will perform the best.

We attempt several methods, however the general idea is to get a set of problem instances where on approximately half of the instances heuristic A outperforms heuristic B, and on half the instances heuristic B outperforms heuristic A. This balanced dataset prevents the model simply using the prior probabilities to pick one heuristic over the other. We then divide the dataset into training and test subsets. The predictive model is trained on the training set, and then the model is applied to the test set to determine the classification accuracy. The only variables available to the predictive model are the problem characteristics.

Our success criterion for these experiments is whether the model has a better accuracy than we would expect from a random selection. This both validates the approach, and is a potentially valuable result in itself. Even a small increase in performance over a random selection might cause valuable insights in the "real world", where often the same problem must be resolved regularly with different data.

For the classification models we deliberately use a range of standard modelling techniques from the statistical software packages SAS and SPSS Clementine. The techniques include neural networks, logistic regression, and decision trees. All models were created using the default settings in the modelling software.

6.4.2 General comparison

There were 246 problem instances that were solved by all 45 heuristics, for a total of 11,070 experiments. Figure 6.1 shows the sum of all reward collected for these instances, by heuristic.

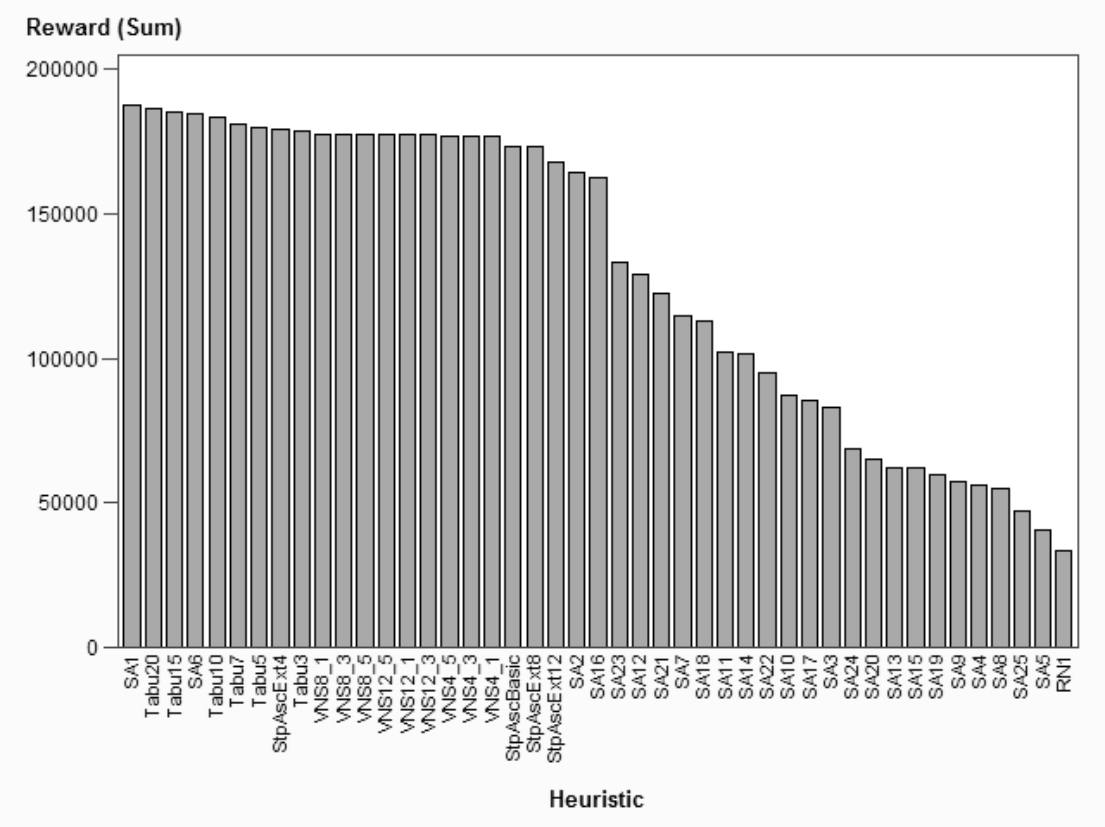


Figure 6.1: Sum of reward collected by heuristic for 246 instance overlap set

Results from one problem instance are not directly comparable with results from another instance, since the amount of reward collected will be directly proportional to a number of factors in addition to the performance of the heuristic: the number of arcs, the cost budget, and the amount of reward on the graph. We define a new measure, which we call the **score**, *Z*:

Z = reward collected / (total reward on graph * budget to cost ratio)

The denominator of this equation is a crude proxy for how much reward we might "expect" to be collected, all other things being equal. Figure 6.2 reproduces the above graph for the mean score values, also giving 95% confidence bands for the mean, which are calculated automatically by SAS.

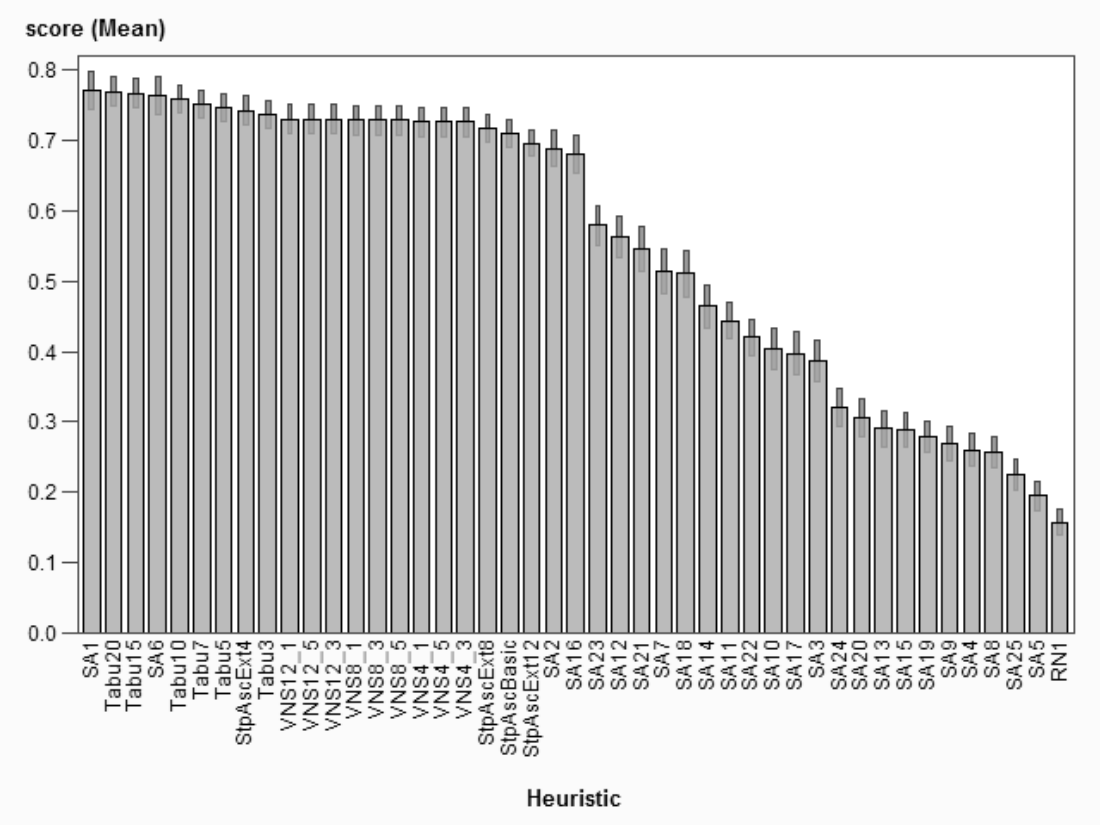


Figure 6.2: Score means for Steepest Ascent heuristics with 95% confidence bars

We are able to make some general observations about these results:

- The derived score metric appears to result in the same rankings and relative shape as the raw reward.
- Several of the Simulated Annealing heuristics performed very well (including the best), but a large number of Simulated Annealing heuristics performed very poorly.
- All of the Tabu Search heuristics performed well, and the higher the tabu tenure the better the performance. Overall, the Tabu Search family of heuristics performed the most consistently well.
- All of the Variable Neighbourhood Search heuristics performed very similarly.
- One of the Steepest Ascent heuristics performed quite well, the others not as well.

We restrict consideration to the "competitive" heuristics, making a cut-off point after SA16, due to the step change in performance (excluding all but four of the SA heuristics and the RN heuristic), and consider the average running time of each heuristic family. The VNS and SA heuristics had running times on the order of 20 minutes, compared to 2 minutes for the TS heuristics and 20 seconds for the StpAsc heuristics.

6.4.3 Analysis of Steepest Ascent results

The four Steepest Ascent heuristics were run on all 1440 problem instances. Recall that the four heuristics are the same, except for the move-types that are allowed. Three heuristics had the extended ASRP move-types, with different lookahead parameters (4, 8 and 12), and one heuristic had the basic

move-types, which are equivalent to the extended move-types with lookahead = 1. So any differences are due to the effect of the different lookahead parameter.

6.4.3.1 General performance of Steepest Ascent heuristics

The first consideration is the overall performance of the heuristics. Figure 6.3 shows the total reward collected, and Figure 6.4 shows the mean score, for all 1440 problem instances. It seems clear that, in aggregate, increasing the lookahead parameter *worsens* the performance of Steepest Ascent, with two clear levels of performance. This seems counter-intuitive, but can possibly be explained by reference to the default limit of 5000 on the number of moves examined. The higher the lookahead, the less of the neighbourhood that can be examined.

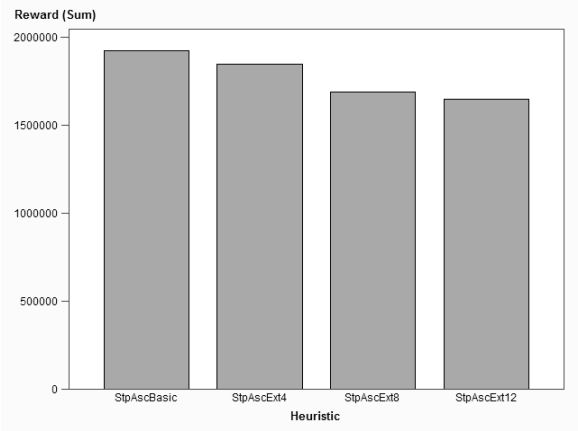


Figure 6.3: Total reward for Steepest Ascent

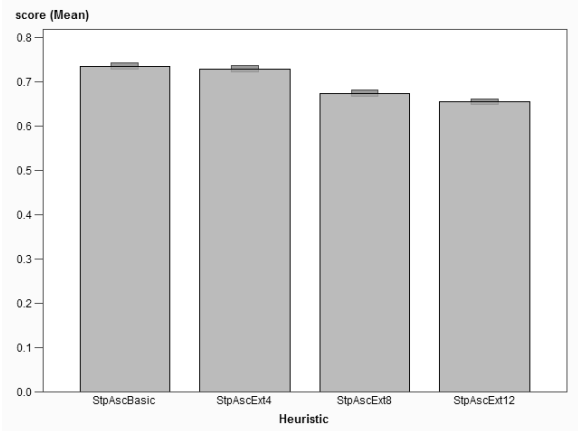


Figure 6.4: Mean score for Steepest Ascent

We next check whether the order of heuristic performance is a strict dominance relation, or whether it varies depending on the problem instance. For each instance, the heuristics are ranked 1-4 and then the proportion of the 1440 instances that the heuristic has each rank is calculated. In the event of a tie, both heuristics receive the better rank.

Table 6.5: Proportion of problem instances at each rank for Steepest Ascent

Heuristic	Rank			
	1	2	3	4
StpAscBasic	**52%**	24%	11%	13%
StpAscExt4	37%	**51%**	10%	3%
StpAscExt8	8%	16%	**48%**	28%
StpAscExt12	4%	9%	31%	**55%**

Table 6.5 shows that there is no strict dominance of the heuristics. Note that the same rank proportions were obtained for *reward* and *score*. Each of the heuristics is best, and worst, on some instances. However, there is a clear relationship between a low lookahead parameter and higher performance. From our perspective, the interesting aspect of this is to determine if there is a *systematic* reason relationship between heuristic performance and the characteristics of the problem instance.

6.4.3.2 Modelling relative heuristic performance of Steepest Ascent

Several initial attempts to use decision trees and neural networks to predict relative heuristic performance failed. The models had a predictive accuracy barely above random. Multiple variations were explored, including using the problem characteristics and the rank to classify the observation as the correct heuristic, and using the problem characteristics and the heuristic to predict the rank.

Although it appears that we cannot easily predict relative heuristic performance for four heuristics, it is interesting to consider whether this can be achieved in a restricted sense.

Table 6.6: Combinations of ranks for Steepest Ascent

StpAscBasic	StpAscExt4	StpAscExt8	StpAscExt12	freq	Percent
1	2	3	4	401	27.8%
1	2	4	3	247	17.2%
2	1	3	4	220	15.3%
2	1	4	3	85	5.9%
3	1	2	4	78	5.4%
4	1	2	3	67	4.7%
1	3	2	4	47	3.3%
4	1	3	2	42	2.9%
3	2	1	4	37	2.6%
1	3	4	2	28	1.9%
4	2	1	3	26	1.8%
3	1	4	2	25	1.7%
4	3	2	1	24	1.7%
4	3	1	2	17	1.2%
2	3	1	4	14	1.0%
4	2	3	1	11	0.8%
1	4	3	2	9	0.6%
2	3	4	1	8	0.6%
2	4	1	3	8	0.6%
1	4	2	3	7	0.5%
3	2	4	1	7	0.5%
3	4	1	2	6	0.4%
3	4	2	1	6	0.4%
2	4	3	1	3	0.2%
1	2	2	2	2	0.1%
2	1	3	3	2	0.1%
4	1	1	3	2	0.1%
4	1	2	2	2	0.1%
1	1	4	3	1	0.1%
1	2	2	4	1	0.1%
2	3	3	1	1	0.1%
3	1	1	4	1	0.1%
3	1	4	1	1	0.1%
3	4	1	1	1	0.1%
4	1	1	1	1	0.1%
4	1	3	1	1	0.1%
4	2	1	2	1	0.1%

We now attempt to determine whether we can learn to predict which heuristic will perform better, based purely on problem characteristics. Table 6.6 gives all the combinations of ranks in the 1440 problem instances executed by the Steepest Ascent heuristics. So, for example, the top row indicates that there are 401 problem instances where the heuristics performed in the following order of performance: StpAscBasic, StpAscExt4, StpAscExt8, StpAscExt12. In total there are 449 instances where

StpAscBasic performs best and StpAscExt12 performs worst (case A), and 37 instances where StpAscExt12 performs best and StpAscBasic performs worst (case B), where **best** means rank=1 and **worst** means rank=4. Our method is to use these extreme cases to train a prediction algorithm, and then apply this algorithm to the remainder of the instances (where the difference in performance is not so extreme) to test if it can predict relative performance. The objective here is to determine whether predicting relative heuristic performance is possible, rather than to analyze the nature of these particular Steepest Ascent heuristics, hence the choice of only the these two cases.

We include all 37 instances for case B and randomly select 37 instances from case A to obtain a set of 74 instances that is balanced such that there are the same number of instances where each heuristic outperforms the other. A balanced training set ensures that the prediction algorithm does not simply base its classifications on the relative proportions of the cases.

A neural network was built using half of the input dataset for training and half for testing. The neural network was built using the default settings of the statistical modelling software package SPSS Clementine, which automatically trains a multilayer perceptron (artificial neural network) to the given data. On the test dataset it achieved a classification accuracy of 88.6%, which is extremely good compared to a random classification. One of the outputs of the neural net procedure is the relative importance of the input variables; these importance values are given in Table 6.7.

Table 6.7: Relative importance of input characteristics to neural net

Characteristic	Importance
Budget_to_cost_ratio	0.4396
Nodes	0.2362
Avg_shortest_path	0.2322
Pendant_arcs	0.2191
Avg_node_degree	0.2179
Budget	0.1829
Avg_arc_adjacency	0.1812
Mean_node_incidence_reward	0.1218
Straightline_arcs	0.1188
Depot_degree	0.1115
Min_reward	0.1008
Mean_reward	0.0886
Avg_depot_shortest_path	0.0716
Grid_arcs	0.0706
Arcs	0.0680
Grid_cols	0.0469
Total_cost	0.0444
Depot_rows_from_side	0.0261
Density	0.0254
Grid_rows	0.0207
Std_dev_reward	0.0196
Max_reward	0.0090
Total_reward	0.0023

The next step is to test the heuristic on the remainder of the instances. Recall that 74 problem instances were used in the training of the neural network algorithm. That leaves 1366 instances that have had no input to the neural network. These instances form our test set.

Table 6.8: Classification results on the test set

	Classified A	Classified B
Case A	530	209
	71.7%	28.3%
Case B	50	165
	23.3%	76.7%

Table 6.8 presents the results of the test; the results are extremely positive. The percentages displayed are the proportion of the instances for each case that are classified in each bucket. For example, of the 739 problem instances that were case A, 530 were correctly classified as case A (71.7%), and 209 were classified incorrectly as case B (28.3%). The overall classification accuracy of the prediction algorithm is 72.9%.

6.4.3.3 Summary of Steepest Ascent results

There is a clear inverse relationship between look-ahead period and heuristic performance, when considered in aggregate over the whole set of problem instances. However, this is not a strict relationship; there are a number of instances where each of the heuristics performs better than the others. An interesting question is whether this is a random or a systematic effect.

A trial experiment was conducted to determine whether a neural network could predict which of two heuristics, StpAscBasic and StpAscExt12, would perform best on a sample of problem instances that the neural network had not been trained on. The neural network had a classification accuracy of 72.9% on the test set, which clearly proves our hypothesis that it is possible to predict relative heuristic performance based solely on an analysis of the problem characteristics.

6.4.4 Analysis of Simulated Annealing results

The 25 Simulated Annealing heuristics were run on a total of 546 common instances. Recall that the heuristics had four varying parameters, each with five values. Instead of the full factorial design of 625 heuristics, an experimental design of 25 was chosen, as listed in Table 6.2.

6.4.4.1 General performance of Simulated Annealing heuristics

Figure 6.5 gives the total reward collected by each heuristic, and Figure 6.6 gives the mean score, with 95% confidence limits. Recall that the **score** is defined as:

reward collected / (total reward on graph * budget to cost ratio).

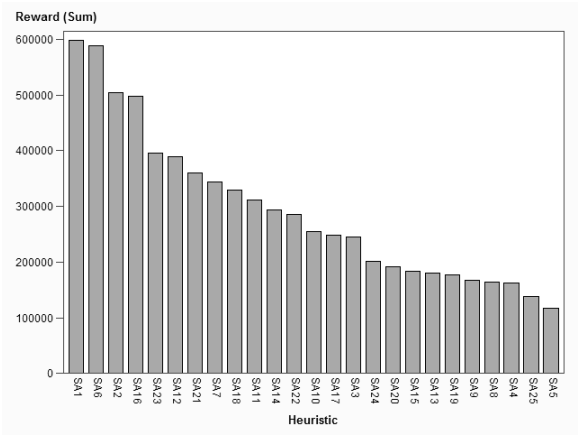


Figure 6.5: Total reward for Simulated Annealing

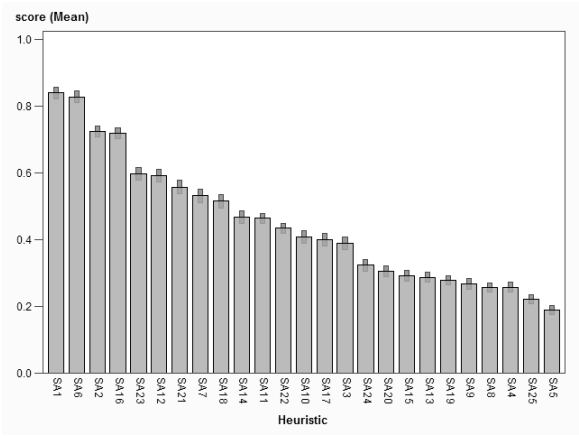


Figure 6.6: Mean score for Simulated Annealing

Note that the top four Simulated Annealing heuristics perform better than the others, with the top two a clear step above the second two.

It is interesting to consider how well each heuristic performs in relation to the others. For each problem instance we rank the heuristics based on their performance, and then count the number of times they achieved each rank. Table 6.9 displays the counts of these ranks, ordered by decreasing total reward, and heat-mapped to indicate the magnitude of that frequency (dark cells correspond to higher values).

Table 6.9: Distribution of ranks for Simulated Annealing heuristics ordered by total reward

	Rank																								
Heuristic	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**Total Reward**
SA1	384	144	7	1										2		1	1	3	2		1					598,064
SA6	152	359	23	3			1									1	2	1	2					1	1	588,424
SA2	6	10	230	210	47	19	6	6	2		3			2		1	1			2		1				504,672
SA16	5	17	228	203	52	16	9	3	2		1			3			1		3	1	1			1		497,777
SA23	1	3	17	26	147	131	80	47	36	15	16	9	2	1	2		1	2	3	2		1			4	395,484
SA12		2	8	28	120	135	94	48	39	27	20	7	2	3	2		2	1	3	2		1		1	1	389,617
SA21		6	15	36	68	79	107	82	63	43	22	8	5	3	2		1		1	1	2	2				359,921
SA7	1	2	8	19	48	45	70	106	96	66	32	20	9	7	5	1	2	2	3	2	1			1		343,728
SA18		2	5	12	28	53	74	88	79	79	61	29	13	9	2		2	2	3	1	1	1		1	1	328,618
SA11			1	2	21	27	38	38	57	66	57	52	45	41	28	23	14	15	5	8	3	3	1	1		312,159
SA14			2	3	5	20	30	53	64	75	90	81	54	28	19	9	4	5	1	2				1		293,417
SA22					2	12	13	30	43	62	70	88	42	54	36	28	30	9	11	4	3	1	2	5	1	284,794
SA10					2	1	10	14	21	38	66	72	103	96	51	33	12	14	5	4	2	1		1		254,692
SA17			1	1	4	2	6	11	21	38	42	63	102	84	84	35	24	6	8	6	2	3		2	1	248,934
SA3				1		3	8	13	16	20	40	70	94	98	95	35	23	9	5	7	2	4	2	1		245,016
SA24					1		1	2	4	4	7	22	26	47	61	103	65	65	48	34	24	19	10	1	2	201,732
SA20									3	2	7	6	21	26	48	77	79	58	74	47	47	24	21	4	2	191,918
SA15											3	3	8	18	24	65	93	85	66	65	41	41	25	7	2	184,101
SA13						1	1		1	3	1	4	6	14	34	39	53	82	76	86	64	34	35	11	1	180,695
SA19							1	2		1	3	5	8	17	23	45	55	64	64	66	89	56	34	11	2	177,415
SA9				1					1	1	3	4	2	5	13	16	43	47	59	70	78	74	87	35	7	167,434
SA8									1	2	2	5	2	8	6	20	31	45	46	73	61	99	89	48	8	163,708
SA4								2	1		1	1	2	4	5	18	25	38	50	60	75	116	101	38	9	162,395
SA25					1	1				1	1	1	2	3	1	5	7	8	16	12	28	37	88	287	47	137,471
SA5												3		3	3	7	2	7	2	8	2	11	18	72	408	117,169

Note that the strong diagonal band in Table 6.9 indicates that there is a clear hierarchy of performance among the heuristics. Table 6.10 recalls the parameters for the best 4 heuristics.

Table 6.10: Parameters for top 4 Simulated Annealing heuristics

Parameter	SA1	SA6	SA2	SA16
Temp	1500	1000	1500	250
Rate	0.99	0.99	0.95	0.99
Iterations	80	50	50	10
Threshold	0.001	0.2	0.05	0.05

There is no single parameter value that all four of the best heuristics have in common. There are some commonalities, however. The first three have a high temperature, and high iterations between temperature reductions; they all have a high cooling rate, and they all have a relatively low threshold. These are conceptually the settings that correspond to a longer, slower annealing process. We also note that SA1, which had the highest score of all heuristics considered, has the longest annealing process. An oddity appears to be SA16, which is ranked in the top tier of heuristics, even though most of its parameters are low. This suggests that the cooling rate, which is the highest value of 0.99, dominates the other parameters in terms of defining performance.

We now consider each of the parameters separately, and find the mean score for the heuristics with each of the parameter values over the subset of 546 problem instances that all the SA heuristics were executed on. Recall that the score is the reward collected divided by the product of the total reward on the graph and the budget-to-cost ratio.

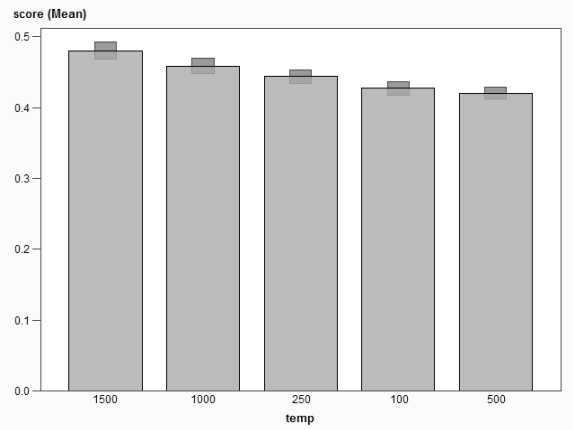


Figure 6.7: Mean score by "temp"

Figure 6.7 shows the performance of the Simulated Annealing heuristics, grouped by their value for the initial temperature. A higher initial temperature generally correlates to a higher mean score, except for where the initial temperature is 500, which leads to worse results than the other temperatures.

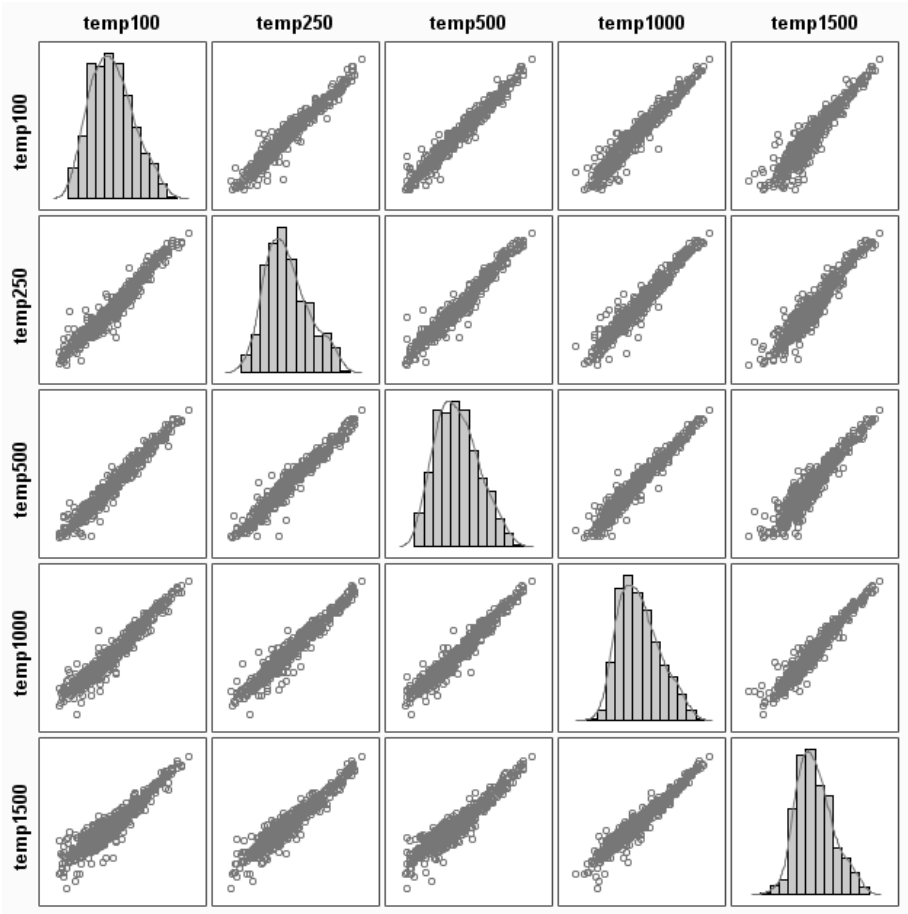


Figure 6.8: Mean score for each problem instance by "temp"

Figure 6.8 illustrates that there is not a major difference in mean score as a result of the temperature parameter. For each problem instance, the mean of the scores of the five heuristics with each temperature value were found, and plotted against each other in a scatter matrix. So, for example, a point on the top right scatter graph represents one problem instance, and the mean score of the heuristics where temp=100 is plotted against the mean score of the heuristics where temp=1500. The strong linear relationships imply that, generally, it is not possible to distinguish between the performances of the heuristics based on the value of the initial temperature parameter; a high score on a problem instance with one parameter value implies a high score with the other values. The slight relationship between aggregate performance and temperature seen in Figure 6.7 is not enough to distinguish between heuristics for a given problem instance.

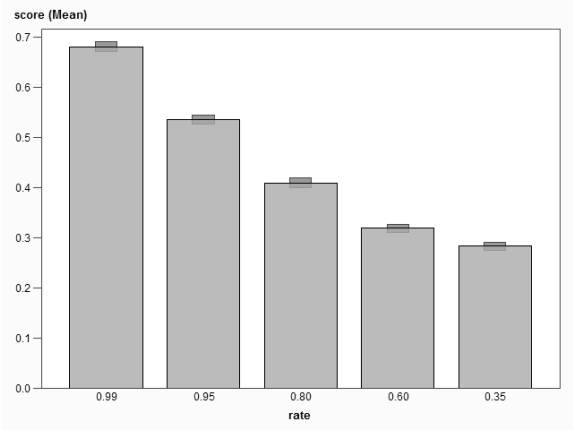


Figure 6.9: Mean score by "rate"

Figure 6.9 shows a much stronger relationship between cooling rate and performance; the slower the cooling rate, the better the performance of Simulated Annealing.

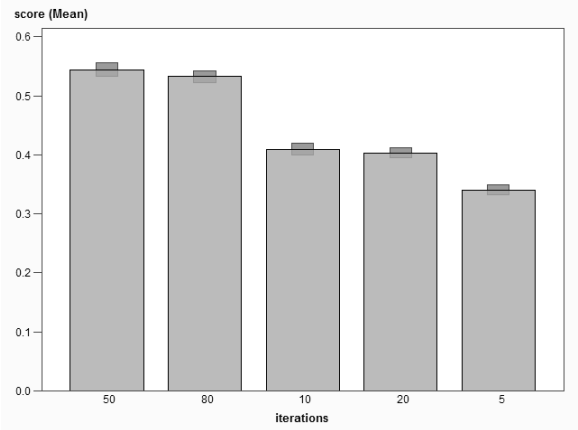


Figure 6.10: Mean score by "iterations"

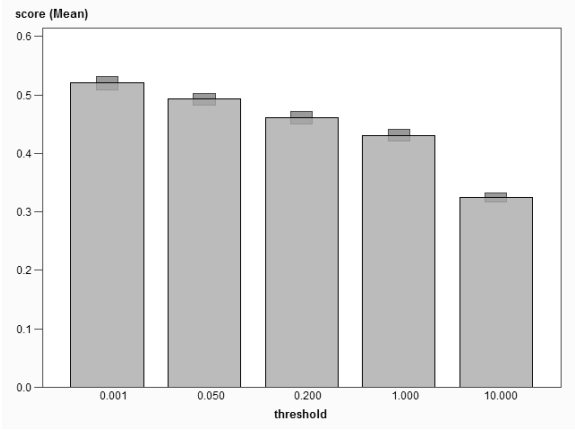


Figure 6.11: Mean score by "threshold"

Figure 6.10 shows that there is a performance difference based on the number of iterations between temperature reductions. Those heuristics with a high number of iterations, 50 and 80, perform the best, and it seems that above a certain point there may not be any gain in additional iterations. There is a clear step down to the heuristics with 10 and 20 iterations, where again there is not much difference. There is another clear step down to the heuristics with only 5 iterations. It seems that, generally, more iterations between temperature reductions results in better performance.

Figure 6.11 again shows that heuristics with lower thresholds have better performance. A lower threshold means that the heuristic proceeds for longer before terminating, so the better performance of these heuristics implies that many of the final improvements are in the last iterations, when the temperature is very low and the probability of accepting non-improving moves is very low.

6.4.4.2 Principle components analysis of the problem instance characteristics

A useful activity is to attempt to produce a smaller number of explanatory variables to describe the problem instance characteristics. Principle Components Analysis (PCA), developed by Pearson [209], transforms a set of (possibly) correlated variables into a new set of uncorrelated variables, such that the new variables are linear functions of the original variables. The first component explains as much of the variability in the data as possible, and the next component as much of the remaining variability, etc., so that a smaller number of principle components can explain a majority of the variability in the data. PCA is a standard method of variable dimensionality reduction in multivariate data analysis.

A PCA was performed on the 22 problem instance characteristics on the full set of 1440 problem instances. The PCA was performed using the default settings in SAS, which uses the correlations matrix and the degrees of freedom as the divisor for variance. Table 6.11 gives the proportion of the variability explained by each principle component.

Table 6.11: Proportion of problem characteristic variation explained by each principle component

Component	Eigenvalue	Proportion	Cumulative
1	7.4882	0.3404	0.3404
2	5.0011	0.2273	0.5677
3	3.0082	0.1367	0.7044
4	1.3541	0.0616	0.7660
5	1.3068	0.0594	0.8254
6	1.0184	0.0463	0.8717
7	0.7323	0.0333	0.9050
8	0.6636	0.0302	0.9351
9	0.4710	0.0214	0.9565
10	0.4212	0.0191	0.9757
11	0.1857	0.0084	0.9841
12	0.0946	0.0043	0.9884
13	0.0837	0.0038	0.9922
14	0.0759	0.0035	0.9957
15	0.0403	0.0018	0.9975
16	0.0385	0.0018	0.9993
17	0.0059	0.0003	0.9995
18	0.0044	0.0002	0.9997
19	0.0028	0.0001	0.9998
20	0.0019	0.0001	0.9999
21	0.0014	0.0001	1.0000
22	0.0001	0.0000	1.0000

It can be seen that the PCA was extremely successful. 87% of the variability in the data was captured by only six components. A varimax rotation was performed on the principle components. This is an operation that maximizes the sum of the variances of the squared loadings, which aids in their

interpretability. The coefficients of these six components under varimax rotation and standardization of inputs are given in Table 6.12.

Table 6.12: Coefficients of the first six principle components under varimax rotation

	Component					
Characteristic	**PC1**	**PC2**	**PC3**	**PC4**	**PC5**	**PC6**
Nodes	-0.9320	0.1680	-0.0549	0.1150	-0.0098	0.0589
Avg_arc_adjacency	0.8900	0.3970	0.0840	-0.0213	0.0262	-0.0881
Avg_depot_shortest_path	-0.8790	-0.0075	-0.0770	0.2250	-0.0066	0.0480
Mean_node_incidence_reward	-0.8660	0.2050	-0.0291	0.1130	-0.0025	-0.2460
Budget	0.8410	0.4900	0.0815	0.0754	0.0309	-0.0858
Budget_to_cost_ratio	0.7470	0.4560	-0.0185	0.4450	0.0259	-0.0885
Grid_arcs	-0.6550	0.1220	-0.0670	-0.2500	-0.0155	-0.0076
Mean_reward	0.5570	0.7500	0.0316	0.1300	0.0483	-0.1910
Straightline_arcs	0.4790	0.1120	0.0554	0.1710	0.0564	0.4540
Arcs	0.0124	0.9710	0.0309	0.0397	0.0297	-0.0258
Avg_node_degree	0.3290	0.9320	0.0662	0.0576	0.0328	-0.0198
Depot_rows_from_side	0.3190	0.9190	0.0234	0.1960	0.0309	-0.0209
Grid_rows	-0.3840	0.7970	0.0889	-0.0856	-0.0203	0.3380
Total_reward	-0.3440	0.7140	0.0471	-0.0858	-0.0178	0.2820
Avg_shortest_path	0.2350	0.7060	0.0476	0.0370	0.6430	-0.0067
Depot_degree	-0.3430	0.6760	0.1050	-0.0894	-0.0214	0.3330
Min_reward	0.0895	0.0837	0.9620	0.0034	0.0057	0.0175
Density	0.0919	0.1030	0.9490	-0.1530	0.0064	0.0151
Std_dev_reward	-0.0812	-0.0078	-0.7280	0.6310	-0.0015	-0.0628
Max_reward	-0.0289	0.0679	-0.2020	0.9310	-0.0026	-0.0326
Grid_cols	-0.0286	-0.0192	-0.0038	-0.0103	0.9910	0.0147
Pendant_arcs	-0.0289	0.1130	0.0158	-0.0924	-0.0050	0.8730

Interpretation of principle components is not always easy; sometimes it is obvious what they represent, other times it is not as straightforward. To assist interpretation, all coefficients less than 0.4 have been greyed out in Table 6.12, and then the variables have been sorted. Although it is not really necessary for our purposes to have a convenient label, a rough interpretation can be attempted:

1. A mix of several aspects that is not straightforward to interpret, but seems to reflect a budget-cost effect. High values of BUDGET, BUDGET_TO_COST_RATIO, and MEAN_REWARD suggest a connection to how much reward is possible to collect. High negative values of NODES, GRID_ARCS, AVG_DEPOT_SHORTEST_PATH, and MEAN_NODE_INCIDENCE_REWARD suggest a contrast with the size and connectedness of the graph. The greyed-out values with magnitude greater than 0.3 also seem related to these aspects.

2. This component seems to reflect a depot effect, and to capture aspects relating to the size and connectedness of the graph, and the amount of reward available. It seems that together PC1

and PC2 capture the size and shape of the graph, which accounts for 57% of the variability in the data.

3. A contrast between the MIN_REWARD and DENSITY vs. STD_DEV_REWARD. Capturing some of the subtlety in the reward distribution in combination with the density of the graph.

4. Captures the remainder of the variability concerning the reward distribution, with an influence of the BUDGET_TO_COST_RATIO, which is a rough proxy for the expected amount of reward that could be collected.

5. Captures some additional information concerning how reachable parts of the graph are.

6. The remaining information about the shape of the graph; the number of straight-line and pendant arcs

6.4.4.3 Modelling relative heuristic performance of Simulated Annealing

In this exercise we take a different approach than with the Steepest Ascent heuristics. Using the *score* as a basis for comparison, we again calculate the "rank" of each heuristic. We then use the graph characteristics in *combination* with the Simulated Annealing parameters to create a predictive model of the rank.

The predictive model chosen for this analysis was a CHAID decision tree, with which we attempted to predict the score for each heuristic using the problem instance characteristics and the Simulated Annealing parameters as inputs to the model. A CHAID decision tree differs from other common decision tree approaches in that it can generate non-binary trees; the default settings in the SPSS Clementine modelling package were used. CHAID is one of the oldest decision tree algorithms, introduced in 1980 by Kass [155]. The ordered predicted scores for each problem instance were then used to generate predicted ranks for each heuristic. The main predictive variables used in the decision tree are listed below:

- Cooling rate
- Avg depot shortest path
- Initial temperature
- Budget to cost ratio
- Iterations between temperature reductions
- Min reward
- Avg node degree
- Avg arc adjacency
- Mean node incidence reward

Note that the only non-predictive SA parameter is the threshold.

With 546 problem instances and 25 heuristics there are $546 \times 25 = 13{,}650$ rank estimates. If the model predicts no better than random then we would expect the proportion of successes to be $1/25 = 0.04$. The model under the null hypothesis is Binomial(13650, 0.04), with mean 546 and variance 524. The model

predictions resulted in a very high 1836 successful rank predictions out of 13,650 possible predictions (13%). Clearly, the predictive model is better than random.

The number of exact successes in our modeled prediction of ranks is a point of interest, however this is not the main criterion for success of the predictive model. Instead our objective is to achieve a high success rate in the pair-wise comparison of *relative* ranks. For example, if heuristics A and B actually have ranks of 4 and 13, respectively, on a given problem instance, and we predict ranks of 2 and 5, then this counts as a success; the model correctly predicted that A would outperform B on that problem instance.

The results of this analysis were even more positive. Consider that with 25 heuristics there are 300 pairs of heuristics. There are 546 problem instances, so for each pair we calculate the proportion of correct relative rank predictions out of 546. In total there were 546 × 300 = 163,800 possible successes, and the decision tree model achieved 147,155 (90%).

Figure 6.12 shows the distribution of the results for each heuristic pair, where each data point gives the proportion of problem instances correct (out of 546) for a heuristic pair. Note that *all* heuristic pairs were predicted correctly more than half the time.

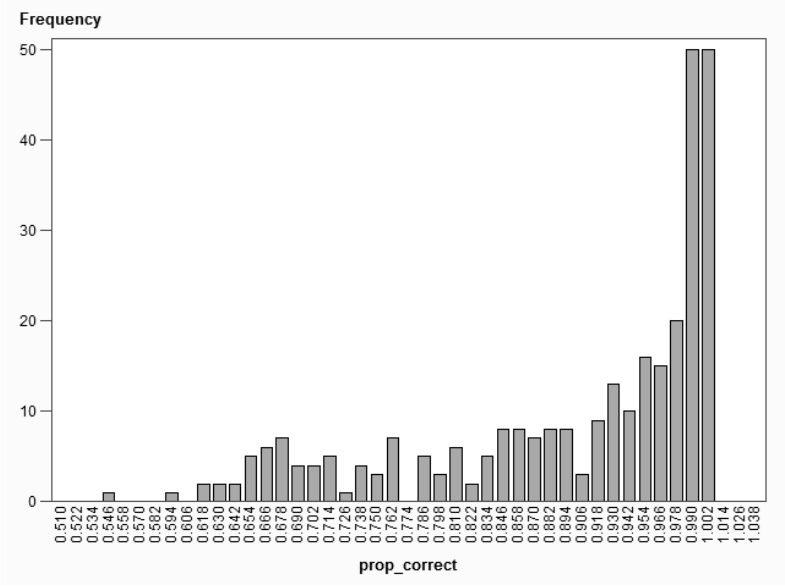


Figure 6.12: Proportion of problem instances correct for each heuristic pair

6.4.4.4 Summary of Simulated Annealing results

The 25 Simulated Annealing heuristics formed a relatively strict hierarchy of performance, and there was a clear relationship between certain parameter values and performance.

A predictive model was developed that was able to predict relative heuristic performance extremely well, using only problem instance characteristics and the Simulated Annealing parameters: initial temperature, cooling rate, iterations between temperature reductions and minimum temperature threshold. The decision tree achieved a much better predictive accuracy than a random selection. However, this result must be interpreted cautiously; since the relative performance of the heuristics is so

strictly hierarchical, most of the predictive accuracy may be attributed to the dominance of certain SA parameter values over others, rather than the interaction effect of the problem instance characteristics. In other words, it is possible to predict that heuristic A will perform better than heuristic B on problem instance *p*, but primarily because heuristic A *usually* performs better than B.

6.4.5 Analysis of Tabu Search results

The six Tabu Search heuristics were run on a total of 1050 common problem instances. Recall that the only difference between the heuristics was the *tabu tenure* parameter.

6.4.5.1 General performance of Tabu Search heuristics

Figure 6.13 gives the total reward collected by each heuristic, and Figure 6.14 gives the mean score, with 95% confidence limits.

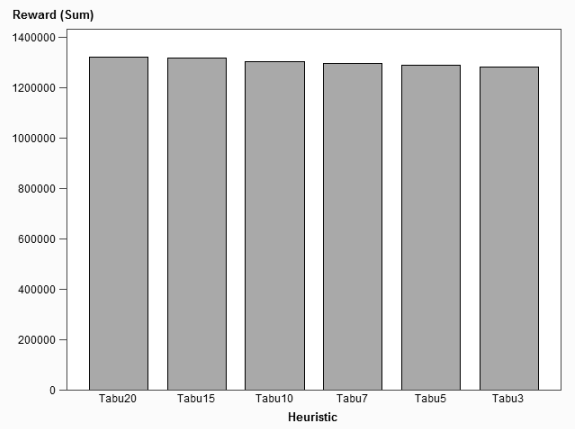


Figure 6.13: Total reward for Tabu Search

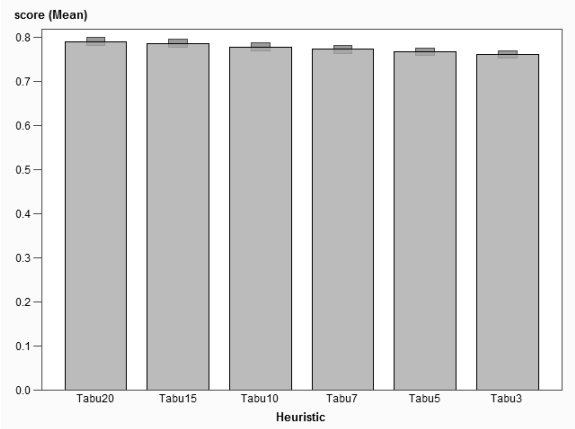


Figure 6.14: Mean score for Tabu Search

There is a slight direct and positive relationship between the tabu tenure and the overall performance, however there is not a large difference between any of the heuristics; they all perform similarly in aggregate. Note that since the best aggregate performance was achieved with the highest tabu tenure examined, it is possible that even higher tabu tenures would result in better performance still. Recall that the purpose of the experiments in this chapter is not to design the best possible heuristic, but simply to test the relationship between problem instance characteristics and heuristic performance. To examine the distribution of their relative performances, we rank each heuristic based on its performance on each problem instance, and count the number of problem instances on which it achieved that rank. The results are presented in Table 6.13. Note that these are presented as raw numbers rather than percentages as in Table 6.5 because the percentages were too small and the raw numbers are clearer.

Table 6.13: Distribution of ranks for Tabu Search heuristics ordered by total reward

Heuristic	Rank						Total reward
	1	2	3	4	5	6	
Tabu20	485	271	123	88	50	33	1321931
Tabu15	345	337	177	100	62	29	1314610
Tabu10	177	188	309	177	119	80	1301794
Tabu7	114	112	222	285	176	141	1293825
Tabu5	74	77	140	210	337	212	1287788
Tabu3	42	47	106	152	262	441	1281924

There seems to be a good distribution of ranks, which means that each heuristic is better than the others on some problem instances, and worse on others. In the next section we examine whether this is a systematic difference that is predictable based on an analysis of the problem characteristics.

6.4.5.2 Modelling relative heuristic performance of Tabu Search

For the Tabu Search heuristics, we wish to determine whether it is possible to predict for a given problem instance which of two Tabu Search heuristics will perform better.

We define **set A** to include the problem instances where Tabu3, Tabu5 and Tabu7 all performed better than Tabu10, Tabu15 and Tabu20; for these instances lower tabu tenures were preferable. We define **set B** to include the problem instances where Tabu3, Tabu5 and Tabu7 all performed *worse* than Tabu10, Tabu15 and Tabu20; for these instances higher tabu tenures were preferable.

Our approach, then, is to use an equal number of problem instances from sets A and B to train a predictive model, and then to apply this model to the remainder of the problem instances to test its accuracy. There were 39 problem instances in set A and 85 problem instances in set B, so our training set consists of 39 problem instances from each set: 78 instances in total. Several models were attempted and the best was chosen: a classification and regression decision tree using the A/B flag as the dependent variable and the problem instance characteristics as the predictors. A Classification and Regression Tree (CART) is a common non-parametric classification technique; the default settings within SPSS Clementine were used.

For the testing of the model, we perform pair-wise tests of the heuristics; each problem instance yields 15 pairs of heuristics.

A concern was removing the effect of the dominance of each heuristic over the others. For example, if heuristic H1 outperforms heuristic H2 on 75% of the problem instances in the test dataset, then a model could achieve 75% accuracy simply by always predicting H1. To eliminate this effect we consider each pair of heuristics separately, and randomly select a subset of the problem instances so that each heuristic outperforms the other on the same number of problem instances. This allows us to judge the predictive ability of the model fairly; given a random selection we would expect the model to be correct 50% of the time, so 50% is our baseline. For each pair of heuristics we randomly sample (without replacement) 50 problem instances where Heuristic 1 outperforms Heuristic 2, and 50 instances where the reverse is

true (the minimum number of instances we can choose from for any pair is 92). The proportion correct gives one data point for that heuristic pair, and we repeat the sampling process 500 times for each pair to find the distribution of the mean. According to the Central Limit Theorem, this distribution is normal.

Table 6.14: Results of pair-wise Tabu Search prediction

Heuristic 1	Heuristic 2	Steps	Mean	Std dev	Lower 95% CL for mean	Upper 95% CL for mean
Tabu3	Tabu20	5	61.62%	4.06%	61.26%	61.97%
Tabu3	Tabu15	4	59.78%	4.18%	59.42%	60.15%
Tabu5	Tabu20	4	59.14%	4.40%	58.75%	59.52%
Tabu5	Tabu15	3	58.49%	4.52%	58.09%	58.89%
Tabu3	Tabu10	3	56.00%	4.51%	55.61%	56.40%
Tabu7	Tabu15	2	55.94%	4.47%	55.55%	56.34%
Tabu7	Tabu20	3	55.56%	4.60%	55.16%	55.97%
Tabu5	Tabu10	2	55.00%	4.62%	54.60%	55.41%
Tabu7	Tabu10	1	53.96%	4.69%	53.55%	54.38%
Tabu10	Tabu20	2	52.98%	4.75%	52.57%	53.40%
Tabu5	Tabu7	1	52.56%	4.69%	52.15%	52.98%
Tabu3	Tabu7	2	52.22%	4.55%	51.82%	52.62%
Tabu10	Tabu15	1	52.20%	4.69%	51.79%	52.61%
Tabu3	Tabu5	1	50.67%	4.69%	50.26%	51.08%
Tabu15	Tabu20	1	50.00%	4.51%	49.61%	50.40%

Table 6.14 gives the results of the analysis, which are encouraging. For all pairs of heuristics except the last, the lower 95% confidence limit is above 50%, which means that we are 95% certain that our model performs better than a random selection. The confidence limits are calculated by the statistical software and represent the two-sided confidence limits for the mean with n-1 degrees of freedom. The column labeled "Steps" describes how many "heuristic steps" the pair are away from each other. Bearing in mind that the step sizes are not equal increments of the tabu tenure, we can see that generally, the more different the heuristics are, the better the model is able to predict which will outperform the other.

6.4.5.3 Summary of Tabu Search results

The Tabu Search heuristics all performed approximately equally, at least in aggregate, although there is a clear relation between a higher tabu tenure and better performance.

An experiment was conducted, in which a decision tree model was created using a carefully selected training dataset, and then this model was used to predict which heuristics would perform best on each problem instance, for all pairs of heuristics.

The results of this were positive; for all pairs of heuristics except one the model performed better than a random selection, with 95% confidence.

6.4.6 Analysis of Variable Neighbourhood Search results

The nine Variable Neighbourhood Search (VNS) heuristics were run on a total of 552 common problem instances. Recall that the VNS heuristics varied on two parameters: the number of diversification iterations in the diversification phase, and the maximum look-ahead distance for the extended move-types used in the diversification phase.

6.4.6.1 General performance of Variable Neighbourhood Search heuristics

Figure 6.15 gives the total reward collected by each heuristic, and Figure 6.16 gives the mean score, with 95% confidence limits.

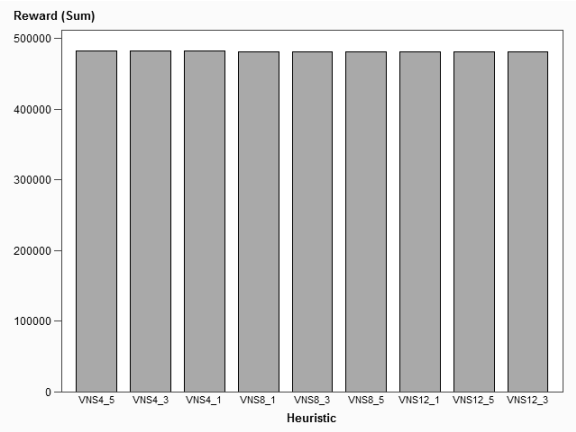


Figure 6.15: Total reward for VNS

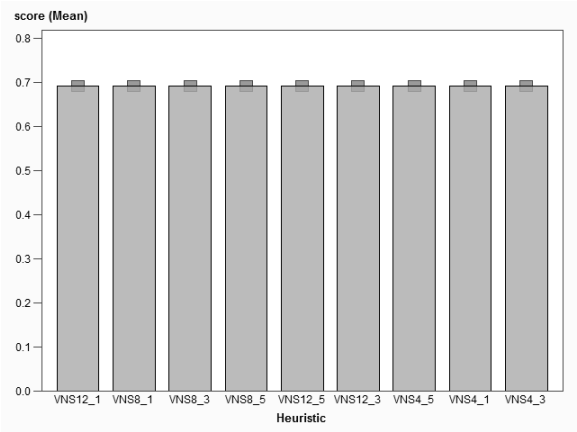


Figure 6.16: Mean score for VNS

As can be seen from the two graphs, there is very little difference between the performances of the heuristics. This is confirmed in Table 6.15, which shows the number of problem instances for each rank distribution, where the ranks can be tied (for example the second row shows that three heuristics were first equal, implying the same solution, and the remaining six heuristics shared another solution). This table shows the first seven rows, representing 75.91% of the problem instances; the full table contains an additional 92 rows with various combinations.

Table 6.15: Partial summary of rank distribution of VNS heuristics

Lookahead=4			Lookahead=8			Lookahead=12					
n=1	n=3	n=5	n=1	n=3	n=5	n=1	n=3	n=5	Freq	Pct	Cum Pct
1	1	1	1	1	1	1	1	1	286	51.81%	51.81%
1	1	1	4	4	4	4	4	4	60	10.87%	62.68%
7	7	7	1	1	1	1	1	1	35	6.34%	69.02%
4	4	4	4	4	4	1	1	1	14	2.54%	71.56%
1	1	1	1	1	1	7	7	7	9	1.63%	73.19%
4	4	4	7	7	7	1	1	1	8	1.45%	74.64%
1	1	1	4	4	4	7	7	7	7	1.27%	75.91%

Based on the results above, it does not seem that the VNS heuristics distinguish themselves from each other sufficiently to justify trying to model their relative performance.

6.4.7 Modelling relative heuristic performance of two different heuristics

The previous sections examine each of the "families" of heuristics separately. In this section we perform a brief analysis to determine if we can predict the relative heuristic performance of two heuristics from different families, based solely on problem characteristics, and using the techniques developed in the previous sections.

The heuristics were chosen based on the following criteria:

- They must both be relatively successful heuristics (in the top group from Figure 6.1),
- They must both be similar in aggregate performance, and
- They must both have a sufficient number of relative "wins"

The heuristics selected were SA6 and Tabu10. These heuristics were executed on a total of 516 common problem instances, and the distribution of "wins" for each heuristic are presented in Table 6.16.

Table 6.16: Frequency of ranks between SA6 and Tabu10

Heuristic	rank1	rank2
SA6	337	179
Tabu10	181	335

In order to keep the proportions balanced, 179 problem instances were randomly chosen on which SA6 outperformed Tabu10, and 179 instances were chosen where Tabu10 outperformed SA6. This set was further divided into balanced subsets of 89 problem instances for training the model, and 90 instances for testing.

A neural network model was constructed on the training dataset, and scored on the test dataset. This model was built using the default settings in the SPSS Clementine data modelling software. The results of this test are presented in Table 6.17.

Table 6.17: Classification results on the test set for TS and SA

	Predicted winner	
Actual winner	**SA6**	**Tabu10**
SA6	68	22
	75.6%	24.4%
Tabu10	18	72
	20.0%	80.0%

This experiment provides clear evidence that it is possible to predict relative heuristic performance, based on problem characteristics. The overall classification success rate is 77.78%.

6.5 Hybrids

One of the design imperatives and strengths of the Modular Local Search framework is the ease with which hybrid heuristics can be defined. If modules have been created for particular heuristics, then these can be combined in new ways simply by specifying the combination of modules, to create new heuristics with no new development effort.

The purpose of this section is to demonstrate this capability of the MLS framework, using the modules developed in the early part of the chapter. Note that our goal here is not to develop the best possible heuristic or hybrid, but only to demonstrate the procedure by which these are defined.

We develop five hybrid heuristics, based on combinations of the key modules for the heuristic "families" defined in Section 6.3. We develop one hybrid of Tabu Search and Simulated Annealing, and two each of Simulated Annealing with Variable Neighbourhood Search, and Tabu Search with Variable Neighbourhood Search.

We may draw a conceptual distinction between two types of hybrid: those which have multiple phases, each phase associated with the characteristics of one source heuristic; and hybrids which actually use the modules of multiple source heuristics within a single iteration of the search scheme. The hybrids we introduce that use Variable Neighbourhood Search are phased hybrids, and the hybrid of Tabu Search and Simulated Annealing is a mixed hybrid. These could also be considered *sequential* hybrids and *parallel* hybrids.

Some results are presented, and interesting features highlighted.

6.5.1 MLS configurations

Note that the settings and parameters for the following heuristics were chosen quite arbitrarily, in order to demonstrate the use of the hybridization process, rather than to create the most effective heuristic. Given the quite dramatic performance difference of the original heuristics with different parameter values, it seems reasonable to assume that the same situation would apply to the hybrids, and that combinations with "better" parameter values could exist.

6.5.1.1 Hybrid of Simulated Annealing and Tabu Search

Simulated Annealing and Tabu Search both modify the search primarily through influencing the admissibility of neighbouring solutions; in MLS they each have their own characteristic *admissibility condition* modules.

The MLS configuration for the hybrid of Simulated Annealing and Tabu Search is given in Algorithm 6.28. Note that this configuration has two admissibility conditions; recall that *both* of these admissibility conditions must be satisfied to make a solution admissible. This heuristic is primarily a version of Simulated Annealing, with the extra Tabu Search admissibility condition and the memory parameter and memory update module required to support this. The candidate list size is 1, as in Simulated Annealing, so the first admissible neighbour examined is accepted, although it would be an easy extension to increase this parameter.

Algorithm 6.28 MLS configuration HYBRID – SA & TS
- **Move-types:** *basic*
- **Admissibility conditions:**
  - ANNEALING PROBABILITY (Algorithm 6.19) – *active*
  - TABU ARCS WITH ASPIRATION (Algorithm 6.23) – *active*
  - FEASIBLE (Algorithm 6.3) – *active*
- **Candidate list size:** 1
- **Memory update:** UPDATE TABU ARCS (Algorithm 6.24)
- **Triggers and responses:**
  - **Trigger-1:** ITERATIONS SINCE LAST TRIGGER (*trigger-1*) (Algorithm 6.8) – *active*
    - **Response:** REDUCE ANNEALING TEMPERATURE (Algorithm 6.20)
  - **Trigger-2:** TEMPERATURE THRESHOLD (Algorithm 6.21) – *active*
    - **Response:** TERMINATE (Algorithm 6.11)
- **Memory parameters:**
  - Annealing temperature: 1000
  - Cooling rate: 0.95
  - Iterations before temperature reduction: 30
  - Temperature threshold: 0.001
  - Tabu tenure: 10

end

6.5.1.2 Hybrids of Simulated Annealing and Variable Neighbourhood Search

The usual usage of Variable Neighbourhood Search (VNS) is as a method of diversification when a local optimum is reached; the neighbourhood structure is changed so that the search can continue. The implementation we introduced in this chapter for the Arc Subset Routing Problem, in Section 6.3.8, uses the change of neighbourhood as a temporary diversification phase; the neighbourhood is changed to the extended move-types for a set number of moves, and then changed back.

In Simulated Annealing (SA), the search does not "get stuck" in a local optimum; solutions are examined until one passes the admissibility conditions and is then accepted, even if it worsens the objective function value. Technically, the point where the temperature is low enough that no non-improving solutions are accepted, and there are no improving solutions available, could be considered an apparent local optimum, however it is usually treated as the termination criterion for the heuristic.

In our hybrid of Simulated Annealing and Variable Neighbourhood Search, VNS is used as a distinct diversification phase after the SA heuristic has reached its usual termination point, and then the SA phase is repeated. This hybrid is a *multi-phase*, or *sequential*, hybrid. The MLS configuration, presented as Algorithm 6.29, is the most complicated we have yet seen, since it describes a quite sophisticated multi-phase procedure.

Algorithm 6.29 MLS configuration HYBRID – SA & VNS

Move-types: *basic*, *extended*

Admissibility conditions:

ANNEALING PROBABILITY (Algorithm 6.19) – *active*

IMPROVING (Algorithm 6.4) – *inactive*

FEASIBLE (Algorithm 6.3) – *active*

Candidate list size: 1

Triggers and responses:

Trigger-1: ITERATIONS SINCE LAST TRIGGER (*trigger-1*) (Algorithm 6.8) – *active*

Response: REDUCE ANNEALING TEMPERATURE (Algorithm 6.20)

Trigger-2: TEMPERATURE THRESHOLD (Algorithm 6.21) – *active*

Response: SWITCH TO EXTENDED MOVE-TYPES (Algorithm 6.27)

Response: SET CANDIDATE LIST SIZE (100000) (Algorithm 6.30)

Response: DEACTIVATE ADMISSIBILITY CONDITION (*ANNEALING PROBABILITY*) (Algorithm 6.14)

Response: ACTIVATE ADMISSIBILITY CONDITION (*IMPROVING*) (Algorithm 6.15)

Response: DEACTIVATE TRIGGER (*trigger-1*) (Algorithm 6.12)

Response: DEACTIVATE TRIGGER (*trigger-2*) (Algorithm 6.12)

Response: ACTIVATE TRIGGER (*trigger-3*) (Algorithm 6.13)

Trigger-3: ITERATIONS SINCE LAST TRIGGER (*trigger-3*) (Algorithm 6.8) – *inactive*

Response: SWITCH TO BASIC MOVE-TYPES (Algorithm 6.26)

Response: SET ANNEALING TEMPERATURE (1000) (Algorithm 6.31)

Response: SET CANDIDATE LIST SIZE (1) (Algorithm 6.30)

Response: DEACTIVATE ADMISSIBILITY CONDITION (*IMPROVING*) (Algorithm 6.14)

Response: ACTIVATE ADMISSIBILITY CONDITION (*ANNEALING PROBABILITY*) (Algorithm 6.15)

Response: DEACTIVATE TRIGGER (*trigger-3*) (Algorithm 6.12)

Response: ACTIVATE TRIGGER (*trigger-1*) (Algorithm 6.13)

Response: ACTIVATE TRIGGER (*trigger-2*) (Algorithm 6.13)

Trigger-4: TRIGGER TRIP COUNT (*trigger-3*) (Algorithm 6.10) – *active*

Response: TERMINATE (Algorithm 6.11)

Memory parameters:

Annealing temperature: 1000

Cooling rate: 0.95

Iterations before temperature reduction: 30

Temperature threshold: 0.001

Lookahead: 4

Iterations in diversification phase: {1, 3}

Trigger trip count threshold: 3

end

We utilize two variations of this hybrid, varying the number of iterations in the VNS diversification phase: one iteration (SA-VNS1) and three iterations (SA-VNS3).

Trigger-2 is the main trigger that switches from the SA phase to the VNS phase, and Trigger-3 is the trigger that switches back from the VNS diversification phase to the next SA phase. Trigger-3 is tripped three times before termination; there are three SA phases and three VNS phases, the heuristic terminates after the last VNS phase.

This hybrid heuristic is the first example of which we are aware in which one of the core search scheme modules, the admissibility conditions, being swapped in and out of activity. The ANNEALING PROBABILITY admissibility condition starts out active, and then the responses make this condition inactive and the IMPROVING admissibility condition active.

The candidate list size is given as 1 in the configuration, and this is what this parameter starts out as. However, during the diversification phase this is increased to 100,000 (effectively unlimited) and then decreased back to 1 again during the next Simulated Annealing phase. In this particular heuristic these two values are predetermined, but this is not necessarily so; this mechanism allows the parameter values to be set dynamically by the response modules, allowing the heuristic to "evolve" into a configuration that was not explicitly anticipated during its design. The candidate list size is modified using Algorithm 6.30, although there are many alternative forms that this could take, such as a non-parametric "swap candidate list size" module.

Algorithm 6.30 MLS response SET CANDIDATE LIST SIZE (*size*)
    Set the *candidate list size* to *size*
end

The *annealing temperature* memory parameter is modified during each of the Simulated Annealing phases; it is reduced until it is below the threshold. This means that at the conclusion of each of the VNS phases it needs to be reset to its initial value. In the interests of making this as simple as possible while still meeting the needs of the heuristic, we use Algorithm 6.31 to set the temperature to the amount specified, however an alternative method would be to have an additional memory parameter for the *initial* temperature, and then have a nonparametric response module that resets the temperature.

Algorithm 6.31 MLS response SET ANNEALING TEMPERATURE (*temp*)
    Input: *T* *// The annealing temperature*

    T ← *temp*
end

6.5.1.3 Hybrids of Tabu Search and Variable Neighbourhood Search

The hybrid presented here of Tabu Search (TS) and Variable Neighbourhood Search (VNS) is a sequential, or multi-phase, hybrid, similar to that described above for Simulated Annealing and VNS. In Tabu Search there is no clear point at which to start the diversification phase; there is no apparent local optimum reached, since non-improving solutions can be selected if they are the best available, and

there is no natural termination point such as the temperature reaching its minimum threshold in Simulated Annealing. Sophisticated detection mechanisms are possible to detect when the search has "stagnated" and would benefit from diversification, however, for the purposes of demonstrating the hybrids, a simple iteration count suffices.

Algorithm 6.32 MLS configuration HYBRID – TS & VNS
- **Move-types:** *basic*, *extended*
- **Admissibility conditions:**
  - TABU ARCS WITH ASPIRATION (Algorithm 6.23) – *active*
  - IMPROVING (Algorithm 6.4) – *inactive*
  - FEASIBLE (Algorithm 6.3) – *active*
- **Candidate list size:** 100,000
- **Triggers and responses:**
  - **Trigger-1:** ITERATIONS SINCE LAST TRIGGER (*trigger-1*) (Algorithm 6.8) – *active*
    - **Response:** SWITCH TO EXTENDED MOVE-TYPES (Algorithm 6.27)
    - **Response:** DEACTIVATE ADMISSIBILITY CONDITION (*TABU ARCS WITH ASPIRATION*) (Algorithm 6.14)
    - **Response:** ACTIVATE ADMISSIBILITY CONDITION (*IMPROVING*) (Algorithm 6.15)
    - **Response:** DEACTIVATE TRIGGER (*trigger-1*) (Algorithm 6.12)
    - **Response:** ACTIVATE TRIGGER (*trigger-2*) (Algorithm 6.13)
  - **Trigger-2:** ITERATIONS SINCE LAST TRIGGER (*trigger-3*) (Algorithm 6.8) – *inactive*
    - **Response:** SWITCH TO BASIC MOVE-TYPES (Algorithm 6.26)
    - **Response:** DEACTIVATE ADMISSIBILITY CONDITION (*IMPROVING*) (Algorithm 6.14)
    - **Response:** ACTIVATE ADMISSIBILITY CONDITION (*TABU ARCS WITH ASPIRATION*) (Algorithm 6.15)
    - **Response:** DEACTIVATE TRIGGER (*trigger-2*) (Algorithm 6.12)
    - **Response:** ACTIVATE TRIGGER (*trigger-1*) (Algorithm 6.13)
  - **Trigger-3:** TRIGGER TRIP COUNT (*trigger-2*) (Algorithm 6.10) – *active*
    - **Response:** TERMINATE (Algorithm 6.11)
- **Memory parameters:**
  - Tabu tenure: 10
  - Lookahead: 4
  - Iterations in diversification phase: {1, 3}
  - Trigger trip count threshold: 2

end

This hybrid has a similar multi-phase structure to the SA-VNS variation. It highlights a key feature of the MLS framework, that it is relatively easy to define extremely complex multi-phase heuristics that can completely change their structure in response to certain stimuli, due to the trigger-response model.

There are two variations used in experimentation; the number of iterations in the diversification phase is either one (TS-VNS1) or three (TS-VNS3). Note that the candidate list size is simply set to a large number, 100000, for both phases of the heuristic.

6.5.2 Results for hybrids

The hybrid heuristics were executed on all 1440 of the problem instances in the test set. However, what is interesting is how these hybrids compared with the "regular" heuristics, especially those from which their modules were chosen. Recall that there were 246 problem instances on which all of the regular heuristics were executed. Figure 6.17 presents the sum of the reward collected for each of these problem instances; this is a copy of Figure 6.1, but with the hybrids added and excluding those heuristics below SA16, which were clearly sub-standard and had a large drop-off on the graph.

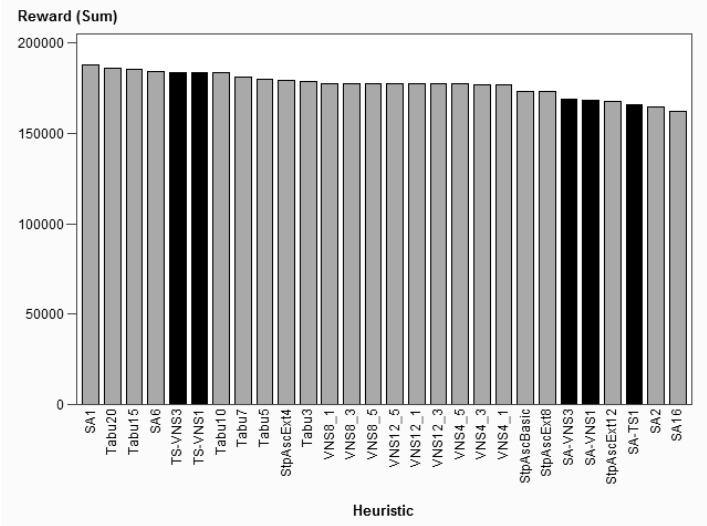


Figure 6.17: Sum of reward collected by heuristic for 246 instance overlap set, including hybrids

Note that none of the hybrids is the "best" heuristic examined during the experimentation, which is as expected. It is not a fair comparison to compare the Variable Neighbourhood Search heuristics with their hybrid counterparts, since in the hybrids we limited the number of VNS diversification phases to three and two respectively for the SA and TS variants, in order to limit the execution time. However, it is meaningful to compare the Tabu Search hybrids with their regular counterparts, and the same for the Simulated Annealing heuristics.

In particular, note that the three Simulated Annealing hybrids perform relatively poorly, at least in aggregate. Recall from the Simulated Annealing results in Section 6.4.4.1, and particular Figure 6.9, that the performance of the SA heuristics was strongly dependent on the cooling rate; heuristics with a cooling rate of 0.99 dominated those with a rate of 0.95. The SA hybrids utilized a cooling rate of 0.95, which is likely sub-optimal. It is worth noting that all three SA hybrids performed better than all the other regular SA heuristics, except for SA1 and SA6, which both had a cooling rate of 0.99. It seems that the low cooling rate for the hybrids is the controlling factor in determining their aggregate performance.

There is one main parameter in the Tabu Search heuristic, the tabu tenure, so the closest comparison between a heuristic and the hybrids that use it is between Tabu10 and the two TS-VNS hybrids (which have tabu tenure of 10). Note that the hybrids both outperform the regular heuristic, but only slightly.

Although Tabu10, TS-VNS1, and TS-VNS3 are close in aggregate reward for the 246 problem instances above, it is interesting to consider the distribution relative performances. There were 1051 problem instances that these three heuristics were performed on; Table 6.18 presents the distribution of relative performances, as ranks.

Table 6.18: Combinations of ranks for TS-VNS hybrids and Tabu10

TS-VNS1	TS-VNS3	Tabu10	Instances	Percent
1	2	3	149	14.2%
3	1	2	141	13.4%
2	1	3	136	12.9%
3	2	1	126	12.0%
1	3	2	118	11.2%
2	3	1	116	11.0%
1	1	1	78	7.4%
2	1	2	38	3.6%
1	1	3	37	3.5%
3	1	1	37	3.5%
1	2	2	27	2.6%
2	2	1	25	2.4%
1	3	1	23	2.2%

Tabu10 is the best heuristic in 405 problem instances (39%). As can be seen in Figure 6.18 and Figure 6.19, there is a generally linear relationship between the scores of the hybrids and that of Tabu10; on low-score problem instances this relationship is especially strong as on these instances the heuristics achieved the same score and the same solution.

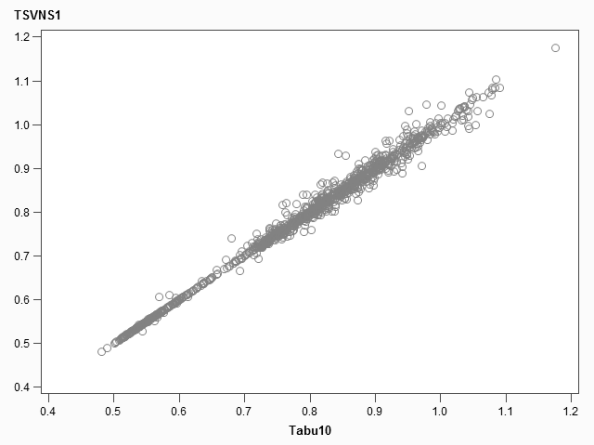


Figure 6.18: Scatter plot of scores: TS-VNS1 vs Tabu10

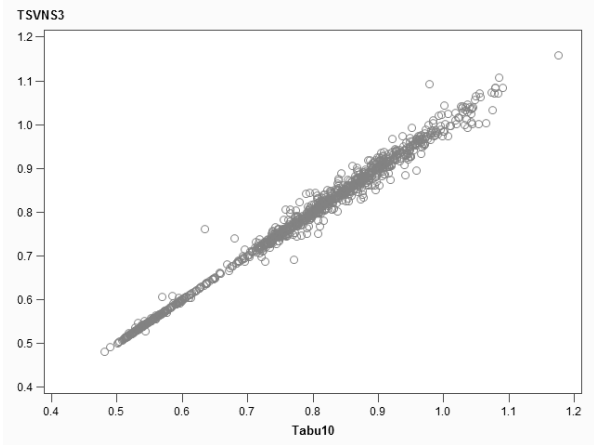


Figure 6.19: Scatter plot of scores: TS-VNS3 vs Tabu10

There is a slight, but clear, difference between the scores of the heuristics; each is better on some instances and worse on others. The next step is to determine whether this difference is systematic and predictable based on the problem instance characteristics.

We use the same procedure that we have used previously, comparing Tabu10 and TS-VNS3, and attempting to create a predictive model that predicts which heuristic will outperform the other, based on problem instance characteristics. Table 6.19 presents some summary information about these two heuristics. Each row describes the cases where a particular outcome occurs, where an outcome is defined in terms of one heuristic outperforming the other.

Table 6.19: Comparison of Tabu10 and TS-VNS3

Winner	Sum reward	Mean score	Instances	Pct wins
TS-VNS3	653514	0.800	501	47.7%
Tabu10	572420	0.818	408	38.8%
Draw	85435	0.621	142	13.5%

A number of predictive models were developed to attempt to distinguish between Tabu10 and TS-VNS3, based on the problem instance characteristics, and none of them were able to perform better than a random selection. This indicates that there is no *systematic* reason why Tabu10 performs better on some problem instances and worse on others; this difference is likely due to random variation, since these two heuristics are structurally very close to each other, and seem to perform similarly on similar problem instances.

Table 6.20: Comparison of Tabu10 and SA-TS

Winner	Sum Reward	Mean Score	Instances	Pct wins
Tabu10	1304783	0.779	978	93.1%
SA-TS1	1112098	0.680	73	6.9%

We briefly perform one more piece of analysis, comparing Tabu10 with the SA-TS hybrid. Table 6.20 summarizes the performance of these heuristics. Note that athough Tabu10 performs much better overall than SA-TS1, there are still some problem instances on which SA-TS1 performs better.

We construct a balanced modelling dataset by taking all 73 of the instances where SA-TS1 "wins", and 73 of the instances where Tabu10 wins, giving 146 problem instances. We divide this into a training set of 85 instances (60%) and a testing set of 61 instances (40%). An artificial neural network was applied, which achieved a 65.6% correct classification rate (correctly predicting which heuristic would achieve better performance).

6.5.3 Discussion of hybrids

The purpose of this section on hybrids was to demonstrate the ease with which hybrids can be developed with the MLS system. It requires no new programming to develop quite sophisticated multi-phase hybrids, or heuristics that combine the characteristics of multiple source heuristics; they are specified *declaratively* by listing the modules that are included and that are active at the first iteration. A description of MLSML, and examples of MLSML code can be found in Appendix C.

The fact that a heuristic is a hybrid does not, of course, automatically make it a good heuristic. The introduction of hybrids dramatically increases the size of the available "heuristic space", but large regions of that heuristic space are likely to be unsuited for any given problem instance. The heuristics demonstrated here were chosen somewhat arbitrarily, so we would not expect them to be more effective than the "regular" heuristics, which represent well developed heuristic paradigms. However, it is interesting to note that there *were*, in fact, particular problem instances on which each of the hybrids outperformed the regular heuristics.

6.6 Discussion

This chapter has presented the results of extensive computational experiments of Modular Local Search heuristics on the Arc Subset Routing Problem. There were several objectives for this chapter.

The first objective was to demonstrate the use of MLS to specify some common types of metaheuristics: Steepest Ascent, Simulated Annealing, Tabu Search and Variable Neighbourhood Search. These heuristics have quite different modes of operation, but share a similar core in that they are all local search-based approaches. Each requires several specialist MLS modules, but most of the structure of the heuristic is common and is supplied by the MLS framework itself.

The next objective was to investigate the relationship between problem instance characteristics and heuristic performance. One of the fundamental problems in the application of metaheuristics to real-world problems is knowing *which* metaheuristic to use. Many research papers and case studies simply apply one, or several, metaheuristics, with little justification as to why these were chosen. Over time a body of knowledge can be assembled that indicates the general success of one method over another for a particular problem class, but this approach suffers from several weaknesses. Firstly, there is no consistency in how heuristics are specified and implemented; each researcher is free to implement the heuristics according to their own interpretation, making direct comparisons between the heuristics

difficult. Secondly, the goal in publishing new heuristics seems to be to show how the new technique is better than existing methods. This tends to result in the researcher choosing comparison heuristics against which the new technique excels.

The use of the MLS framework automatically compensates for the first problem. Heuristics are expressed in the same framework, and share most of the same operational characteristics, so comparisons between them can be focused on the modules where they differ.

In this chapter we have attempted to develop some basic methods by which a more scientific approach can be taken to deciding which heuristic would be most suited for a given problem. Our focus has been on demonstrating the validity of such an approach, rather than honing these methods. We leave further development of these methods to future research.

Our demonstration-of-feasibility approach has been to consider heuristics in pairs, and to use the problem instance characteristics as inputs to a classification model that predicts which of the two heuristics will achieve better performance on each problem instance. A correct classification rate of 50% would mean that our models were no better than random, however we were consistently able to achieve classification rates better than this. The analysis in this chapter has clearly demonstrated that it is possible to perform prior analysis of ASRP problem instances, and determine which of two heuristics would be most appropriate. The techniques at this stage are not able to make this selection with perfect accuracy, however they provide enough of an edge that with a large number of problem instances they would provide a significant advantage over a random selection, or simply using one of the heuristics.

The final objective of this chapter was to demonstrate the construction of several hybrid heuristics. The goal here was to demonstrate the ease with which quite complex multi-phase heuristics can be specified *with no new programming*; these heuristics simply used existing MLS modules combined in new ways. These hybrids were run on the set of ASRP problem instances, and did manage to outperform the "regular" heuristics on some instances.

Coda

▼ Summary

In this chapter we have performed extensive computational experiments using MLS expressions of some fundamental metaheuristic families, on a variety of ASRP problem instances. Analysis of these results provided evidence that it is possible to predict which heuristics will perform better on certain problem instances based on an analysis of the characteristics of that problem instance.

▼ Link

In the next chapter we consider methods of generating "interesting" problem instances.

CHAPTER 7

Heuristic Problem Design

In this chapter we explore various methods for developing "interesting" sets of problem instances. Several heuristics that select a subset of "maximally diverse" instances from a larger set are developed and tested. A novel method of "designing" interesting problem instances using a local search approach, is presented and executed as a proof-of-concept.

7.1 Introduction

One of the factors that became apparent from the experiments in Chapter 6 is that problem instance design is an important consideration. Instead of using a large number of problem instances, more insight can potentially be gained from a smaller set that has been carefully chosen. Creating this set is a challenging problem in itself, even without the application of heuristics to these problem instances. The intention of this chapter is to explore some ways that an interesting test set of problem instances can be obtained.

The motivating goal is to derive a small set of problem instances that allow us to distinguish between the behaviour of various heuristics. We do this by attempting to find a set of instances that are **diverse** with respect to the problem characteristics defined in Chapter 6. A small set of instances is essential to

make the cost of the computational experiments manageable. Recall that a problem **characteristic** is a metric that can be calculated for a problem instance as part of a pre-processing step.

In order to increase the richness of the problem space, just for the purposes of developing the problem selection techniques, we introduce another element to the basic Arc Subset Routing Problem: **penalties for non-service**. The purpose of modifying the problem is to create a "richer" problem space, to allow greater diversity.

Our approach is to generate a large number of trial instances and then select a maximally diverse subset of these. We first test some of the methods on a tiny set of instances, and then extend the approach to a giant set of instances.

The goal in this chapter is to explore several ideas briefly, testing their potential and moving on, rather than investigating any particular idea in great detail. This chapter is structured as a series of small investigations into different aspects of the problem design.

7.2 New problem features

For the experiments in this chapter we make several modifications to our approach. The introduction of penalties prompts the development of techniques to assign those penalties to arcs during problem instance generation. A number of new problem characteristics are developed.

7.2.1 The Arc Subset Routing Problem with Penalties

We define the Arc Subset Routing Problem with Penalties (ASRPP). This problem is equivalent to the ASRP, except that any arc that is not included in the solution incurs a penalty for non-service. The objective function becomes the sum of the rewards for all the arcs included in the solution, minus the sum of the penalties for all the arcs that are not included in the solution.

We modify the formulation of the ASRP from Section 3.1.3 to give the following maximization objective function for the ASRPP; the remainder of that formulation is the same.

$$\sum_{e \in E} r_e x_e - \sum_{e \in E} p_e (1 - x_e)$$

7.2.2 Problem characteristics

In Section 6.2.5, 22 problem characteristics were defined and used in the modelling and analysis for that chapter; in this chapter we revise this set of characteristics, creating an improved list. The motivation for this change was to make the characteristics more general and comparable across problem instances of different sizes. For example, the *number of pendant arcs* has been changed to the *proportion of pendant arcs*. Also, variables that previously measured the reward as minimums and maximums have been changed to the coefficient of variation (the standard deviation divided by the mean). A number of measures relating to the introduction of penalties have been added, and some redundant characteristics have been removed, for example the *budget* is not necessary since the *budget to cost ratio* contains more relevant information. The final 18 problem characteristics are listed below, and those that differ from Chapter 6 are described.

- ARCS. (*Unchanged*).
- AVG_ARC_ADJACENCY. (*Unchanged*).
- AVG_DEPOT_SHORTEST_PATH. (*Unchanged*).
- AVG_NODE_DEGREE. (*Unchanged*).
- AVG_SHORTEST_PATH. (*Unchanged*).
- BUDGET_TO_COST_RATIO. (*Unchanged*).
- CV_NODE_INCIDENCE_REWARD. The incidence reward for a node is the sum of the rewards of all incident arcs. This metric is the coefficient of variation of these calculated across all nodes.
- CV_PENALTY. The coefficient of variation of the penalties of all arcs.
- CV_REWARD. The coefficient of variation of the rewards of all arcs.
- DENSITY. (*Unchanged*).
- DEPOT_DEGREE. (*Unchanged*).
- DEPOT_ROWS_FROM_SIDE. (*Unchanged*).
- MEAN_REWARD_PENALTY_RATIO. For each arc the ratio of the reward to the penalty is calculated and then the mean of these is found.
- NODES. (*Unchanged*).
- PROPORTION_PENDANT_ARCS. A pendant arc is incident on a node of degree one. This metric is the count of these divided by the number of arcs in the graph.
- PROPORTION_STRAIGHTLINE_ARCS. A straightline arc is connected to only two other arcs, one at either end. This metric is the count of these divided by the number of arcs in the graph.
- STD_REWARD_PENALTY_RATIO. For each arc the ratio of the reward to the penalty is calculated and then the standard deviation of these is found.
- STD_NODE_DEGREE. The degree of a node is the number of incident arcs, in {1,2,3,4}. This metric is the standard deviation across all nodes.

7.2.3 Penalty assignment methods

The methods for assigning rewards that were defined in Chapter 6 can also be used to assign penalties: uniform between a range, one reward seed with iterative deviation of adjacent arcs' penalties within a range, and multiple seeds with iterative deviation of adjacent arcs' penalties within a range.

The additional method we introduce is that of assigning the penalties to be proportional to rewards, with a random factor. This method has two parameters: the *proportion* and the *maximum deviation*. For an arc a, with penalty p_a and reward r_a, the penalty is calculated as follows:

$$p_a = \theta . r_a \, (1 + \delta)$$

where θ is the **proportion** of the reward for arc a that the penalty is set to, and δ is the deviation, sampled from a uniform distribution δ ~`Uniform`(*-maxDev*,*maxDev*).

7.3 The Maximally Diverse Subset Selection Problem

We define the **Maximally Diverse Subset Selection Problem** (MDSSP). Given a set of problem instances, and a distance metric between every pair of instances, the MDSSP is to select a subset of a specified size, such that the sum of the pair-wise distances is maximized.

The MDSSP can be formulated as a binary integer program. Given a set P of points $p = (p_1, p_2, \ldots, p_N)$ in N-dimensional space, and a distance d_{ij} between each pair of points i and j, select a subset $S \subset P$ with maximum total distance (the sum of the distances between all included points), where the size of S, $|S|$ is specified.

$$\text{Max} \quad \sum_{\substack{p,q \in P, \\ p<q}} d_{pq} y_{pq} \tag{1}$$

$$\text{s.t.} \quad x_p + x_q - 2y_{pq} - E_{pq} = 0 \qquad (p,q \in P \mid p < q) \tag{2}$$

$$\sum_{p \in P} x_p = |S| \tag{3}$$

$$x_p, y_{pq}, E_{pq} \in \{0,1\} \tag{4}$$

where x_p represents whether point p is included in the selected subset S ($x_p = 1$ if included, 0 otherwise). y_{pq} represents whether the distance between points p and q should be counted in the objective function (1 if included, 0 otherwise), and constraint (2) forces y_{pq} to the appropriate value; when x_p and x_q are both 1 then y_{pq} is 1, otherwise it is 0. The E_{pq} variables are only included to balance the equations, their values are not important otherwise (for example they are equal to 1 when $x_p = 1$ but $x_q = 0$; we want y_{pq} to be zero so we need to balance the equation). Constraint (3) sets the subset size.

Note that we only consider the distances d_{pq}, not the reverse distances d_{qp}. This is represented by setting $p < q$. So for a set of points of size N, we have T distances, where

$$T = \sum_{n=1}^{N-1} n$$

With a set of points of size N there $N + 2T$ variables and T+1 constraints (plus the binary constraints).

The MDSSP is similar to some other problems in the literature, specifically maximum weighted clique problems. Urošević et al. [240] study a minimization problem where the number of arcs is specified, rather than the number of vertices, which they call the Minimum Weighted k-Cardinality Tree Problem (MWkCP), and present a Variable Neighbourhood Decomposition Search heuristic. Fischetti et al. [99] show that this problem is *NP*-hard. The MDSSP can be transformed into the MWkCP, therefore the MDSSP is also *NP*-hard. Macambira and de Souza [178] study the polyhedral properties of a variation where the weights to be maximized are on the vertices rather than the arcs of the subgraph, and Macambira [179] applies a basic Tabu Search heuristic to this same problem. More recently Pullan

[215] applies a metaheuristic called Phased Local Search to a variation where the weights are on the edges, but the size of the vertex set is not specified.

7.4 A tiny illustrative problem

In order to test and illustrate some of the concepts developed in this chapter we define a "tiny" problem. A set of 216 instances was generated, using a range of generation settings, as defined below. The goal of this exercise is to select the 20 most "diverse" instances.

The generation process has the following settings, which were varied factorially. These settings were described in Chapter 6.

- **2** × Generation methods: GRIDDESELECT, GRIDGROW-3-SEEDS
- **3** × Generation densities: 40%, 60%, 80%
- **2** × Reward setting methods: `Uniform`(10, 20), 3 seeds max pct deviation(seed1=5, seed2=10, seed3=20, maxDev=0.1)
- **3** × Penalty setting methods: `Uniform`(5, 10), 3 seeds max pct deviation(seed1=2, seed2=5, seed3=10, maxDev=0.1), Proportional to rewards(proportion=0.4, maxDev=0.1)
- **2** × Depot assigning methods: most rich, least rich
- **3** × Budget proportions: 50%, 75%, 100%

7.5 Measures of distance

We define "diversity" with respect to the problem characteristics in a number of different ways. Central to these definitions is the concept of the **distance** between two problem instances. We adapt two common measures of distance between two points in a multi-dimensional space. There are two factors that must be considered:

- Some characteristics are on a completely *different scale* to others. For example TOTAL_REWARD can go into the thousands, whereas DENSITY varies between zero and one.
- Some characteristics are more *important* than others when distinguishing between problems. For example DENSITY is one of the most obvious ways that two instances can differ, and a way that it seems reasonable to assume would affect heuristic performance, whereas DEPOT_ROWS_FROM_SIDE might not be so important.

7.5.1 Euclidean distance

The simplest distance measure is **Euclidean distance** (ED), treating each problem characteristic as a dimension. The Euclidean distance between points $P = (p_1, p_2, \ldots, p_n)$ and $Q = (q_1, q_2, \ldots, q_n)$ is defined as:

$$ED_{PQ} = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \ldots + (p_n - q_n)^2} = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}$$

The Euclidean distance treats all dimensions equally, so characteristics that have a larger scale will potentially have a larger impact. A way to account for the different scale of characteristics is to

calculate the **normalized Euclidean distance** (NED) by first dividing each characteristic value by its standard deviation.

$$NED_{PQ} = \sqrt{\sum_{i=1}^{n} \frac{(p_i - q_i)^2}{\sigma_i^2}}$$

where σ_i is the standard deviation of the values of characteristic i. Normalizing the Euclidean distance introduces the new disadvantage that it is dependent on the set of problem instances being examined, since the standard deviation is calculated from these instances. Nevertheless, it may be used as a valid measure to compare the distances between any pair of problem instances in a given set.

If we calculate the NED for the 216 instances of the tiny problem, using the characteristics defined in Section 7.2.2, we get 23,220 individual distances, with the following normal distribution:

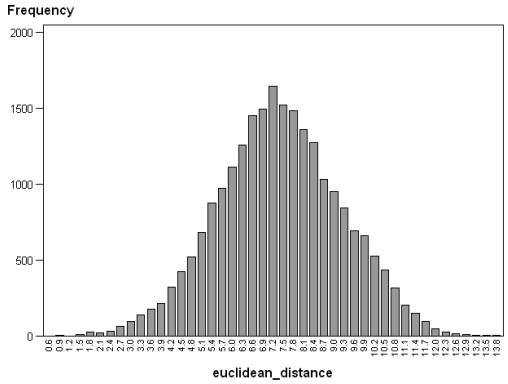


Figure 7.1: Distribution of normalised Euclidean distances for the tiny problem

Consider that some characteristics are more *important* than others when distinguishing between problems. We can further extend the ED and the NED by introducing **weights** for each characteristic, representing their relative importance, to obtain the **weighted Euclidean distance** (WED):

$$WED_{PQ} = \sqrt{\sum_{i=1}^{n} w_i (p_i - q_i)^2}$$

where w_i is the weight associated with characteristic i, which subsumes the standard deviation if necessary.

7.5.2 Interpoint distance

An alternative measure of distance is known as the **Mahalanobis distance** (MD). It differs from the Euclidean distance in that it takes into account the correlations of the data set and is scale-invariant. It is commonly used in discrimination analysis and clustering. The Mahalanobis distance is calculated as a measure of the distance from the point of observation to the mean of the distribution, rather than as the distance between two points. For this reason it is not as suitable for our purposes. However, there is a variation that does seem appropriate.

Mahalanobis distance is usually defined from a group of values with mean $\mu = (\mu_1, \mu_2, \ldots \mu_N)^T$ and covariance matrix S for a multivariate vector $x = (x_1, x_2, \ldots x_N)^T$ as

$$MD_x = \sqrt{(x-\mu)^T S^{-1}(x-\mu)}$$

The alternative version, sometimes known as the **generalized interpoint distance** (GID), is defined between two points from the same distribution $P = (p_1, p_2, \ldots p_N)^T$ and $Q = (q_1, q_2, \ldots q_N)^T$ as

$$GID_{xy} = \sqrt{(p-q)^T S^{-1}(p-q)}$$

If the covariance matrix is the identity matrix then the generalized interpoint distance reduces to the Euclidean distance. If it is diagonal then it reduces to the normalized Euclidean distance defined above.

The interpoint distance would be a potential alternative to the Euclidean distance, and it accounts for the problem of different variable scales of the characteristics, however still does not incorporate the importance of characteristics.

The generalized interpoint distances were calculated between all instances of the tiny problem. An important point to remember is that the covariance matrix must be invertible, so none of the characteristics can be a linear combination of the others. The following distribution of interpoint distances was obtained:

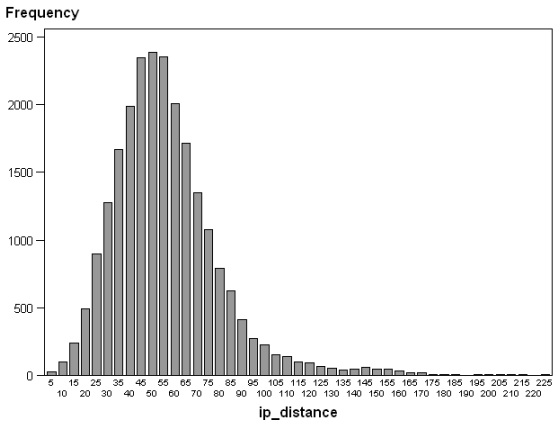


Figure 7.2: Distribution of generalized interpoint distances for the tiny problem

An interesting question is whether the normalized Euclidean distance and the generalized interpoint distance give the same ranking to the instances. Plotting the interpoint distance against the Euclidean distance gives the following scatter plot:

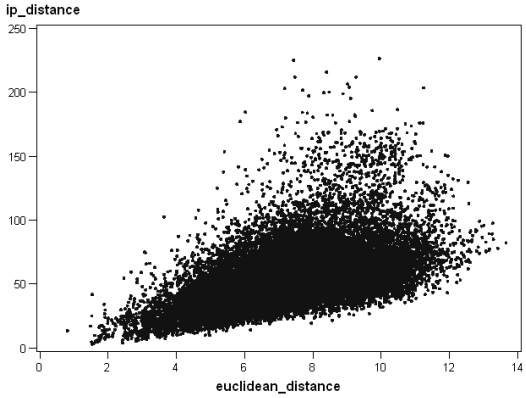


Figure 7.3: Scatter plot of GID vs NED for the tiny problem

The shape is broadly linear, and the two distributions have a correlation coefficient of 0.52, but it can be seen that there are considerable differences between the rankings given by the two distance measures. Note that the values of the Euclidean distance and interpoint distance measures are not directly comparable as they have different scales.

One final note is that like the normalized Euclidean distance, which is dependent of the problem set under consideration through the standard deviation of each characteristic, the generalized interpoint distance is dependent on the specific problem set under consideration too, via the covariance matrix.

7.5.3 Inter-instance metrics

The Euclidean and interpoint distances, and their variations, give a measure of the similarity of two specific instances. In addition we define several aggregate metrics for a particular subset of arcs. If we are selecting s instances from a set of n instances, $S \subseteq N$, then the set of individual distances between the elements of S is $D_S = \{d_{pq} \mid p,q \in S\}$, and the cardinality of D_S is given by the following formula:

$$|D_S| = \sum_{i=1}^{s-1} i$$

The **total distance** (TD) for a subset S of problem instances is the sum of the distances between every pair of instances:

$$TD_S = \sum_{i \in S} \sum_{j \in S \setminus i} d_{ij}$$

The **total individual distance** (TID) of an instance p for a subset S is the sum of the distances from p to each of the other instances in S:

$$TID_p^S = \sum_{i \in S \setminus p} d_{pi}$$

7.6 Heuristics

As discussed in Section 7.3, the Maximally Diverse Subset Selection Problem is *NP*-hard, which means that we expect large instances to be unsolvable by standard integer programming algorithms. However, even the tiny problem of selecting 20 instances from 216 is too large to solve optimally on a desktop computer. There are 8.7×10^{27} possible subsets, so explicit enumeration is not a valid solution method. With the integer programming formulation presented in section 7.3, the tiny problem requires 46656 variables, 23221 constraints, and 93096 constraint coefficients, and the commercial-grade solver built into the optimization software SAS/OR was unable to find a solution. For this reason we develop some basic heuristics. Each of the following four heuristics can be performed using any measure of distance available.

7.6.1 Heuristic H1

Heuristic H1 uses selection based on total individual distance.

1. Calculate the set D_N of distances between all pairs of instances in N.
2. For each instance, sum the distances from this instance to every other instance to calculate its total individual distance.
3. Sort the instances in N by descending order of total individual distance.
4. Select the top s distances.

7.6.2 Heuristic H2

Heuristic H2 uses selection based on ranking individual distances.

1. Calculate the set D_N of distances between all pairs of instances in N.
2. Sort all of these distances in descending order.
3. Process the list, one distance d_{ij} at a time, adding both instance i and instance j to S, until there are s distinct instances in S.

7.6.3 Heuristic H3

Heuristic H3 iteratively discards a “non-promising” instance, and recalculates the remaining distances.

1. Calculate the set D_N of distances between all pairs of instances in N.
2. For each instance, sum the distances from this instance to every other instance to calculate its total individual distance.
3. Sort the instances by ascending order of total individual distance.
4. Discard the instance p at the top of the list (lowest total individual distance), to give the remaining subset $R = N \setminus p$.
5. Return to step 2 and recalculate the total individual distances with respect to the remaining subset R. Repeat until the list of remaining instance R contains s instances, and R becomes S.

7.6.4 Heuristic H4

Heuristic H4 performs the opposite process to H3; it iteratively includes the "most-promising" instance, and recalculates remaining distances.

1. Calculate the set D_N of distances between all pairs of instances in N.
2. For each instance, sum the distances from this instance to every other instance to calculate its total individual distance.
3. Sort the instances by descending order of total individual distance.
4. Select the instance p at the top of the list (highest total individual distance). Include p in S; the remaining instances are in $R = N \setminus p$.
5. Return to step 2 and recalculate the total individual distances with respect to the remaining subset R. Repeat until S contains s instances.

7.7 Solving the tiny problem

The tiny problem of choosing 20 maximally diverse instances from the set of 216 instances server was solved using each of the four heuristics H1-H4, with two separate distance measures; normalized Euclidean distance and generalized interpoint distance. In addition, 1000 random subsets were selected, to benchmark the heuristics' performance. Selecting the best of these could be considered another heuristic.

Table 7.1 gives the results of the computational experiment. The running times are dependent on the implementation used, but can be effectively compared to each other. The heuristics were coded in the SAS data analysis language, and were run on a laptop computer running Windows XP Pro with a 2.4Ghz processor and 3GB RAM.

Table 7.1: Results for the tiny problem

Distance measure	Heuristic	Rank	Total distance	Time (s)
normalised Euclidean distance	Best random	4	1588.92	182.2
normalised Euclidean distance	Worst random	7	1202.51	
normalised Euclidean distance	Mean random	5	1405.66	
normalised Euclidean distance	H1	2	1647.97	0.2
normalised Euclidean distance	H2	6	1272.24	0.2
normalised Euclidean distance	H3	1	1814.85	48.6
normalised Euclidean distance	H4	3	1639.11	6.3
interpoint distance	Best random	5	15050.29	188.4
interpoint distance	Worst random	7	7358.55	
interpoint distance	Mean random	6	10608.13	
interpoint distance	H1	2 (equal)	22809.28	0.1
interpoint distance	H2	4	20752.69	1.5
interpoint distance	H3	1	22968.60	28.3
interpoint distance	H4	2 (equal)	22809.28	6.1

The heuristics all perform better than all the random selections for the GID, and mostly for the NED. Heuristic 3 performed the best under both distance measures, although it required significantly more computation time, as expected since the distances have to be recalculated at each iteration. The time for the "best" random metric is the time required to generate all 1000 instances.

Heuristic 2 performed quite poorly under both distance measures, but especially with the NED. This poor performance is possibly due to the fact that H2 does not consider the total distance associated with a particular problem instance; if it has one very large distance then it is likely to be included, even if all its other distances are small. As a general rule, heuristics that rank items individually are outperformed by those which rank on more global bases.

It appears that the two distance measures give quite different subsets. Even though Heuristic 3 was the best in both cases, there was only a 50% overlap in the instances selected (10 instances).

7.8 Additional measures of diversity

The normalized Euclidean distance and the generalized interpoint distance (based on Mahalanobis distance) are measures between pairs of instances. These can be used to describe subsets of instances by calculating the total distance, which was our measure of diversity in the previous section. The heuristics developed above provide methods to get a subset of instances that have maximal total distance. We now introduce some other measures that describe how diverse these instances are. The following metrics are calculated for *each characteristic*, with respect to a set of instances.

The following metrics also refer to the *standardized* values of each characteristic, found by dividing each value by the standard deviation of values for that characteristic. This avoids the problem of having characteristics with different scales.

7.8.1 Total absolute difference for a characteristic

The **total absolute difference** (TAD) for a characteristic is the sum of the absolute differences between the values of that characteristic, for all instance pairs:

$$TAD_c = \sum_{i=1}^{s-1} \sum_{j=i+1}^{s} \left| x_j^c - x_i^c \right|$$

where TAD_c is the total absolute difference for characteristic c, x_i^c is the value of characteristic c for instance i.

7.8.2 Average consecutive difference for a characteristic

If all the values for a characteristic are ordered, and the differences between consecutive values are found, then let the **average consecutive difference** (AvgCD) for a characteristic be the mean of these:

$$AvgCD_c = \frac{\sum_{i=1}^{s-1} \left(x_{i+1}^c - x_i^c \right)}{s}$$

where $AvgCD_c$ is the average consecutive difference for characteristic c, s is the number of instances (and hence s-1 is the number of consecutive ordered differences), x_i^c is the value of characteristic c for instance i, and the instances are in ascending order of the values of c.

7.8.3 Standard deviation of consecutive differences for a characteristic

If all the values for a characteristic are ordered, and the differences between consecutive values are found, then let the **standard deviation of consecutive differences** (StdCD) for a characteristic be the sample standard deviation of these:

$$StdCD_c = \sqrt{\frac{\sum_{i=1}^{s-1} \left(\left(x_{i+1}^c - x_i^c \right) - AvgCD_c \right)^2}{s-1}}$$

where $StdCD_c$ is the standard deviation of consecutive differences for characteristic c, s is the number of instances (and hence s-1 is the number of consecutive ordered differences), x_i^c is the value of characteristic c for instance i, and the instances are in ascending order of the values of c.

7.8.4 Coefficient of variation of consecutive differences for a characteristic

The coefficient of variation of a sample is the ratio of the standard deviation to the mean. It is a dimensionless number that can be used to compare the amount of variance between populations with different means. If all the values for a characteristic are ordered, and the differences between consecutive values are found, then let the **coefficient of variation of consecutive differences** (CVCD) for a characteristic be the coefficient of variation of these:

$$CVCD_c = \frac{StdCD_c}{AvgCD_c}$$

where $CVDC_c$ is the coefficient of variation of consecutive differences for characteristic c, $StdCD_c$ is the standard deviation of consecutive differences for characteristic c, and $AvgCD_c$ is the mean of consecutive differences for characteristic c.

7.8.5 Maximum consecutive difference for a characteristic

If all the values for the characteristic are ordered, and the differences between consecutive values are found, then let the maximum consecutive difference (MaxCD) for a characteristic be the largest of these:

$$MaxCD_c = \max_{i=1..(s-1)} \left(x_{i+1}^c - x_i^c \right)$$

where $MaxCD_c$ is the maximum consecutive difference for characteristic c, s is the number of instances, x_i^c is the value of characteristic c for instance i, and the instances are in ascending order of the characteristic values.

7.8.6 Discussion of the diversity measures

The diversity measures defined above for a characteristic attempt to quantify whether the values for that characteristic are *finely grained* and *non-clumped*. In a maximally diverse subset, we would expect the TAD and the AvgCD to be *maximized* for each characteristic, which would mean they were finely grained. We would expect MaxCD, StdCD and CVCD to be *minimized* for each characteristic, which would mean the values are evenly spread, with little "clumping" of values.

Suppose we have are selecting four problem instances, and these are measured with two characteristics. Figure 7.4 plots some possible scenarios for the values of the characteristics.

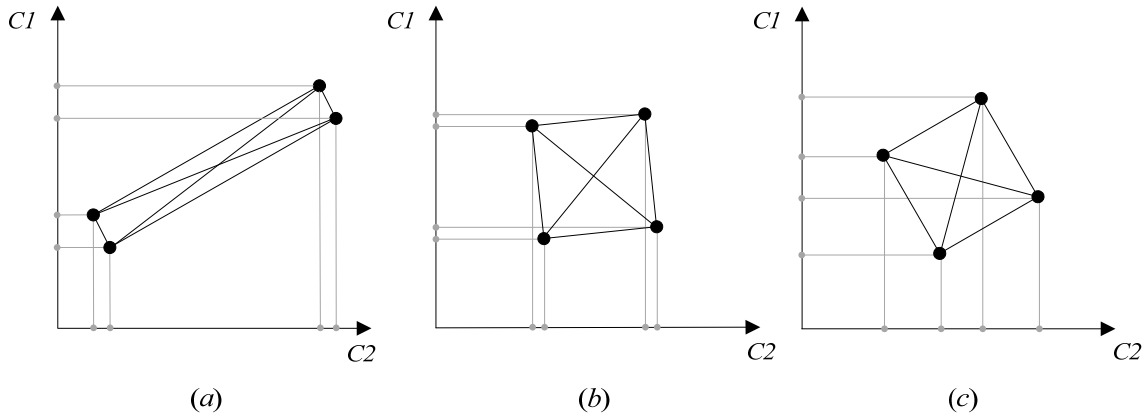


Figure 7.4: Examples of problem instances with differing diversity

The three scenarios all have the same total distance, however they are not all equally desirable.

Figure 7.4(a) has two clusters of points; these points clearly do not have evenly spread and finely grained characteristic values. Figure 7.4(b) and Figure 7.4(c) are identically "shaped", and have the same total distance, as well as each point having the same total individual distance; there are no clusters. Table 7.2 gives the values of the diversity measures for these example sets. These are based on a normalized Euclidean distance, where (a) is based on a rectangle with sides of length 10 and 60, and (b) and (c) are based on a square with sides of 38.3 (chosen to make the total distances equal). Note that the diagrams are representative only, but the calculations of the metrics are all consistent; the consecutive differences in each characteristic for (c) are evenly spaced.

Table 7.2: Diversity measures for the example sets

Metric	Characteristic	a	b	c
TD		261.7	261.7	261.7
TAD	C1	115.7	163.8	**171.4**
	C2	**227.4**	163.8	171.4
AvgCD	C1	11.2	14.7	**17.1**
	C2	**19.6**	14.7	17.1
StdCD	C1	3.5	14.5	**0.0**
	C2	27.0	14.5	**0.0**
CVCD	C1	0.3	1.0	**0.0**
	C2	1.4	1.0	**0.0**
MaxCD	C1	**15.2**	31.5	17.1
	C2	50.8	31.5	**17.1**

The cells highlighted in the table are those with the best values. Although (a) has some best values for a single characteristic, overall (c) has the best diversity measures, and this confirms the impression by eye that the characteristic values are finely grained and evenly spaced.

7.9 A giant selection problem

We now present a much larger selection problem to test the ideas developed in this chapter, and using a much more finely grained generation mechanism. In this "giant" problem we desire to select 300 instances from a set of 150,000.

The generation process has the following settings, which were varied factorially. These settings were described in Chapter 6.

- **3** × Generation methods: GRIDDESELECT, GRIDGROW-1-SEED, GRIDGROW-3-SEEDS
- **17** × Generation densities: 40% to 80% by 3% increments
- **7** × Reward setting methods:
  - Uniform: (10, 20), (5, 20), (5, 50)
  - 1 seed max pct deviation: (seed=20, maxDev=0.05), (seed=20, maxDev=0.1), (seed=20, maxDev=0.25)
  - 3 seeds max pct deviation: (seed1=5, seed2=20, seed3=50, maxDev=0.1)
- **10** × Penalty setting methods:
  - Uniform: (5, 20), (5, 20), (5, 50)

    - Proportional to rewards: (proportion=0.1, maxDev=0.1), (proportion=0.3, maxDev=0.3), (proportion=1.1, maxDev=0.3)
    - 1 seed max pct deviation: (seed=20, maxDev=0.05), (seed=20, maxDev=0.1), (seed=20, maxDev=0.25)
    - 3 seeds max pct deviation: (seed1=5, seed2=20, seed3=50, maxDev=0.05)
- **3** × Depot assigning methods: random, most rich, least rich
- **18** × Budget proportions: 40% to 108% by 4% increments

This gives a total of 3×17×3×18×7×10 = 192,780 instances. The generation procedure generated the instances for the GRIDGROW-1-SEED and GRIDGROW-3-SEEDS graph generation methods quickly – within a day. The GRIDDESELECT instances took much longer, however. The instances with lower densities were produced first, and these were taking a minute each! This time slowly decreased as the density increased (fewer arcs to remove in the deselection process), however it reduced only to 38s. Time constraints forced the decision after several weeks to stop the generation at 150,000 instances overall, corresponding to all the instances for the GRIDGROW-1-SEED and GRIDGROW-3-SEEDS graph generation methods and those instances for the GRIDDESELECT method with the lowest densities. This decision was justified by the fact that the difference in graph structures between the GRIDDESELECT and GRIDGROW-1-SEED methods is most dramatic at low densities.

7.10 Solving the giant problem

The tiny problem revealed that Heuristic 3 seems to be the most effective. Unfortunately this heuristic is impractical for large problems like the giant problem, and the same limitation applies to all the heuristics from Section 7.6. The difficulty is that they all involve the calculation of the distance between all pairs of instances. This involves a join of the problem characteristic dataset to itself; for a set of 150k instances this would give 11,249,925,000 distances to compute, and would take more computing time and resources than were available.

We develop a modified version of the heuristic to cope with large data volumes. The general approach is to break the superset of instances into smaller samples and select a diverse subset from each using Heuristic 3, then combine these subsets and repeat the process until a subset of the desired size is obtained. The normalized Euclidean distance was chosen as the distance measure, since it requires much less computational effort to calculate.

We use terms “sample” and “subset” in this section as follows:

- A **sample** is a random selection from the current superset, used to break the superset into manageable pieces.
- A **subset** is the diverse selection chosen from the sample using the heuristic.

The first configuration that was attempted was to select 50 samples of 3000, choosing 100 from each. This would result in a new superset of 5000 instances. However, the process of choosing 100 from 3000 still took prohibitively long; the heuristic was taking half a minute for each discard, which would have taken almost a whole day just to process the first 3000 instance sample. The samples need to be smaller.

After considerable trial and error, three different methods were performed:

Method 1. Samples of 500, choosing subsets of 100. This progressed in five iterations for a total time of approximately 13 hours.

Method 2. Samples of 100, choosing subsets of 20. This progressed in 5 iterations for a total time of approximately 1.5 hours.

Method 3. Samples of 20, choosing subsets of 15. This progressed in 19 iterations for a total time of approximately 1.8 hours.

For some of the iterations, a slightly different number of selections was made, where this made sense. For example, the fourth and fifth iterations of Method 1 selected 300 instances from the samples of 600.

There are clearly many other possible configurations possible, however for our purposes these three are sufficient to achieve the goals of performing the selection, and understanding whether there is an impact of using different selection methods.

7.10.1 Results

The results from the three selection methods are remarkably close. Each method selected 300 instances from 150,000 and Figure 7.5 shows the overlap of instances selected.

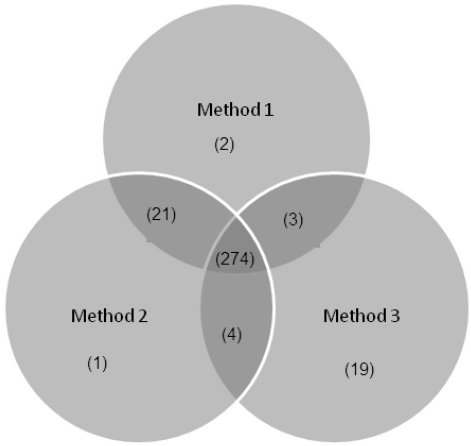


Figure 7.5: Overlap of instances for the three methods on the giant problem

These results are extremely surprising. 274 instances were selected by all three methods. Consider that each of these methods involved many iterations of taking different random samples and repeated remixing the remainders. This strong overlap implies two things:

1. The choice of sampling method is not hugely important, since very similar subsets get selected anyway.

2. The subsets are close to being the most diverse possible, with respect to the diversity measure being used for selection: total distance.

If we consider *total distance*, which is the measure used in the selection heuristics, we see that the total distance is very close, with Method 1 producing a slightly better result, but the other two methods are within 0.1% of Method 1. Table 7.3 gives the total distances of the subsets selected, along with the best of 100 random selections. By this measure, all of the heuristics are almost twice as effective as the random samples.

Table 7.3: Total distance for the results of giant problem

Method	Total distance
1	528910
2	528875
3	528461
Random	272190

The next step is to consider the results using the characteristic-based diversity measures from Section 7.8. For these results the three methods are considered, along with the "best" random sample, to provide some basis of comparison.

7.10.2 Total absolute difference

Table 7.4 shows the *total absolute difference* values for each characteristic, and for each selection method. The best (highest) TAD for each characteristic is highlighted.

Table 7.4: Total absolute distance by characteristic for the giant problem

Characteristic	Random	Method 1	Method 2	Method 3
Arcs	48131	63430	63475	**63844**
Avg_arc_adjacency	41893	72039	**72169**	72123
Avg_depot_shortest_path	46696	110355	**110588**	110490
Avg_node_degree	47397	69039	68949	**69257**
Avg_shortest_path	39915	94150	94155	**94255**
Budget_to_cost_ratio	52912	57050	**57110**	57098
CV_node_incidence_reward	40494	**132562**	132416	131838
CV_penalty	48953	**115719**	115619	115357
CV_reward	46902	120909	**121072**	120751
Density	48282	69321	69361	**69593**
Depot_degree	49129	53898	54038	**54159**
Depot_rows_from_side	49108	68573	**68630**	68292
Mean_reward_penalty_ratio	38723	70847	70966	**71052**
Nodes	48937	57086	57073	**57185**
Proportion_pendant_arcs	46314	68469	68529	**68580**
Proportion_straightline_arcs	40015	**95409**	95024	94995
Std_reward_penalty_ratio	40817	**206124**	206040	205845
Std_node_degree	50206	52364	52750	**53112**
Average	45824	87630	**87665**	87657

The three heuristic methods are significantly better than random with this measure, which is reassuring since *total absolute difference* is related to the *total distance* measure used to make the selections.

7.10.3 Average consecutive difference

Table 7.5 shows the *average consecutive difference* values for each characteristic, and for each selection method. The best (highest) AvgCD for each characteristic is highlighted.

Table 7.5: Average consecutive distance by characteristic for the giant problem

Characteristic	Random	Method 1	Method 2	Method 3
Arcs	**0.01075**	**0.01075**	**0.01075**	**0.01075**
Avg_arc_adjacency	**0.01136**	**0.01136**	**0.01136**	**0.01136**
Avg_depot_shortest_path	0.02385	0.02694	0.02694	**0.02704**
Avg_node_degree	**0.01236**	**0.01236**	**0.01236**	**0.01236**
Avg_shortest_path	0.01725	**0.02225**	**0.02225**	**0.02225**
Budget_to_cost_ratio	**0.01096**	0.01096	0.01096	0.01096
CV_node_incidence_reward	0.02297	**0.03107**	**0.03107**	**0.03107**
CV_penalty	0.01879	**0.02509**	**0.02509**	**0.02509**
CV_reward	0.01794	0.02473	0.02473	**0.02475**
Density	**0.01164**	**0.01164**	**0.01164**	**0.01164**
Depot_degree	**0.00958**	**0.00958**	**0.00958**	**0.00958**
Depot_rows_from_side	0.01544	**0.02008**	**0.02008**	**0.02008**
Mean_reward_penalty_ratio	0.01203	**0.02083**	**0.02083**	**0.02083**
Nodes	0.01290	**0.01359**	**0.01359**	**0.01359**
Proportion_pendant_arcs	0.01261	**0.01428**	**0.01428**	**0.01428**
Proportion_straightline_arcs	0.01643	**0.02027**	**0.02027**	**0.02027**
Std_reward_penalty_ratio	0.02406	**0.10329**	**0.10329**	**0.10329**
Std_node_degree	0.01277	0.01428	0.01428	**0.01457**
Average	0.01520	0.02241	0.02241	**0.02243**

The most obvious observation here is that the selection methods are the same for most of the characteristics, and the same as the random version for some also. This is initially surprising, however after consideration it makes some sense. The first factor is that most of the instances from each selection method overlap (274 out of 300), so the variation is already limited. The other factor is that many of these characteristics take a discrete set of values, so there are only so many consecutive distances that are possible; once these are represented then any new instances will contribute a consecutive distance of zero. For example, suppose there are five discrete values that a characteristic may take, then there are only four consecutive distances, and the remainder will be repeats with consecutive distances of zero. For that reason, this metric may not be very useful.

7.10.4 Standard deviation of consecutive differences

Table 7.6 shows the *standard deviation of consecutive differences* for each characteristics and selection method. The best (lowest) StdCD for each characteristic is highlighted.

Table 7.6: Standard deviation of consecutive differences by characteristic for the giant problem

Characteristics	Random	Method 1	Method 2	Method 3
Arcs	**0.04534**	0.04818	0.04818	0.04818
Avg_arc_adjacency	**0.11305**	**0.11305**	**0.11305**	**0.11305**
Avg_depot_shortest_path	0.13516	**0.04453**	0.04469	0.04468
Avg_node_degree	**0.15085**	**0.15085**	**0.15085**	**0.15085**
Avg_shortest_path	**0.05416**	0.06742	0.06742	0.06573
Budget_to_cost_ratio	**0.04084**	0.04098	0.04098	0.04085
CV_node_incidence_reward	0.09312	0.06004	0.06005	**0.05976**
CV_penalty	**0.05357**	0.05545	0.05540	0.05531
CV_reward	0.04523	**0.03662**	0.03667	0.03740
Density	**0.03727**	0.03951	0.03946	0.03914
Depot_degree	**0.09531**	**0.09531**	**0.09531**	**0.09531**
Depot_rows_from_side	**0.08316**	0.09432	0.09432	0.09432
Mean_reward_penalty_ratio	0.10871	0.09305	**0.09305**	0.09307
Nodes	**0.01928**	0.02854	0.02854	0.02783
Proportion_pendant_arcs	**0.02059**	0.02465	0.02469	0.02496
Proportion_straightline_arcs	0.04740	**0.04278**	**0.04278**	0.04313
Std_reward_penalty_ratio	**0.14331**	0.51419	0.51419	0.51437
Std_node_degree	**0.02101**	0.02457	0.02471	0.02886
Average	**0.07263**	0.08745	0.08746	0.08760

Again, many of the selection methods have similar values, possibly due to the overlap in instances. The ones that are all the same are those have a very limited range of values (such as the average arc adjacency). Surprisingly, the random selection has a lower standard deviation of consecutive differences for many characteristics. It is not immediately obvious why this would be the case, however it is possibly related to the discrete nature of the problem instance generation settings.

7.10.5 Coefficient of variation of consecutive differences

Table 7.7 shows the *coefficient of variation of consecutive differences* for each characteristic and selection method. The best (lowest) CVCD for each characteristic is highlighted.

Table 7.7: Coefficient of variation of consecutive differences by characteristic for the giant problem

Characteristics	Random	Method 1	Method 2	Method 3
Arcs	**4.21597**	4.47977	4.47977	4.47977
Avg_arc_adjacency	**9.94976**	**9.94976**	**9.94976**	**9.94976**
Avg_depot_shortest_path	5.66718	1.65262	1.65864	**1.65214**
Avg_node_degree	**12.20649**	**12.20649**	**12.20649**	**12.20649**
Avg_shortest_path	3.13926	3.03004	3.03029	**2.95417**
Budget_to_cost_ratio	**3.72575**	3.73897	3.73901	3.72716
CV_node_incidence_reward	4.05434	1.93238	1.93285	**1.92355**
CV_penalty	2.85133	2.20987	2.20801	**2.20417**
CV_reward	2.52066	**1.48073**	1.48283	1.51128
Density	**3.20277**	3.39598	3.39141	3.36377
Depot_degree	**9.94976**	**9.94976**	**9.94976**	**9.94976**
Depot_rows_from_side	5.38488	**4.69828**	**4.69828**	**4.69828**
Mean_reward_penalty_ratio	9.04021	4.46797	**4.46796**	4.46872
Nodes	**1.49501**	2.09995	2.09995	2.04711
Proportion_pendant_arcs	**1.63349**	1.72567	1.72859	1.74735
Proportion_straightline_arcs	2.88487	**2.11099**	**2.11099**	2.12818
Std_reward_penalty_ratio	5.95759	**4.97805**	4.97810	4.97981
Std_node_degree	**1.64521**	1.72004	1.73013	1.98035
Grand Total	4.97358	**4.21263**	4.21349	4.22066

There are no obvious patterns here, except to note that the random selection has a lower CV for many characteristics than the selection methods, due to the standard deviation measure.

7.10.6 Maximum consecutive difference

Table 7.8 shows the *maximum consecutive difference* for each characteristic and selection method. The best (lowest) value for each characteristic is highlighted.

Table 7.8: Maximum consecutive difference by characteristic for the giant problem

Characteristics	Random	Method 1	Method 2	Method 3
Arcs	**0.20695**	0.39798	0.39798	0.39798
Avg_arc_adjacency	**1.13242**	**1.13242**	**1.13242**	**1.13242**
Avg_depot_shortest_path	2.24667	**0.51810**	**0.51810**	**0.51810**
Avg_node_degree	**1.84755**	**1.84755**	**1.84755**	**1.84755**
Avg_shortest_path	**0.61867**	1.00926	1.00926	0.97636
Budget_to_cost_ratio	0.19617	**0.19515**	**0.19515**	**0.19515**
CV_node_incidence_reward	1.40347	**0.74225**	**0.74225**	**0.74225**
CV_penalty	**0.71582**	0.76569	0.76569	0.76569
CV_reward	0.36813	**0.23695**	**0.23695**	0.25875
Density	**0.22390**	**0.22390**	**0.22390**	**0.22390**
Depot_degree	**0.95473**	**0.95473**	**0.95473**	**0.95473**
Depot_rows_from_side	**0.46174**	**0.46174**	**0.46174**	**0.46174**
Mean_reward_penalty_ratio	1.85237	**1.48891**	**1.48891**	**1.48891**
Nodes	**0.10421**	0.20842	0.20842	0.20842
Proportion_pendant_arcs	0.19588	**0.15339**	**0.15339**	**0.15339**
Proportion_straightline_arcs	0.41380	**0.38354**	**0.38354**	**0.38354**
Std_reward_penalty_ratio	**2.35709**	7.71205	7.71205	7.71205
Std_node_degree	0.21333	**0.20436**	**0.20436**	0.32387
Average	**0.86183**	1.03536	1.03536	1.04138

These are the same for many characteristics, which makes sense for the same reasons as the AvgCD. There are some dramatic exceptions, for example the AVG_DEPOT_SHORTEST_PATH for the selection methods is more than four times that of the random sample, and the STD_REWARD_PENALTY_RATIO is three times larger. The significance of this not clear.

7.10.7 Discussion

The characteristic-based metrics were not used as part of the selection. We can make several inferences from the above results.

It seems that the *total distance* metric used in the selection heuristic does not guarantee that the characteristics will be much more finely grained and evenly spread than a random selection of instances. Some measures were better, others were worse; the TAD was clearly better for the heuristics rather than the random selection, which makes sense since this measure is linked to the total distance. This confirms our initial findings from the thought experiment in section 7.8.6, where all three scenarios had the same total distance, and (b) and (c) even had the same total individual distance for all point, but they had quite different characteristics distributions.

The consecutive difference measures all suffered from the flaw that once the available discrete values have been satisfied, the remainder of instances simply contribute a zero consecutive difference, making the aggregate measures very similar for all the subsets. Even where there are not fixed discrete values that the consecutive differences can take, in a well-distributed subset the differences will all be very small, so the aggregate measures are working on a very small range.

With hind-sight, there is also a question of whether there was sufficient diversity even in the set of 150,000 problem instances. Although the problem instances generated here were much more "finely grained" than previous generation attempts, they potentially still suffer from the inherent shortcomings of parameterized problem instance generation. In the next section we develop a completely new approach which may have more promise to meet the goal of producing "interesting" problem instances.

7.11 Using MLS to design problem instances

This section takes a completely different approach to creating a set of interesting problem instances.

The previous approach in this chapter was to generate a large number of instances, and then select the most diverse; the motivation being that a diverse set of instances would be the best way to detect features that predict relative heuristic performance, by running the heuristics on the instances, and then focusing on those that seem interesting (if any).

An alternative strategy is to actively *force* the instances to be "interesting", rather than hoping that some will be after the fact. We propose a new combinatorial problem, and use a Modular Local Search heuristic to solve it. This both attacks our goal in a new way and demonstrates the flexibility of the MLS framework to be applied easily to a new problem.

For our purposes, interesting problem instances are those that result in quite different relative performances between heuristics. There are an extremely large number of possible problem instances in "problem space" – our goal here is to find those that show the greatest difference in heuristic performance; those that some heuristics perform well on and that others perform poorly on. We could attempt to do this by hand, however we have another tool at our disposal which is more suited for this type of task. Local search heuristics are designed to find good regions of a search space, and the Modular Local Search framework is a flexible way to design and implement local search heuristics.

A brief investigation into this problem was performed, more as a proof-of-concept than to study it intensively.

7.11.1 The Problem Instance Creation Problem

We define the **ASRP Problem Instance Creation Problem** (APICP). The objective of this problem is to create a problem instance that makes a nominated MLS heuristic *A* perform better than another MLS heuristic *B*.

At the risk of using confusing terminology, a **problem instance** for the APICP consists of the two heuristics *A* and *B*. A **solution** to the APICP is an ASRP problem instance. If we let $z(A)$ and $z(B)$ be

the reward collected from the execution of heuristics A and B, respectively, then the **objective function** to be maximized is:

$$Z(A,B) = z(A) - z(B)$$

7.11.2 MLS modules and configuration

Conceptually, the modules required to support this problem are very simple. Almost all of the structure of the MLS algorithm stays the same. Exactly the same MLS heuristics as for the ASRP can be used, with a very few modifications.

Programmatically, there were almost no changes required to the MLS program to solve the APICP. The main changes were that a new **Solution** class was written and new move-types and admissibility conditions needed to be written. These are classes that need to be specifically written for each new problem domain.

A problem-specific MLS solution class is a subclass of the generic Solution class, and it has to implement one method, `getObjective`, which returns the value of the objective function. For the ASRP, a *solution* consists of an ordered sequence of nodes and arcs; these ordered lists need to be stored in the Solution class, and the `getObjective` method goes through these arcs and adds up the reward. For the APICP, the solution subclass is slightly more complicated. Instead of a list of nodes and arcs, it contains an ASRP instance. The `getObjective` method performs quite a complicated operation. It generates a completely new ASRP version of MLS, based on heuristic *A*, and evaluates it. The final best solution resulting from this is stored as *rewardA*. It then repeats this whole process for heuristic *B* to produce *rewardB*. The objective function returned is *rewardA – rewardB*. In order to accomplish this quite strict object-oriented programming was required, but it illustrates the power of the MLS framework that this is possible – a "solution" can be anything at all, as long as there is a way to evaluate its objective function.

Recall that a solution for the APICP is an ASRP instance, consisting of a graph of arcs of unit length, distributed on a nominal grid, reward values for each of those arcs, a maximum cost budget, and a depot node. There are many possible **move-types** that could be considered for this problem. The following list gives some examples:

- Add an arc to the graph (from the underlying grid) – *small change*
- Delete an arc from the graph (must not disconnect the graph) – *small change*
- Add an arc (from the underlying grid) and delete an arc (must not disconnect the graph) – *small change*
- Delete an arc from the graph (if this disconnects the graph then all arcs no longer connected to the depot are also deleted, effectively deleting a whole section of the graph) – *large change*
- Move the depot in a random direction to an adjacent vertex – *small change*
- Move the depot to another randomly selected vertex – *large change*
- Increase the budget by a small amount – *small change*
- Increase/decrease the budget by a small amount – *small change*
- Randomly select a budget between 20% and 120% of the total graph cost – *large change*

Many others are possible, including compound moves combining multiple elements of the above.

Admissibility for APICP solutions is simply whether the graph is connected.

7.11.3 A proof-of-concept trial

To prove the validity of using MLS to design problem instances with desired characteristics we implement a small example. We select two heuristics from the experiments in Chapter 6 and attempt to "design" problem instances that result in the greatest performance difference between them.

Two instances of Steepest Ascent (Algorithm 6.17) were chosen: StpAscBasic and StpAscExt12. These heuristics were selected for this experiment because the analysis in Section 6.4.3 suggested that their relative performance was dependent on problem structure. They were also among the heuristics with the fastest running times; in this problem the examination of every neighbour involves the application of both heuristics, so a quicker running time is desirable.

For a solution to the APICP (which is an ASRP problem instance), let the objective function Z be the difference between the reward collected by the two heuristics:

$$Z = r_{StpAscBasic} - r_{StpAscExt12}$$

Recall that these two heuristics were executed on all 1440 problem instances from Chapter 6, and StpAscBasic outperformed StpAscExt12, in aggregate. To limit the degrees of freedom for this experiment, we fix the budget at 0.75 of the total number of arcs in the graph. Figure 7.6 gives the distribution of Z for the 360 problem instances that had this budget ratio. There 62 problem instances (17%) where Z was negative, i.e., for which StpAscExt12 outperformed StpAscBasic.

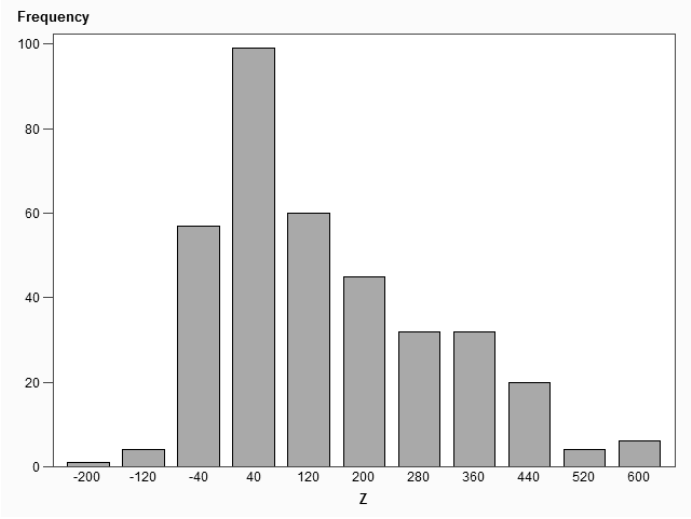


Figure 7.6: Distribution of Z for instances where budget = 0.75 * arcs

As an initial investigation we propose a very simple experiment. We generate a grid with reward randomly distributed from `Uniform`(5, 20) and randomly select 20 connected arcs from this grid. This activity constitutes the construction phase. Then we perform a simple local search that only has one move type, which is adding an arc; the MLS heuristic assembles a graph one arc at a time.

The MLS heuristic we use is a version of Iterative Sampling Local Search (ISLS), which to our knowledge has not been explored in the literature, but which we introduced briefly in Section 4.7.5. ISLS is suited as a simple heuristic where the neighbourhood is either very large or very expensive to evaluate. Our simple variation has the following properties:

- Only one move-type: *add arc*
- The only admissibility condition is that the resulting graph be connected, so moves that worsen the objective function are possible.
- The move-list size is set to *unlimited*. This is not a limiting factor since a 15×15 grid has at most 420 arcs that can be added.
- The candidate list size is set to 10.
- The examinations maximum is set to 50.

This represents a very simple local search routine. By limiting the possible move-types to adding an arc we limit the size of the neighbourhood, but even so examining the full neighbourhood at each iteration is not possible because the evaluation of each neighbour is computationally expensive. Instead we sample from the neighbourhood, examining at most 50 neighbours for admissibility to get a candidate list of 10 solutions, on which both heuristics are performed to calculate the objective function, and the best of these is selected as the target solution.

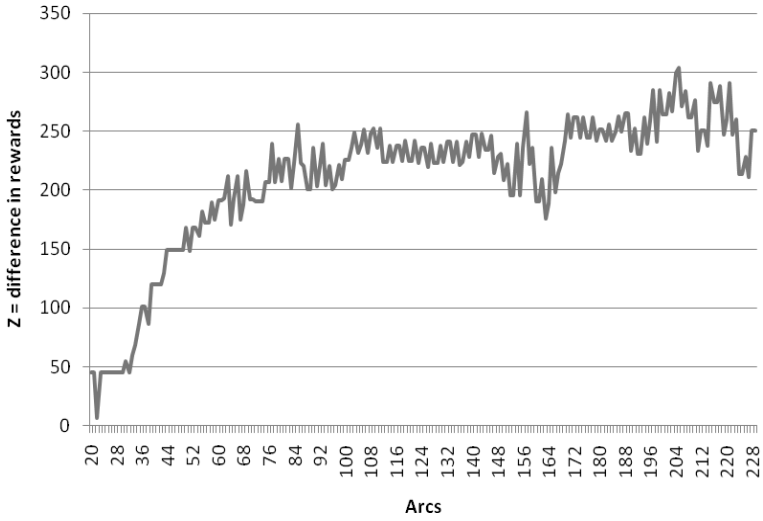


Figure 7.7: Trajectory of Z when adding arcs (StpAscBasic – StpAscExt12)

Figure 7.7 gives the objective function trajectory. At each iteration an arc is added to the graph, and the budget is adjusted; the process was stopped manually after 210 iterations (approximately 8 hours). The objective function on the graph represents the best of the solutions examined, with a maximum difference of 303.38 achieved at 205 arcs. It seems clear from this experiment that it is possible to design problem instances where StpAscBasic outperforms StpAscExt12, but since 83% of the randomly generated instances from Chapter 6 resulted in this outcome, it is not a surprising or impressive result.

Consider now the reverse problem, to find problem instances where StpAscExt12 outperforms StpAscBasic. These were much rarer in the experimental problem set from Chapter 6. The two

approaches, changing the objective to a minimization problem, and redefining Z, are equivalent; for convenience we redefine Z as Z^* and retain the maximization objective:

$$Z^* = r_{StpAscExt12} - r_{StpAscBasic}$$

Figure 7.8 gives the objective function trajectory for Z^*. The heuristic was stopped at the same point as the previous experiment, at 210 arcs. The most obvious feature of this graph is that it was much harder for the MLS heuristic to find ASRP instances that maximized the objective function; the search spent much of the time at zero, which means that the two heuristics found the same solution. However the search was able to find some solutions that satisifed the criteria, with a maximum difference of 44.84589 being found when the number of arcs was 129 and 130.

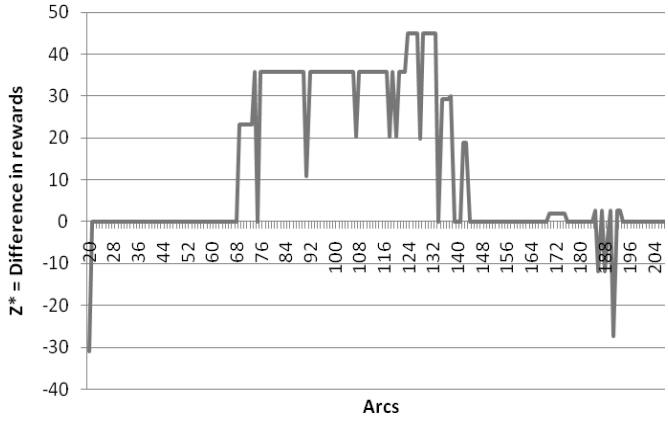


Figure 7.8: Trajectory of Z^* when adding arcs (StpAscExt12 – StpAscBasic)

The neighbourhood for this MLS heuristic was simplistic. Arcs could only be added; arcs added in the early stages of the search when the size of the graph was small could not be removed later.

The next investigation instead focuses on iterative improvement of a graph of a fixed size. Figure 7.9 shows the spread of Z values (the difference between the rewards of StpAscBasic and StpAscExt12) for the 360 problem instances from Chapter 6 that have a budget of 75% of the total cost.

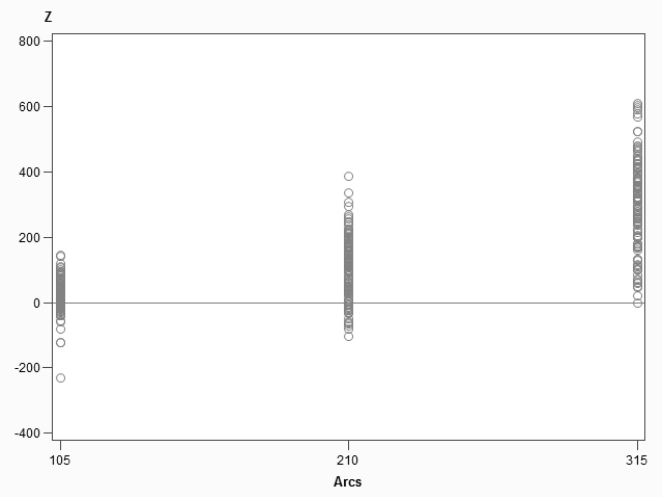


Figure 7.9: Distribution of Z by number of arcs

Of the three density settings only the two lower settings, 105 arcs and 210 arcs, corresponding to 25% and 50% of the complete grid, have a significant number of problem instances where StpAscExt12 outperformed StpAscBasic. Figure 7.10 and Figure 7.11 show the distributions of these differences.

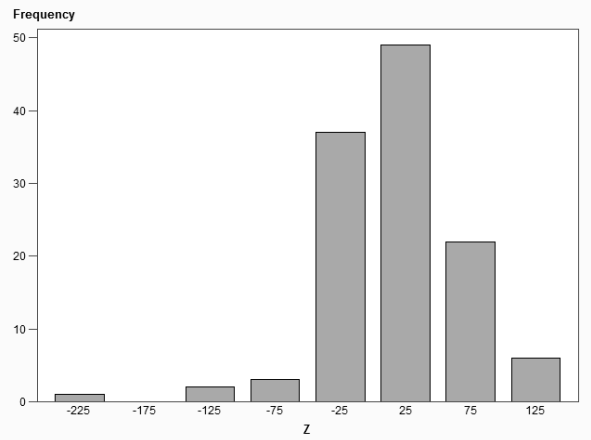


Figure 7.10: Distribution of Z (105 arcs)

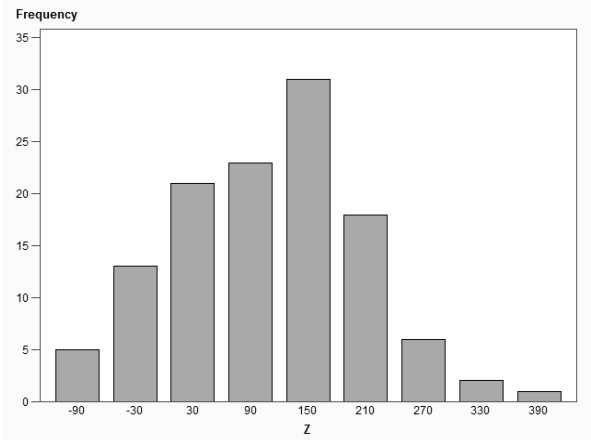


Figure 7.11: Distribution of Z (210 arcs)

For the purposes of this experiment, we fix the number of arcs in the graph at 105 (25% of the complete grid), since this setting has a higher proportion of problem instances where StpAscExt12 outperforms StpAscBasic. We change the move-type from adding an arc to swapping two arcs; so an arc from the complete grid that is not currently included becomes included, and an included arc gets removed, keeping the total number of arcs at 105. The initial set of arcs is selected randomly, until a connected subgraph is obtained.

In the restricted subset of 120 problem instances shown in Figure 7.10 the highest value of Z was 144.64 and the lowest value was -229.30, so these become the targets to beat for the MLS heuristic.

Two variations of the MLS heuristic were used. Both versions are essentially the same as the version used earlier, except with a different move-type. The first variation kept the same candidate list size of 10, and the second variation increased this to 20. The increase was attempted because the maximization

of Z^* did not yield great results; an increased candidate list size means that each iteration takes longer, but should result in better improvements.

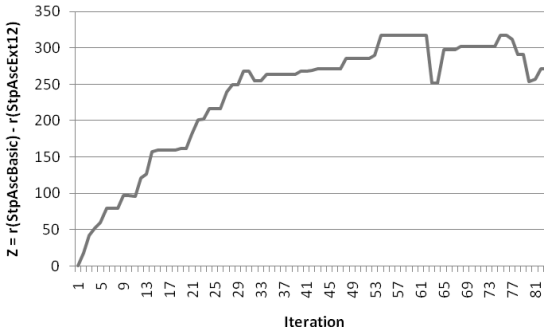


Figure 7.12: Trajectory of Z with 10 candidates

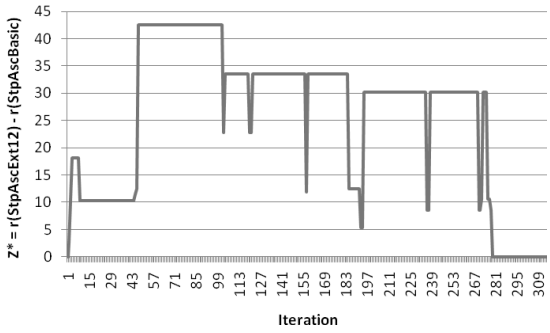


Figure 7.13: Trajectory of Z^* with 10 candidates

Figure 7.12 and Figure 7.13 give the trajectories of the objective function for Z and Z^*, respectively. With Z, the search was stopped after 84 iterations when the upwards momentum had stalled and the best value of 317.36 had clearly exceeded the target of 144.64. With Z^*, the search was stopped after 315 iterations when it had returned to zero. The best value of 42.61 was much less than the target of 229.30.

Figure 7.14: Trajectory of Z with 20 candidates

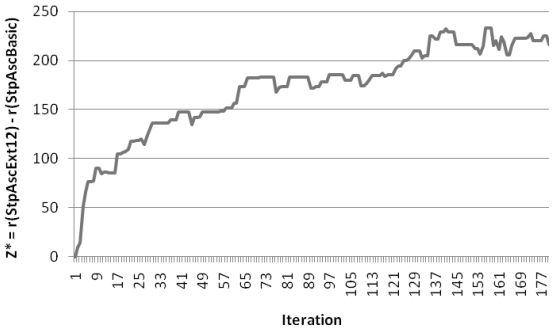


Figure 7.15: Trajectory of Z^* with 20 candidates

Figure 7.14 and Figure 7.15 give the trajectories for Z and Z^* with the candidate list size increased to 20. Both experiments resulted in better search trajectories, dramatically so for Z^*. For Z the search was stopped after 440 iterations, and the best objective function value of 367.56 was achieved after 222 iterations. For Z^* the search was stopped after 180 iterations and the best value of 233.34 was achieved after 157 iterations.

7.11.4 Discussion

It seems clear that the objectives were met, and this validates the approach of using a local search-based design process to design problem instances that meet certain criteria. Our specific example involved designing problem instances that favour one heuristic over another, and vice versa.

There are several ways that seem promising to explore this direction of investigation in future research.

It would be interesting to generate a large number of problem instances that have the desired characteristics of creating a large difference in the performance of two heuristics, and then to analyze the characteristics of those instances to determine if there are any features that high Z or high Z* problem instances have in common.

Another approach would be to plot the changes in characteristics, at each iteration, to see if the techniques are "selecting for" certain characteristics. It would seem probable that there is a high level of noise in this data, so some summary measures could be developed and combined over multiple runs.

It would also be instructive to examine the instances where there is a large change in the objective function from one instance to the next during the search. The slight change that was made to the problem instance had a large effect, so the arcs modified in the change must have been important.

In general, the goal of such research would be to better understand the relationship between problem instance characteristics and heuristic performance, which relates to one of the main research questions of this thesis. The results from the previous chapter indicated that there *is* a systematic relationship, and the design experiments in this chapter further validate that idea. The next step would be to understand *what about* the characteristics influences the heuristics, and how sensitive this is. If successful, this type of investigation would provide a framework for selecting heuristics for *new* problems, based on analysis of their characteristics, and their similarity to previously-studied problems.

Coda

▼ Summary

In this chapter we explored a number of different methods for obtaining "interesting" sets of problem instances.

The main approach was to extend the random generation procedure of previous chapters, and to augment it with attempting to select a *maximally diverse* subset of the generated problem instances. This problem is similar to the class of maximum weighted clique problems that have proven difficult in the literature. We introduce several novel measures of distance and diversity, and develop some heuristics. These heuristics are further extended to some computational approaches to manage extremely large problem sets.

The second approach focused on actually constructing individual instances that exhibited the desired outcomes. We used local search, through the MLS framework, to iteratively modify problem instances to maximize the difference in performance of two heuristics. The approach was very successful; even with quite simple MLS heuristics guiding the search, we were able to exceed the performance gap found in the generated instances from the previous chapter, and we were also able to optimize the other way. This was an introduction of a novel technique that seems to have some promise. It was also a demonstration of the ease with which MLS can be adapted to new problem domains.

▼ Link

In the next chapter we conclude the research with a brief investigation of several "advanced" MLS techniques that could provide promising areas for future research.

CHAPTER 8

Advanced MLS Applications

Some advanced applications of the MLS framework are briefly explored. A multi-level MLS structure is developed to allow one MLS heuristic to guide the development of another, and an adaptive learning mechanism is introduced. Both new concepts are intended as demonstrations, however the results of some brief computational experiments suggest that they are powerful techniques.

8.1 Introduction

The major contribution of this thesis is the introduction of Modular Local Search (MLS), a framework that allows easy and sophisticated hybridization of metaheuristics, especially *multi-phase* hybridization, where the structure and operation of the metaheuristic change significantly during its execution. The previous chapters have presented demonstrations of MLS on the Arc Subset Routing Problem (ASRP), illustrating common metaheuristic paradigms and some basic hybrids of these. However, the true benefits of using a system such as MLS only become apparent when attempting to implement more sophisticated metaheuristic ideas.

This chapter presents two demonstrations of how MLS can be used to develop more sophisticated metaheuristic approaches. Each of these ideas is potentially a very rich area of investigation by itself. We examine each briefly to demonstrate the usefulness of MLS in exploring this type of metaheuristic research. In all cases the actual experiments are performed on the ASRP, and the results are intended to demonstrate the technique rather than provide the most robust realization of the ideas, however we have concentrated on demonstrating ideas that should work for any problem domain, rather than ones which are specific to the ASRP.

8.1.1 Test problems

A small set of problem instances were chosen to test the advanced heuristics in this chapter. Five problem instances were chosen from those generated for the *tiny problem* in Section 7.4. The instances for the tiny problem were generated with three density settings (40%, 60%, 80%), and three budget-to-cost ratio settings (50%, 75%, 100%). The chosen instances had the following properties, as shown in Table 8.1.

Table 8.1: Density and budget characteristics for the test set of problem instances

Instance	Diagram	Arcs	Density	Budget	Budget-to-cost
P1	Figure 8.1	168	40%	84	50%
P2	Figure 8.2	168	40%	126	75%
P3	Figure 8.3	252	60%	126	50%
P4	Figure 8.4	252	60%	189	75%
P5	Figure 8.5	336	80%	336	100%

For illustrative purposes, the graphs for each of these problem instances are displayed in the figures below.

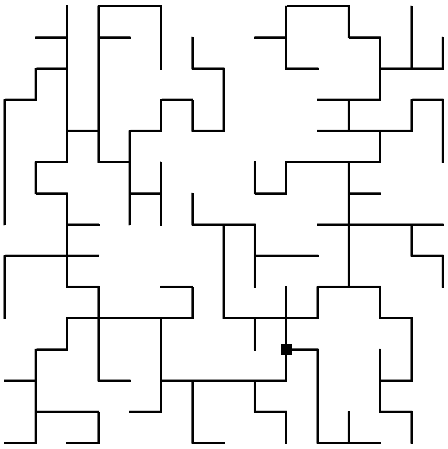


Figure 8.1: Graph for problem instance P1

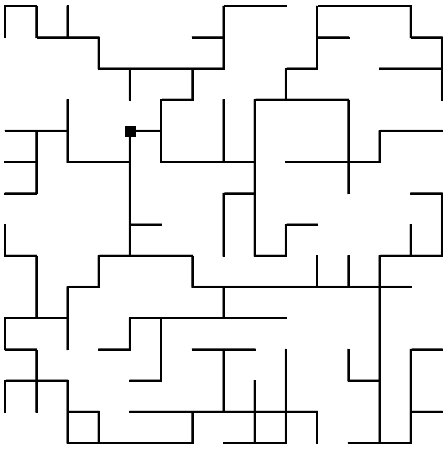


Figure 8.2: Graph for problem instance P2

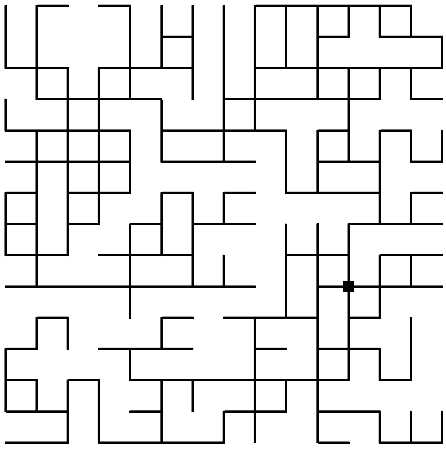


Figure 8.3: Graph for problem instance P3

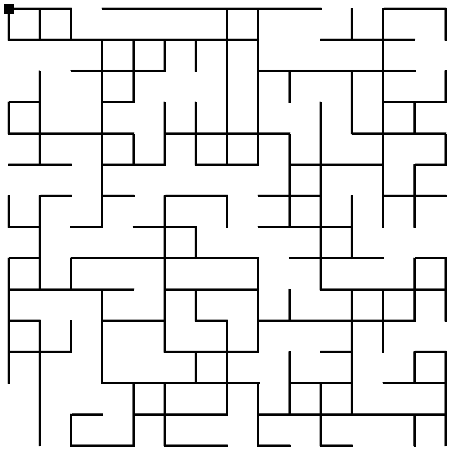


Figure 8.4: Graph for problem instance P4

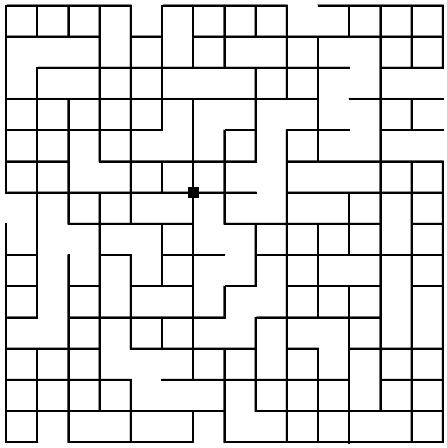


Figure 8.5: Graph for problem instance P5

8.1.2 Comparison heuristics

To provide some benchmarks against which the new procedures can be measured, we apply several of the heuristics from Chapter 6 to the test problem instances. One heuristic was chosen from each of the following types: Steepest Ascent, Simulated Annealing, Tabu Search, and Variable Neighbourhood Search. The selection was made based on which had the best aggregate performance (as shown in Figure 6.1): StpAscExt4, SA1, Tabu20, and VNS8_1. In addition, the construction heuristic that builds the initial solution for all methods, RN1 (see Algorithm 6.16), was included.

Finally, we include a new metaheuristic, Iterative Sampling Local Search (ISLS). ISLS was introduced in Section 4.7.5, and a variation was utilized in Section 0. The MLS configuration for ISLS is presented in Algorithm 8.1.

Algorithm 8.1 MLS configuration ITERATIVE SAMPLING LOCAL SEARCH
Move-types: *basic*
Admissibility conditions:
FEASIBLE (Algorithm 5.1)
Candidate list size: 200
Examinations maximum: 200
Triggers and responses:
Trigger: TOTAL ITERATIONS (Algorithm 6.9) – *active*
Response: TERMINATE (Algorithm 6.11)
Memory parameters:
Iteration threshold for termination: 1000
end

Note that the candidate list size and examinations maximum are both set to 200. This means that up to 200 neighbours are evaluated, and the best of these is chosen at each iteration, even if it worsens the objective function value.

Table 8.2: Reward collected by each heuristic on the test problem instances

Heuristic	P1	P2	P3	P4	P5
RN	243.34	185.19	762.75	480.23	1112.96
ISLS	654.01	1008.72	1294.88	1828.90	3859.00
StpAscExt4	673.69	1006.81	1490.92	2278.17	4212.57
SA1	679.18	***1045.63**	1428.13	2023.50	4231.97
Tabu20	***683.00**	1024.58	1412.66	1928.89	4183.91
VNS8_1	662.35	1014.62	***1512.26**	***2322.16**	***4324.54**

Table 8.2 presents the results of applying the benchmark heuristics on the test problems. In this chapter we treat the objective as maximizing the reward collected subset to the distance budget, and ignore the penalties associated with the arcs of these problem instances. The best-performing heuristic is highlighted for each problem instance.

8.2 Using MLS to design MLS heuristics

The set of possible heuristics is infinite. The use of the MLS framework provides some structure, however even with the limited set of modules and parameters introduced in this thesis, the *heuristic space* is vast; there are more combinations of modules and parameter values than can be fully explored.

It is possible to consider the problem of designing an MLS heuristic, that of selecting the best choice of modules and parameters, to be a combinatorial problem itself. The concept in this section is to use a particular MLS initial configuration as a "solution" that can be acted on by a higher-level MLS heuristic, moving through heuristic space to attempt to find the best MLS configuration for a given problem instance. In this problem, which we call the **MLS Design Problem** (MDP), the objective

function is the value of the ASRP solution obtained by executing the MLS configuration on a given problem instance.

8.2.1 MLS structure

The implementation of the MDP utilizes the existing MLS framework. As with any new problem domain, a new set of problem-specific classes were developed to model the MDP. These primarily consisted of a new **Solution** class, and new **MoveType** and **Move** classes.

Taking advantage of the object-oriented structure of the MLS framework, the new MDP Solution class is mainly a container for an ASRP MLS configuration. The evaluation of the objective function for this solution involves creating a new MLS heuristic and executing this heuristic on the given problem instance, and the best solution obtained from this execution becomes the MDP solution value.

There are many possible move-types that could be considered to move through the MLS heuristic space. They can be broadly classed as adding or removing a module, or increasing or decreasing a parameter. Note that some modules have dependencies, for example if the Annealing Probability admissibility condition is added then the appropriate memory parameters and triggers need to be added also.

We draw a necessary distinction between MDP modules and ASRP modules; the MDP move-types and other modules are those that the high-level control mechanism uses to search heuristic space, the ASRP modules and parameters are those that constitute the MDP solution.

The following sections describe the move-types that were included for this demonstration problem.

8.2.1.1 Adding and removing modules

MDP move-type(s): add (or remove) an admissibility condition. There are four ASRP admissibility conditions considered:

- **All admissible**
  This is a new admissibility condition that allows all solutions to be admissible, even if they are infeasible (cost exceeds the budget). Note that this is the first time we have allowed infeasibility for the ASRP.

- **Improving fitness and feasible OR infeasible but decreasing cost**
  This is a combination admissibility condition. The first part is a union of Algorithm 6.4 and Algorithm 6.3. The second part is a new condition that allows the search to accept solutions that move in the direction of returning to feasibility.

- **Annealing probability and feasible**
  This is a combination of the main Simulated Annealing module (Algorithm 6.19) and the feasibility condition (Algorithm 6.3). In addition, when this module is added (or removed) there are a number of other changes that are made to the MLS configuration:

  - Activate (or deactivate) the SA memory parameters: *annealing temperature* and *cooling rate*.
  - Activate (or deactivate) the SA triggers: *temperature reduction iteration count* and *temperature threshold termination*.

- **Tabu arcs with aspiration and feasible**
  This is a combination of the main Tabu Search module (Algorithm 6.23) and the feasibility condition ((Algorithm 6.3). In addition, when this module is added (or removed) the following change is also made to the MLS configuration:

  - Activate (or deactivate) the TS memory parameter: *tabu tenure*.
  - Add (or remove) the TS memory update function: UPDATE TABU ARCS (Algorithm 6.24).

Note that if all admissibility conditions are removed then the default result is that no neighbours are admissible. This would be a particularly ineffective local search heuristic, and such a configuration is unlikely to be selected as the best neighbour.

MDP move-type(s): add (or remove) an ASRP move-type. There are eight ASRP move-types; the four *basic* move-types (ADD, DROP, SHORTCUT, and DETOUR), and the four *extended* move-types (NADD, NDROP, NSHORTCUT, and NDETOUR). When an extended move-type is added the look-ahead memory parameter is automatically activated, and when the last extended move-type is removed then look-ahead is automatically deactivated. The reason that it is necessary to activate and deactivate the memory parameter, rather than simply leaving it active but unused, is that a deactivated memory parameter is not available to be modified as an MDP move, as we discuss later.

Any combination of ASRP move-types can be active at any time. If no move-types are active then the MLS configuration will not have any neighbours to select; again, this would be a particularly bad heuristic and is unlikely to be selected by the MDP search process.

MDP move-type(s): add (or remove) an ASRP trigger. In general any desired trigger could be added or removed. In our specific case we add two triggers, corresponding to defining a VNS diversification phase that follows local optima in the ASRP MLS heuristic. This particular MDP move-type has the following features:

- The first trigger converts the ASRP heuristic into a phase of VNS diversification (as defined for the VNS heuristics in previous chapters). The *trigger condition* is that an apparent local optimum is reached. This trigger may not be tripped for many iterations, until an apparent local optimum is reached, if ever. Recall that an apparent local optimum is where no admissible candidates were found by the search scheme in the current iteration, so if the admissibility conditions allow all neighbours this trigger may never be tripped. This first trigger is added in an *active* state.

- When tripped the main response is that the set of active move-types is changed. All basic move-types become inactive, and all extended move-types become active, regardless of what their initial states are.

- The secondary responses for the first trigger are to deactivate itself, and to activate the second trigger.

- The second trigger converts the heuristic back into a non-diversification phase. The *trigger condition* is the number of iterations since it was last tripped, with a threshold of one. This trigger has the effect that after a single iteration of the VNS diversification phase, the trigger is tripped and its responses convert the heuristic back. This trigger starts out *inactive* when added, and is specifically activated by the response of the first trigger.

- When tripped the main response is that the set of active move-types is changed, with the opposite "direction" to the first trigger. All extended move-types become inactive, and all basic move-types become active. Note that regardless of what move-types were specified initially for the heuristic, after one round of the VNS diversification phase the move-types are standardized back to basic. There are potentially other ways of modelling this interaction, but this was the most straightforward for the purposes of this demonstration. In addition, the look-ahead parameter for the extended move-types is deactivated.

- The secondary responses for this trigger are to deactivate itself, and the reactivate the first trigger, which will then continue to wait for the next apparent local optimum.

8.2.1.2 Increasing and decreasing memory parameters

The next set of MDP move-types are modifications to memory parameters. For this problem an additional set of attributes were added to the definition of memory parameters. The standard MLSML definition of a memory parameter specifies only the name, its value, and whether it is initially active, as shown in the following MLSML fragment.

```
<memoryParameter>
    <parameterActive>true</parameterActive>
    <parameterName>coolingRate</parameterName>
    <parameterValue>0.75</parameterValue>
</memoryParameter>
```

This parameter is an example of the *cooling rate* for Simulated Annealing heuristics. For the MDP we extend these attributes, as shown below:

```
<memoryParameter>
    <parameterActive>false</parameterActive>
    <parameterName>coolingRate</parameterName>
    <parameterValue>0.75</parameterValue>
    <increaseType>Multiply</increaseType>
    <increaseValue>1.1</increaseValue>
    <minValue>0.1</minValue>
    <maxValue>0.99</maxValue>
</memoryParameter>
```

The additional attributes define how the memory parameter should be modified during an MDP move. There are only two memory parameter move-types for the MDP: *increasing* and *decreasing* the parameter. The additional attributes allow us to specify how these increases and decreases occur.

- The *parameter value* attribute defines the initial value of the parameter when the ASRP heuristic begins.
- The *increase type* may take the values "Add" or "Multiply". If the increase type is "Add" then the *increase value* is added to, or subtracted from, the current memory parameter value. If the *increase type* is "Multiply" then the calculation is slightly more complicated. For an increase the current value is multiplied by the *increase value*, and for a decrease the current value is divided by the *increase value*.
- A *minimum value* and a *maximum value* are also specified, if an increase would push the parameter value higher than the maximum, then it is set to the maximum instead. Likewise, if a decrease would push the parameter value lower than the minimum, then it is set to the minimum instead.

Memory parameters are only available to be increased or decreased by the MDP moves if they are currently active. Memory parameters are inactive if they are not required by any of the currently active modules. For example, the cooling rate above is activated when the annealing probability admissibility condition is added, and deactivated when this module is removed.

For our demonstration, the memory parameters that we allow to be modified are listed below.

- Candidate list size, the maximum number of admissible neighbours that are evaluated;
- Examinations maximum, the total number of neighbours that can be evaluated;
- Look-ahead, the parameter that defines the scope of the extended move-types;
- Annealing temperature, a parameter for the Simulated Annealing modules;
- Cooling rate, a parameter for the Simulated Annealing modules;
- Tabu tenure, a parameter for the Tabu Search modules.

8.2.1.3 Summary of MDP moves

The following lists summarize the moves that are included in the demonstration problem. Recall the distinction between a **move** and a **move-type**. The general move-types for the MDP are: add module, remove module, increase parameter, and decrease parameter. The actual moves are the possible realizations of these on the current problem.

Each of the following *modules* may be either added (if they are not currently included) or removed (if they are currently included):

- Admissibility condition: All admissible
- Admissibility condition: Improving fitness and feasible OR infeasible but decreasing cost
- Admissibility condition: Annealing probability and feasible
- Admissibility condition: Tabu arcs with aspiration and feasible
- Move-type: Add
- Move-type: Drop
- Move-type: Shortcut
- Move-type: Detour
- Move-type: nAdd
- Move-type: nDrop
- Move-type: nShortcut
- Move-type: nDetour
- Triggers: VNS diversification (a pair of triggers)

The following *memory parameters* may be either increased or decreased, if they are currently active:

- Candidate list size
- Examinations maximum
- Look-ahead
- Annealing temperature
- Cooling rate
- Tabu tenure

8.2.2 MLS configurations

This section describes the specific MLS configurations that were used in the experiments for the MDP. Two distinct MLS configurations are defined; one for the MDP control mechanism, and one that forms the template for the ASRP versions. The template heuristic is modified by the MDP procedure, but most of the modules are defined in the configuration and these are simply activated or deactivated by the MDP process.

Two variations of the MDP control heuristic are used, with slightly different settings. The first version, MDP1, is a version of Steepest Ascent; only improving solutions are accepted. This forces each ASRP heuristic selected to be better than the previous one. The second version, MDP2, is more of an ISLS approach, where a subset of neighbours are examined and the best chosen, regardless of whether it is

improving or not. Both versions have limitations on the move-list size and the examinations maximum; each neighbour evaluation is expensive since it involves the full execution of an ASRP heuristic.

8.2.2.1 The MLS template for the ASRP heuristics

Algorithm 6.29 presents the MLS template for the ASRP heuristics. Note the following features:

- All of the potential modules and parameters are specified, but most are declared *inactive* initially. These will be activated or deactivated by the MDP control procedure to create what are equivalent to new heuristics, since an MLS configuration with a module that always stays inactive is equivalent to the same configuration without that module.
- Trigger-3 and Trigger-4 are the triggers associated with the VNS diversification phase.
- The *move-list size* is set to 1000. We do this to limit the run time of each iteration. Similarly, the total number of iterations has been set to 1000 (Trigger-5), again to limit the run time of the heuristic.
- Some memory parameters have been fixed, and others are able to be modified as part of the MDP moves. For those that can be modified there is extra information in the form: ("*Increase type*" *increase value*, *min value*, *max value*).
- The initial configuration is a very primitive heuristic, essentially a random walk; all neighbours are admissible, the first neighbour examined is selected, and the search continues for 1000 iterations then stops.

Algorithm 8.2 MLS configuration ASRP TEMPLATE FOR MDP

Move-types:

basic – *active*

extended – *inactive*

Admissibility conditions:

ALL ADMISSIBLE (Algorithm 8.3) – *active*

IMPROVING FITNESS AND FEASIBLE OR INFEASIBLE BUT DECREASING COST (Algorithm 8.4) – *inactive*

ANNEALING PROBABILITY AND FEASIBLE (Algorithm 8.5) – *inactive*

TABU ARCS WITH ASPIRATION AND FEASIBLE (Algorithm 8.6) – *inactive*

Memory update: UPDATE TABU ARCS (Algorithm 6.24) – *inactive*

Triggers and responses:

Trigger-1: ITERATIONS SINCE LAST TRIGGER (*trigger-1*) (Algorithm 6.8) – *inactive*

Response: REDUCE ANNEALING TEMPERATURE (Algorithm 6.20)

Trigger-2: TEMPERATURE THRESHOLD (Algorithm 6.21) – *inactive*

Response: TERMINATE (Algorithm 6.11)

Trigger-3: LOCAL OPTIMUM (Algorithm 6.7) – *inactive*

Response: SWITCH TO EXTENDED MOVE-TYPES (Algorithm 6.27)

Response: DEACTIVATE TRIGGER (*trigger-3*) (Algorithm 6.12)

Response: ACTIVATE TRIGGER (*trigger-4*) (Algorithm 6.13)

Trigger-4: ITERATIONS SINCE LAST TRIGGER (*trigger-4*) (Algorithm 6.8) – *inactive*

Response: SWITCH TO BASIC MOVE-TYPES (Algorithm 6.26)

Response: DEACTIVATE TRIGGER (*trigger-4*) (Algorithm 6.12)

Response: ACTIVATE TRIGGER (*trigger-3*) (Algorithm 6.13)

Trigger-5: TOTAL ITERATIONS (Algorithm 6.9) – *active*

Response: TERMINATE (Algorithm 6.11)

Memory parameters:

Move-list size: 1000

Iterations in VNS diversification phase: 1

Iterations before temperature reduction: 12

Total iterations before termination: 1000

Temperature threshold: 0.001

Candidate list size: 1 (“Add” 3, min=1, max=1000) – *active*

Examinations maximum: 100 (“Add” 5, min=6, max=1000) – *active*

Annealing temperature: 1000 (“Multiply” 1.2, min=100, max=2500) – *inactive*

Cooling rate: 0.95 (“Multiply” 1.1, min=0.1, max=0.99) – *inactive*

Tabu tenure: 7 (“Add” 1, min=3, max=30) – *inactive*

Lookahead: 4 (“Add” 1, min=2, max=10) – *inactive*

end

A number of new modules are used in the template and need to be defined. These are described below. Algorithm 8.3 is the most fundamental admissibility condition. All solutions are admissible, even if they are infeasible. The opposite of this admissibility condition is having no admissibility conditions at all, since the default outcome is to deny admissibility unless specifically granted.

Algorithm 8.3 MLS admissibility condition ALL ADMISSIBLE

```
    return admissible
end
```

Algorithm 8.4 defines a complex admissibility condition. Neighbours are admissible if they are feasible and improve the fitness function, *or* if they are infeasible but have decreasing cost. Since feasibility/infeasibility is purely a function of the cost of the solution exceeding the budget, a reduction in the cost of a solution is a movement *towards* feasibility.

Algorithm 8.4 MLS admissibility condition IMPROVING FITNESS AND FEASIBLE OR INFEASIBLE BUT DECREASING COST

```
    Scope: ASRP problems
    Input: s, s', B   // The current solution, the trial solution and the cost budget

    if c(s') ≤ B then    // The solution is feasible with respect to the cost budget
        if f(s') > f(s) then
            return admissible
        else
            return inadmissible
        end
    else    // The solution is infeasible
        if c(s') < c(s) then
            return admissible
        else
            return inadmissible
        end
    end
end
```

Algorithm 8.5 is a compound admissibility condition; it is a logical conjunction of Algorithm 6.19 and Algorithm 6.3. A solution is admissible if it satisfies the Simulated Annealing admissibility condition and it is feasible.

Algorithm 8.5 MLS admissibility condition ANNEALING PROBABILITY AND FEASIBLE

Scope: ASRP problems

Prerequisites: A memory parameter must be defined for the annealing temperature

Input: $f(s)$, s', T, B *// The fitness of the current solution, the trial solution, the annealing temperature, and the cost budget*

if $c(s') > B$ **then** *// The trial solution is infeasible with respect to the cost budget*
  return *inadmissible*
else
  $\delta \leftarrow f(s') - f(s)$
  if $\delta > 0$ **then**
    return *admissible*
  else if `Uniform`$(0,1) < e^{-\delta / T}$ **then**
    return *admissible*
  else
    return *inadmissible*
  end
end
end

Algorithm 8.6 is another compound admissibility condition, a logical conjunction of Algorithm 6.23 and Algorithm 6.3. It allows a feasible solution to be admissible if it does not add or remove any arcs on the tabu list, or if a feasible solution satisfies the aspiration criterion of being better than the best-so-far solution.

Algorithm 8.6 MLS admissibility condition TABU ARCS WITH ASPIRATION AND FEASIBLE

Scope: ASRP problems

Prerequisites: The tabu tenure memory parameter, and the tabu list memory element must be defined

Input: s^*, s, s', T, B *// The best-so-far solution, the current solution, the trial solution, the tabu list, and the cost budget*

if $c(s') > B$ **then** *// The trial solution is infeasible with respect to the cost budget*
  return *inadmissible*
else
  $\Delta \leftarrow \{s \cup s'\} \setminus \{ s \cap s'\}$ *// Find all the arcs that have changed from s to s'*
  if $\Delta \cap T = \varnothing$ **then** *// No tabu arcs are changed*
    return *admissible*
  else if $f(s') > f(s^*)$ **then** *// Tabu arcs are changed but the aspiration criterion is met*
    return *admissible*
  else
    return *inadmissible*
  end
end
end

8.2.2.2 MDP control heuristics

Two variations of a MDP control heuristic are presented. Both are relatively simple, and both make use of the MLS features that allow control of the search process by limiting the number of neighbours examined. This is necessary for the MDP because each neighbour that is evaluated requires the full execution of an ASRP metaheuristic, which can become very computationally expensive.

Algorithm 8.7 presents the first MDP control heuristic (MDP1). The key feature that distinguishes MDP1 from MDP2 is that MDP1 is a hill-climbing heuristic; it has the improving admissibility condition, so only solutions that improve the objective function value are permitted. MDP1 is a hybrid of Ascent Search and Steepest Ascent. Ascent Search selects the first admissible neighbour, which can be thought of as choosing the "best of 1" admissible neighbours; Steepest Ascent selects the best of all admissible neighbours, which can be thought of as choosing the "best of all" admissible neighbours. MLS abstracts this concept to allow a hybrid of these: "best of n", where n is the *candidate list size*. For the MDP we set the candidate list size to 3. Recall that there is an implicit maximum of 25 moves that can be performed at any time (13 modules that may either be added or dropped depending on their current status, and 6 memory parameters that may be increased or decreased). The final important feature of MDP1 is that it runs until it reaches a local optimum, where there are no single moves (modules added or removed or memory parameters increased or decreased) that result in an ASRP MLS heuristic that performs better than the current solution. Interestingly this is a true local optimum, rather than merely an apparent local optimum, since all neighbours are examined if no admissible solutions

are found. If the heuristic reaches 80 iterations it also stops. This termination condition is mostly present as a "safety switch" to ensure that the heuristic does not run longer than MDP2, which *does* run until the full 80 iterations have elapsed; it would be surprising for MDP1 to reach 80 iterations without terminating due to a local optimum.

Algorithm 8.7 MLS configuration MDP CONTROL HEURISTIC 1
- **Move-types:**
  - Add module
  - Remove module
  - Increase parameter
  - Decrease parameter
- **Admissibility conditions:**
  - IMPROVING (Algorithm 6.4)
- **Candidate list size:** 3
- **Triggers and responses:**
  - **Trigger-1:** LOCAL OPTIMUM (Algorithm 6.7) – *active*
    - **Response:** TERMINATE (Algorithm 6.11)
  - **Trigger-2:** TOTAL ITERATIONS (Algorithm 6.9) – *active*
    - **Response:** TERMINATE (Algorithm 6.11)
- **Memory parameters:**
  - Iteration threshold for termination: 80

end

Note that even for a completely different problem domain than the ASRP many of the MLS modules are reusable, such as the IMPROVING admissibility condition, and all the triggers and responses.

Algorithm 8.8 presents the second MDP control heuristic (MDP2). The main difference from MDP1 is that all moves are admissible, so the search never reaches an apparent local optimum; it selects the best neighbour of the first five that are examined (randomly). The candidate list size is increased slightly, but is still kept relatively low. Because only five neighbours are examined, it is likely that none of these would be improving, so we would expect MDP2 to have a very "jagged" objective function trajectory. MDP2 is a much more diversifying search process than MDP1. The heuristic is allowed to run for 80 iterations, which in trials allowed the process to be competitive with the benchmark heuristics (full results are presented in Section 8.2.3).

Algorithm 8.8 MLS configuration MDP CONTROL HEURISTIC 2
Move-types:
Add module
Remove module
Increase parameter
Decrease parameter
Admissibility conditions:
ALL ADMISSIBLE (Algorithm 8.3)
Candidate list size: 5
Triggers and responses:
Trigger: TOTAL ITERATIONS (Algorithm 6.9) – *active*
Response: TERMINATE (Algorithm 6.11)
Memory parameters:
Iteration threshold for termination: 80
end

8.2.3 Results

The MLS Design Problem (MDP) has the objective of designing a good MLS heuristic for the ASRP, and the objective function value is the objective function value of the best ASRP solution found by the various ASRP heuristics that are created and modified by the MDP control procedures. Because of this we are able to compare the performance of the MDP heuristics and the regular ASRP heuristics that are described in Section 8.1.2.

Table 8.3: Reward collected by the MDP heuristics and benchmark heuristics on the test problems

Heuristic	P1	P2	P3	P4	P5
RN	243.34	185.19	762.75	480.23	1112.96
ISLS	654.01	1008.72	1294.88	1828.90	3859.00
StpAscExt4	673.69	1006.81	1490.92	2278.17	4212.57
SA1	679.18	1045.63	1428.13	2023.50	4231.97
Tabu20	683.00	1024.58	1412.66	1928.89	4183.91
VNS8_1	662.35	1014.62	1512.26	***2322.16**	***4324.54**
MDP1	***685.70**	***1085.63**	1542.07	1881.06	3834.23
MDP2	***685.70**	1061.72	***1629.85**	2175.30	4140.61

Table 8.3 presents the results for the MDP heuristics, along with the results of the benchmark heuristics. The MDP heuristics achieve the best solution on three of the problem instances, the three "smallest" problem instances.

It is interesting to speculate on why the MDP heuristics are so successful on the smaller instances, and not so successful on the largest instance. Consider the building blocks of the ASRP heuristics that the MDP heuristics have to work with. Although many of the parameters are able to take a much broader range of values than their "regular" counterparts, others are deliberately restricted to streamline the

search; for example, the move-list size is restricted to 1000 moves, which is a small fraction of those available, especially with the extended move-types. On smaller instances, these restrictions have less of an effect. Another insight into this difference is that the 80 iteration limit may be more of a limit on the larger problem instances where there are more potential ASRP routes.

The other interesting factor when considering the performance of the MDP heuristics is the objective function trajectory; how the objective function changes at each iteration over the execution of the heuristic. In the following figures the shorter black line is MDP1, which terminates at a local optimum, and the longer grey line is MDP2, which terminates after 80 iterations.

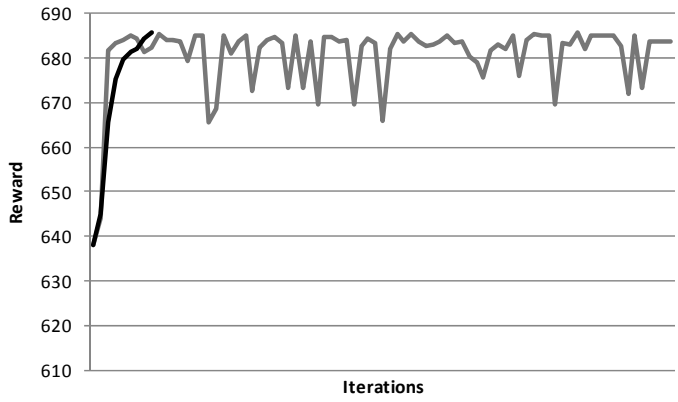


Figure 8.6: Objective function trajectories for MDP1 and MDP2 on P1

Figure 8.6 shows the trajectories for P1. MDP1 reaches its best of 685.70 at iteration 8 and MDP2 reaches the same best after 67 iterations.

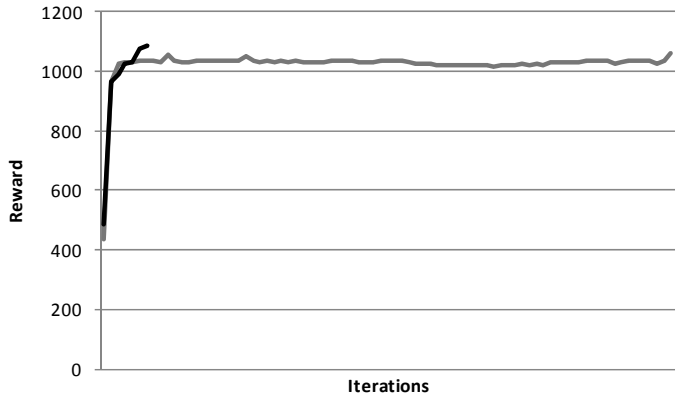


Figure 8.7: Objective function trajectories for MDP1 and MDP2 on P2

Figure 8.7 shows the trajectories for P2. MDP1 reaches its best of 1085.63 at iteration 6 and MDP2 reaches its best of 1061.72 at iteration 80. It seems possible that MDP2 could have climbed higher if the execution had been extended, since it achieved its best on the final iteration. It is interesting that on P2 the hill-climbing variation (MDP1) very quickly arrived at a good solution.

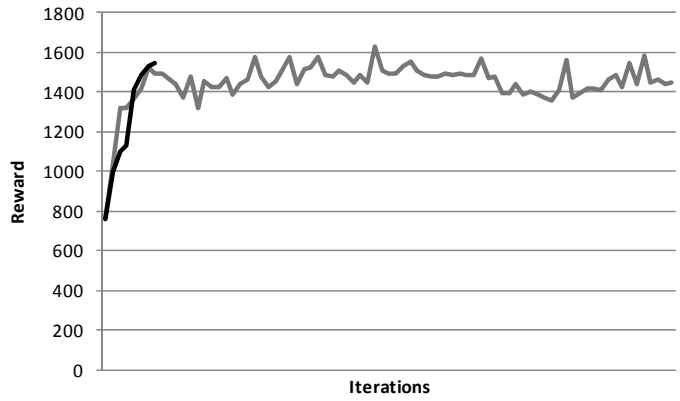


Figure 8.8: Objective function trajectories for MDP1 and MDP2 on P3

Figure 8.8 shows the trajectories for P3. MDP1 reaches its best of 1542.07 at iteration 7 and MDP2 reaches its best of 1629.85 at iteration 38.

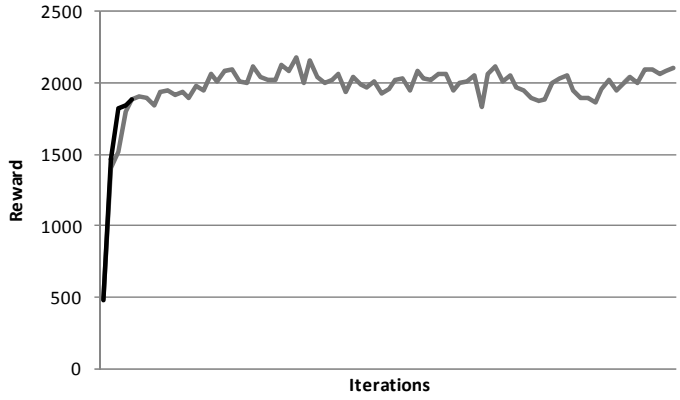


Figure 8.9: Objective function trajectories for MDP1 and MDP2 on P4

Figure 8.9 shows the trajectories for P4. MDP1 reaches its best of 1881.06 at iteration 4 and MDP2 reaches its best of 2175.30 at iteration 27. On this problem instance MDP2 shows a clear improving trend until it reaches its best, and then declines as it moves into less promising regions of the solution space.

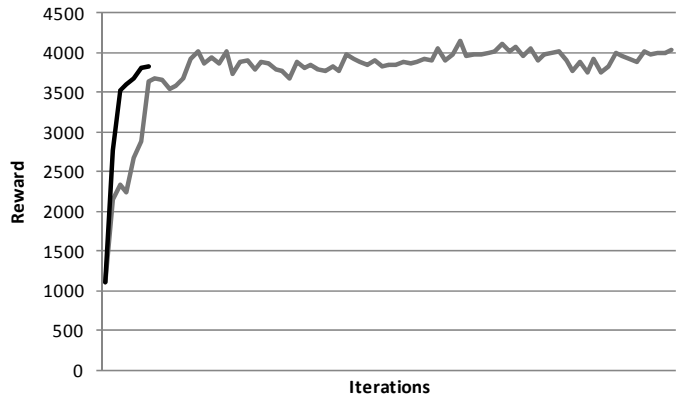


Figure 8.10: Objective function trajectories for MDP1 and MDP2 on P5

Figure 8.10 shows the trajectories for P5. MDP1 reaches its best of 3834.23 at iteration 6 and MDP2 reaches its best of 4140.61 at iteration 50. One observation is that MDP1 reached a good solution on P5 very quickly, whereas on P1 MDP1 had a slower climb. This is perhaps related to the size or density of the graph, or is perhaps just a random variation.

Table 8.4: MDP move frequencies for MDP2

Move-type	Move detail	P1	P2	P3	P4	P5	Total	Percent
Add module	All admissible	2	2	1	3	2	10	2.50%
	Annealing probability		2	1	3	1	7	1.75%
	Compound admissibility			2	1	2	5	1.25%
	Tabu arcs with aspiration	1	2	3		2	8	2.00%
	VNS triggers	2	1	1	1	1	6	1.50%
Remove module	Add move-type	1	1	1	1	1	5	1.25%
	All admissible	3	2	1	3	2	11	2.75%
	Annealing probability		2	1	2	1	6	1.50%
	Detour move-type	1	1	1	1	1	5	1.25%
	Drop move-type	1	1	1	1	1	5	1.25%
	Compound admissibility			1		2	3	0.75%
	Shortcut move-type	1	1	1	1	1	5	1.25%
	Tabu arcs with aspiration		1	3		1	5	1.25%
	VNS triggers	1					1	0.25%
Increase parameter	Candidate list size	10	9	13	14	18	64	16.00%
	Annealing temperature		2	7	6	1	16	4.00%
	Cooling rate		1	3	6		10	2.50%
	Examinations maximum	11	14	8	6	12	51	12.75%
	Lookahead	1			1		2	0.50%
	Tabu tenure	12	7	5		4	28	7.00%
Decrease parameter	Candidate list size	9	6	5	8	10	38	9.50%
	Annealing temperature		1	2	10		13	3.25%
	Cooling rate		3	4	5		12	3.00%
	Examinations maximum	9	14	13	7	11	54	13.50%
	Tabu tenure	15	7	2		6	30	7.50%

Because there are only 32 different moves available to the MDP heuristics, we are able to count the number of times that each move was performed. Table 8.4 summarizes the move frequencies for MDP2 (since MDP1 terminated after only a few iterations it did not provide enough data to provide insight). Note that the "compound admissibility" rows in the table refer to the IMPROVING FITNESS AND FEASIBLE OR INFEASIBLE BUT DECREASING COST admissibility condition. Several observations can be made from these data:

- All of the possible moves were executed at least once, although not always on every problem instance. The frequencies are relatively sparse, especially for the change-of-module moves; running the heuristics for more than 80 iterations could provide more reliable proportions.

- A large majority of the moves are of the change-parameter move-types, rather than the change-module move-types, even though we would expect the change-module move-types to be evaluated more frequently since the selection is random and there are more of them. The

proportions for each move-type are: add module (9%), remove module (11.5%), increase parameter (42.75%), and decrease parameter (36.75%).

- Collectively, 51.75% of the moves related to the *candidate list size* and *examinations maximum* parameters. This implies that these parameter settings have a large impact on the effectiveness of an ASRP heuristic, especially since they are both increased *and* decreased (if they were only increased it would imply that they were simply too low). This is an interesting result, since these two parameters reflect the focus of MLS on limiting the way that the search is conducted – an approach that is relatively unexplored in the literature.

8.2.4 Discussion

We introduced the MLS Design Problem (MDP), which uses high-level MLS heuristics to guide the design of "regular" MLS heuristics for the ASRP, by adding and removing modules, and increasing and decreasing parameters.

Although the experiments were intended primarily to demonstrate the concept, they were surprisingly successful. On a test set of 5 problem instances of various sizes they managed to find the best solution on three of the instances; their strong performance validates the potential of this technique.

The comparable success of MDP1, which quickly converged to its local optimum heuristic suggests that the design approach need not be an exhaustive computational exercise; good heuristics are able to be designed relatively quickly. It is interesting that MDP1 reached a local optimum by iteration 8 for all the test problem instances. When combined with the very "jagged" appearance of the MDP2 trajectories, this implies that a more sophisticated local search procedure guiding the search it seems likely that even better heuristics could be constructed, since avoiding and escaping from local optima is the motivation and strength of modern metaheuristics such as Tabu Search and Simulated Annealing.

The MDP also highlights one of the main strengths of the MLS framework; since all the components are modular and may be mixed and matched, it is possible for a heuristic to modify itself, or, in this case, to modify another heuristic. This section has barely hinted at the potential of this approach, which could provide an extremely rich area of future research.

A number of extensions immediately suggest themselves. The focus of this approach was to design a stand-alone MLS heuristic for the ASRP, although we then used the value of the best ASRP solution, found individually by these heuristics, as the objective function for the MDP. If we alter this focus slightly away from designing a stand-alone heuristic to actually solving the ASRP in the most effective way, then the MDP procedure becomes a two-layer MLS process. Instead of each ASRP heuristic starting from the same initial solution, an elite solution from the previous iteration could feed into the next heuristic. This would be analogous to Iterated Local Search, but with perturbation of the *heuristic* rather than perturbation of the *solution*. Such a two-layer approach, with a base layer that finds the solutions to the main problem, and a control layer that guides the modification of the base layer, becomes a framework for true self-adaptive heuristics. This idea can be extended to even more layers, with another layer controlling the modification of the second level control layer. It would be an interesting investigation to research the benefits and limitations of extra layers. One limitation is that it

is likely to become very computationally expensive; the aspects of the MLS framework that allow the search logic to be controlled will likely be crucial, as we saw in a slight way in our demonstration of the MDP: the *move-list size*, the *candidate list size*, and the *examinations maximum*.

Another interesting modification would be to give each ASRP heuristic an actual time-limit at which it is terminated, unless it has already terminated for another reason. Careful attention would need to be paid to ensuring consistent processing power, but this approach could prompt some interesting insights. It would force trade-offs between neighbourhood size and complexity and number of iterations per unit of time, and the MLS search logic parameters would be crucial.

8.3 Adaptive Diversification Local Search

Section 8.2 demonstrated how a two-layer MLS procedure can be used to guide the modification structure of an MLS heuristic, with the example of the MLS Design Problem. In that example the structure of the heuristic is made to evolve using local-search-based principles. In this section we introduce another concept: learning.

An **adaptive** MLS heuristic is one that is able to draw directly from memory structures that are updated as the heuristic progresses, to inform changes to the MLS structure based on past experience. There are many possible ways that learning could be implemented in the MLS framework; in this section we demonstrate one example.

We introduce **Adaptive Diversification Local Search** (ADLS). ADLS is a multi-phase metaheuristic that has a relatively straightforward local search heuristic as the "main phase", and has a number of different diversification phases available. When an apparent local optimum is reached by the main phase of the heuristic, a short diversification phase follows. The type of diversification phase is selected probabilistically from a weighted list of methods. When a diversification phase is judged to be successful the weighting of that diversification method is increased, and when it is unsuccessful the weighting is decreased. The probability of selecting each type of diversification phase depends on its weighting, so that successful methods become more likely to be selected.

Note that the MLS implementation of ADLS is closer to the "regular" metaheuristics utilized throughout the thesis than the two-layer MDP approach of Section 8.2; it is not a new problem domain and requires no new classes. A "solution" is simply an ASRP route, and ADLS is simply another metaheuristic for the ASRP (although it is has the potential for general application). The only change is the development of some additional *responses* and *memory structures*.

The purpose of introducing ADLS is to demonstrate a simple example of using MLS to implement self-adaptive metaheuristics that are capable of learning. Far more sophisticated applications are possible, but are left for future research.

There are some parallels in this technique to Ant Colony Systems, population-based metaheuristics that are inspired by the foraging behavior of ants, where fruitful paths are given more "pheromone" and paths with more pheromone are more likely to be selected in the future (see Dorigo and Blum [73] for a survey of the theory of Ant Colony Optimization).

8.3.1 MLS structure

The **main phase** of the ADLS heuristic is quite simple. The *basic* set of ASRP move-types are used, along with the IMPROVING and FEASIBLE admissibility conditions. The *candidate list size* is set to 100, with an *examinations maximum* of 100,000 (effectively unlimited). In addition, the UPDATE TABU ARCS memory update function is active, even when the Tabu Search diversification phase is not being executed. The purpose of this is that the list of tabu arcs is maintained so that when the Tabu Search diversification phase starts it already has a pre-populated list of tabu arcs to work on.

For the purposes of this demonstration we define six diversification procedures, each of which is active for three iterations before the heuristic changes back to the main phase:

- **Small VNS**. A VNS-type phase where the move-types are changed from the *basic* set to the *extended* set. In addition the lookahead for the extended move-types is set to 6, and the *candidate list size* is increased to 500. The intention of this phase is to allow a relatively intense search of a broader region of the search space than the basic moves allow.

- **Large VNS**. A VNS-type phase similar to the Small VNS phase, except that the lookahead is set very high, at 50, and the *candidate list size* is decreased to 20. The intention of this phase is to allow relatively extreme moves to be performed to change the current region of the solution space in a major way. The low candidate list size has two purposes; firstly to make the computation quick, and secondly to force potentially worsening moves in order to increase the diversification effect.

- **Tabu acceptance**. This diversification phase is modelled on Tabu Search. The list of tabu arcs is automatically updated even during the main phase, so this phase simply replaces the IMPROVING admissibility condition with the TABU ARCS WITH ASPIRATION admissibility condition. In addition the *candidate list size* is reduced to 50. The tabu tenure is fixed at 15.

- **Probabilistic acceptance**. This diversification phase is modelled on Simulated Annealing. The *temperature* memory parameter is fixed at 500, and there is no need for a *temperature threshold* or *cooling rate*. This phase replaces the IMPROVING admissibility condition with the ANNEALING PROBABILITY admissible condition. The *candidate list size* is reduced to 3.

- **Sampling**. This diversification phase removes the IMPROVING admissibility condition, leaving only the FEASIBLE admissibility condition, so non-improving moves are allowed. The *candidate list size* is decreased to 30.

- **Change fitness**. This diversification phase changes the fitness function from the objective function to REWARD TO COST RATIO (a new fitness function defined below in Algorithm 4.1). Note that this is the only example in this thesis of a change in the fitness function. Guided Local Search (GLS) is the most common example of a change in fitness function, although GLS adds penalty terms rather than changing the function completely. For our purposes the temporary change in fitness function should allow the search to escape the apparent local optimum.

The key MLS components that are necessary to support ADLS are two responses, one update-memory function, and two memory structures.

The first memory structure is a list that contains the names of the six diversification procedures that the heuristic should probabilistically select from, along with a weighting assigned to that procedure. The weightings are all initially set to zero, so are not specified.

The two responses are used to change the structure of the heuristic from the main phase to the selected diversification phase, and then back again.

The first response, START DIVERSIFICATION PHASE (Algorithm 6.12), is a relatively complex procedure. It is a compound response that performs multiple tasks; in previous MLS heuristics these tasks would have been specified as separate responses, for example activating a trigger and swapping move-types. This response combines multiple activities into the single response, since the tasks to be performed depends on the diversification phase being entered; the MLS framework allows the same process to be performed in multiple ways, depending on the requirements of the particular process.

The first activity in the START DIVERSIFICATION PHASE response function is the selection of the diversification phase. This procedure uses a memory parameter, which we call the **diversification alpha**. The probability of selecting diversification method x_i is given in the following formula:

$$P(x_i) = \frac{\max(0, w_i + \alpha)}{\sum_{j=1}^{N} \max(0, w_j + \alpha)}$$

where there are N diversification methods, w_i is the weight of diversification method x_i, and α is the diversification alpha, which must be positive. The formula essentially calculates the proportion of the total weight that each diversification method has. Initially, when all weights are zero, $P(x_i) = 1 / N$. A higher diversification alpha serves to discount the effect of a weighting difference.

For example, with six diversification methods the probability of selecting any particular method is initially $1 / 6 = 0.167$. Let $\alpha = 1$ and suppose that the first diversification method is successful (we define success later). In this case the weighting for that diversification method is incremented to 1. In the next diversification phase the probability of selecting that method has increased to 0.286 and the probability of selecting any other method has decreased to 0.143. This is a large movement in the probabilities. If we let $\alpha = 10$, then the probabilities change to 0.180 and 0.164, respectively, which is a less dramatic change. For the purposes of this demonstration we set the diversification alpha to 6.

The max functions in the formula are present to ensure that no negative probabilities are calculated. This has the effect that if w_i decreases by α, so that $w_i + \alpha = 0$, then the probability of selecting x_i becomes zero.

The remainder of the START DIVERSIFICATION PHASE response is a conditional function that makes the appropriate modifications depending on which diversification method was selected, such as swapping admissibility conditions and changing the *candidate list size*. Algorithm 8.9 presents the MLS response module logic.

Algorithm 8.9 MLS response START DIVERSIFICATION PHASE

Input: $\underline{x}$, $\underline{w}$, α *// The vectors of diversification methods and their weights, and the diversification alpha*

Select a diversification method x with probability $P(x_i) = \dfrac{\max(0, w_i + \alpha)}{\sum_{j=1}^{N} \max(0, w_j + \alpha)}$

if x = "Small VNS" **then**
  Deactivate the *basic* move-types
  Activate the *extended* move-types
  Set the lookahead memory parameter to 6
  Set the *candidate list size* to 500
else if x = "Large VNS" **then**
  Deactivate the *basic* move-types
  Activate the *extended* move-types
  Set the lookahead memory parameter to 50
  Set the *candidate list size* to 20
else if x = "Tabu acceptance" **then**
  Deactivate the *improving* admissibility condition
  Activate the *tabu arcs with aspiration* admissibility condition
  Set the *candidate list size* to 50
else if x = "Probabilistic acceptance" **then**
  Deactivate the *improving* admissibility condition
  Activate the *annealing probability* admissibility condition
  Set the *candidate list size* to 3
else if x = "Sampling" **then**
  Deactivate the *improving* admissibility condition
  Set the *candidate list size* to 30
else if x = "Change fitness" **then**
  Deactivate the *objective* fitness function
  Activate the *reward to cost ratio* fitness function
end
end

The second response, END DIVERSIFICATION PHASE (Algorithm 8.10), is another conditional function that changes everything back to their initial values for the next main phase.

Algorithm 8.10 MLS response END DIVERSIFICATION PHASE

Input: *x* *// The diversification phase just ending*

if *x* = "Small VNS" **then**
Deactivate the *extended* move-types
Activate the *basic* move-types
Set the *candidate list size* to 100
else if *x* = "Large VNS" **then**
Deactivate the *extended* move-types
Activate the *basic* move-types
Set the *candidate list size* to 100
else if *x* = "Tabu acceptance" **then**
Deactivate the *tabu arcs with aspiration* admissibility condition
Activate the *improving* admissibility condition
Set the *candidate list size* to 100
else if *x* = "Probabilistic acceptance" **then**
Deactivate the *annealing probability* admissibility condition
Activate the *improving* admissibility condition
Set the *candidate list size* to 100
else if *x* = "Sampling" **then**
Activate the *improving* admissibility condition
Set the *candidate list size* to 100
else if *x* = "Change fitness" **then**
Deactivate the *reward to cost ratio* fitness function
Activate the *objective* fitness function
end
end

The ADLS heuristic also has an *update-memory* module that determines whether the diversification phase was a success, and either increments or decrements the weight associated with that diversification method appropriately. The update-memory function, UPDATE DIVERSIFICATION WEIGHTS, is *inactive*, so is not executed automatically; it is called explicitly when an apparent local optimum is reached. This function has no effect after the first main phase, but after subsequent main phases it updates the diversification weights according to the following rules:

- The current objective function value (an apparent local optimum) is stored in memory. This is used in the next execution of this module to compare with the new current value.
- The objective function at the end of the previous main phase is compared with the current objective function after this just-ending main phase. If there has been an improvement, then this is attributed to the diversification phase that occurred immediately prior, and its weight is incremented. If there was no improvement then its weight is decremented.

- If all weights have been reduced by α, then no diversification method can be selected, so all weights are reset to zero.

Algorithm 8.11 presents the update-memory function that updates the diversification weights. This function, along with the memory list containing the weights themselves, constitutes the mechanism by which ADLS incorporates *learning*.

Algorithm 8.11 MLS update-memory UPDATE DIVERSIFICATION WEIGHTS

Input: z^*, s', x, $\underline{w}$ *// The previous objective value (from the last time this function was executed), the target solution, the diversification method that executed prior to this main phase, and the list of diversification weights.*

If x != *null* **then** *// Check that a diversification phase has occurred*
  if $z(s') > z^*$ **then** *// If the diversification phase was a success*
    $w_x \leftarrow w_x + 1$
  else
    $w_x \leftarrow w_x - 1$
  end
end

if $w_i = -\alpha \ \forall \ i$ **then** *// If the weights have all reached their limits then reset*
  $w_i \leftarrow 0 \ \forall \ i$
end

$z^* \leftarrow z(s')$ *// Update* z^*

end

8.3.2 MLS configuration

Algorithm 8.12 presents the MLS configuration for the version of ADLS introduced in this chapter.

Algorithm 8.12 MLS configuration ADAPTIVE DIVERSIFICATION LOCAL SEARCH

Move-types:

basic – *active*

extended – *inactive*

Fitness functions:

OBJECTIVE (Algorithm 6.5) – *active*

REWARD TO COST RATIO (Algorithm 8.13) – *inactive*

Admissibility conditions:

FEASIBLE (Algorithm 6.3) – *active*

IMPROVING (Algorithm 6.4) – *active*

ANNEALING PROBABILITY (Algorithm 6.19) – *inactive*

TABU ARCS WITH ASPIRATION (Algorithm 6.23) – *inactive*

Memory update:

UPDATE TABU ARCS (Algorithm 6.24) – *active*

UPDATE DIVERSIFICATION WEIGHTS (Algorithm 8.11) – *inactive*

Triggers and responses:

Trigger-1: LOCAL OPTIMUM (Algorithm 6.7) – *active*

Response: UPDATE DIVERSIFICATION WEIGHTS (Algorithm 8.11)

Response: START DIVERSIFICATION PHASE (Algorithm 8.9)

Response: DEACTIVATE TRIGGER (*trigger-1*) (Algorithm 6.12)

Response: ACTIVATE TRIGGER (*trigger-2*) (Algorithm 6.13)

Trigger-2: ITERATIONS SINCE LAST TRIGGER (*trigger-1*) (Algorithm 6.8) – *inactive*

Response: END DIVERSIFICATION PHASE (Algorithm 8.10)

Response: DEACTIVATE TRIGGER (*trigger-2*) (Algorithm 6.11)

Response: ACTIVATE TRIGGER (*trigger-1*) (Algorithm 6.13)

Trigger-3: TOTAL ITERATIONS (Algorithm 6.9) – *active*

Response: TERMINATE (Algorithm 6.11)

Memory parameters:

Iteration threshold for diversification phase: 3

Total iterations before termination: 1000

Candidate list size: 100

Annealing temperature: 500

Tabu tenure: 15

Lookahead: 4

Diversification methods: {*Small VNS, Large VNS, Tabu Acceptance, Probabilistic Acceptance, Sampling, Change fitness*}

Diversification alpha: 6

end

Algorithm 8.13 is a fitness function that calculates the reward to cost ratio of the solution. Note that this is the first alternative fitness function introduced in this thesis.

Algorithm 8.13 MLS fitness function REWARD TO COST RATIO
Input: *s* *// The solution being evaluated*

$f(s) \leftarrow r(s) / c(s)$ // The fitness is the reward divided by the cost
return $f(s)$
end

8.3.3 Results

The results of executing ADLS on the test problems are presented in Table 8.5, along with the results from the MDP heuristics and the benchmark heuristics, for comparison.

Table 8.5: Reward collected by the ADLS heuristic, compared with other heuristics on the test problems

Heuristic	P1	P2	P3	P4	P5	Total
RN	243.34	185.19	762.75	480.23	1112.96	2784.47
ISLS	654.01	1008.72	1294.88	1828.90	3859.00	8645.51
StpAscExt4	673.69	1006.81	1490.92	2278.17	4212.57	9662.16
SA1	679.18	1045.63	1428.13	2023.50	4231.97	9408.41
Tabu20	683.00	1024.58	1412.66	1928.89	4183.91	9233.04
VNS8_1	662.35	1014.62	1512.26	2322.16	4324.54	9835.93
MDP1	***685.70**	***1085.63**	1542.07	1881.06	3834.23	9028.69
MDP2	***685.70**	1061.72	***1629.85**	2175.30	4140.61	9693.18
ADLS	682.99	1071.30	1569.46	***2378.83**	***4327.56**	***10030.14**

The results are extremely encouraging. The new heuristic achieved a new best result on the two largest problem instances, and high values for the other problem instances. If the total reward over all five problem instances is calculated, then ADLS has the best performance, exceeding the previous best heuristic VNS8_1. Our objective of demonstrating the effectiveness of an advanced MLS structure seems a success.

Given that the implementation of ADLS was a very basic "first try", with no attempt to optimize the many parameter settings, these results seem to suggest that a learning-based approach such as ADLS has potential. An interesting question would be whether a non-learning-based version that simply selects the diversification method randomly with no updating of weights would perform as well; it is possible that the effectiveness comes from the range of diversification procedures rather than the learning aspect.

Coda

▼ Summary

Two novel metaheuristic approaches were introduced in this chapter, and both show promising potential. On the five test problem instances, the best solution for all instances was found by one of the advanced heuristics, outperforming the best of the "regular" heuristics from Chapter 6.

Either of the new approaches would provide ample opportunity for future research. The first concept, which we called the MLS Design Problem (MDP), introduced a higher-level control mechanism to modify the ASRP heuristic, treating the design of the modular heuristic as a combinatorial problem that itself can be tackled using local search. This implementation also demonstrated the application of the MLS framework to another problem domain, highlighting its generality and flexibility.

The second concept introduced a basic version of learning – utilizing the memory structures of MLS, along with the flexible trigger-response model, to develop a sophisticated self-adaptive metaheuristic that demonstrated learning. This experiment barely touched on what could prove to be an extremely promising new field of research.

▼ Link

In the next chapter we discuss a number of research directions that could be explored to extend the concepts introduced in this thesis.

CHAPTER 9

Conclusions and Recommendations for Future Research

9.1 Overview of research

In Chapter 1 we stated two main research questions that this research would attempt to address:

1. Is it possible to predict the relative performance of heuristics on a problem instance based on analysis of the problem instance prior to running the heuristics?

2. Can we develop a modular metaheuristic system, that encapsulates most trajectory-based local search methods, that allows easy hybridization of metaheuristics, and is not simply a programming convenience but also supports the creation of *new* types of metaheuristics?

The concept of modelling the relationship between heuristic performance and problem instance characteristics is a novel one, and lead the research down many paths. One of the key requirements is a problem type that is suitable, one for which many metrics can be calculated to suitably specify the characteristics of each problem instance, and one which is sufficiently *difficult* so as to allow for heuristics to distinguish themselves. Part I of the thesis introduces a new problem to fill this role, the Arc Subset Routing Problem (ASRP). The ASRP is introduced after a literature review of other arc routing problems. Although its primary purpose is as a testing ground for the MLS heuristics and problem-heuristic modelling done subsequently, it is first investigated as a traditional Operations Research problem: it is formulated, some construction heuristics are developed, along with some local search-based improvement procedures, and some tournaments are performed comparing the heuristics. In this "preliminary investigation", some metrics are derived for the ASRP, and these are built upon in later chapters. Also, the move types for the local search improvement procedures that are developed here form the basis of all the local search metaheuristics used in later chapters. This investigation also

serves as a pilot study for the more extensive experimentation later; it was decided that the 10×10 grids used here should be extended to 15×15 grids. The ASRP was shown to be sufficiently easy to create metrics for, and sufficiently interesting to be used as the problem-space for the remainder of the research.

Part II of the thesis addresses the second research question above. We propose and introduce a novel framework, Modular Local Search (MLS), which is designed to allow easy hybridization of various existing metaheuristic concepts. Chapter 4 is a description of each of the components of MLS, and how they work together. Although it is presented as finished concept, the final specification was the result of many iterations of design and experiment until it met the research goals. The MLS framework is the major contribution of this research, and much of the rest of the thesis is devoted to demonstrating its utility: that it can be used to easily implement existing heuristics, that it can be used to create new heuristics that are hybrids of existing concepts, that it is flexible and can be applied to new problem domains, and that it is not simply a programming convenience, but can be used to create *new types* of metaheuristics.

As discussed in Chapter 1, other metaheuristic frameworks tend to be simple programming structures designed to implement existing metaheuristics. For example, they might have a Tabu Search class and a distinct Simulated Annealing class. By contrast, MLS is a *conceptual* framework first, and a programming structure second.

Local search heuristics can be thought of as a sequence of iterations of starting with one solution and attempting to find the next solution. This process, the *search iteration process*, utilizes components that all local search heuristics have in common, either implicitly or explicitly: the neighbourhood scheme, the fitness function, the admissibility conditions, and the search logic. Most basic metaheuristics keep these same components throughout their execution, and so each iteration uses the same rules and process to choose the next solution in the path.

The key insight that motivates MLS is that these components do not need to be the same for each iteration. MLS introduces the trigger-response model, a novel mechanism for modifying a heuristic during its execution. After each iteration, any triggers that have been defined are checked, and if they are "tripped", then the associated responses are performed. These responses typically modify the heuristic in some way, for example by swapping admissibility conditions. This is a technique that allows a heuristic to completely change itself partway through execution in response to search conditions. When combined with adaptive memory structures, this is a very flexible system, easily allowing new heuristics that were not previously possible, simply by *declaring* the desired behaviour. Examples of these new types of heuristics are demonstrated in Chapter 8.

Part III of the thesis is devoted to exploring these new tools, bringing the ASRP together with MLS. Chapter 6 is an attempt to address the first research question above, whether it is possible to predict the relative performance of heuristics based on à priori analysis of the problem instance characteristics, which we refer to as problem-heuristic modelling. To the best of our knowledge this has not been previously studied, but the potential utility of such a procedure is obvious; if researchers or practitioners could know in advance which solution methods are most likely to result in good solutions then this

could result in time and cost savings, and better solutions. Our aim was simply to determine whether or not such an approach was possible, rather than to fully develop a methodology.

A large number of experiments were performed, running a number of metaheuristics on a large set of problem instances. The experimental design is critiqued in a later section, so here the focus is more on the results and what we can conclude from these. Since this type of analysis is novel, we experimented with various analytical techniques, with some success. The general approach was to choose two heuristics, A and B, and then form a balanced set of problem instances where heuristic A performed best on half the instances and heuristic B performed best on the other half. Balancing the dataset like this prevents the model from simply using the prior probabilities to choose one heuristic as the "winner" over the other. The set of instances is then split into training and evaluation sets; the model is trained on one set and tested on the other. The only variables available to the predictive model are the problem instance characteristics that are obtained before any heuristics have been run. Our success criterion is whether the model is able to predict which heuristic has better performance with better accuracy than we would expect from choosing randomly. The results were positive and encouraging. The experiments and analysis in Chapter 6 showed clear evidence that it is, indeed, possible to predict relative performance of two heuristics using a model trained on previous applications of the heuristic to problem instances, with the problem instance characteristics as input to the model.

The other function of Chapter 6 was to demonstrate the MLS framework in use. Versions of Tabu Search, Simulated Annealing, and Variable Neighbourhood Search were specified and run. In addition, a number of *hybrids* were created and demonstrated, to show that new heuristics can be easily created, especially multi-phase heuristics, which are typically difficult to implement.

Although the experiments of Chapter 6 did produce a positive outcome validating the problem-heuristic modelling concept, it was observed that the problem instances were somewhat clustered together, based on their problem characteristic metrics. In retrospect, generating 30 instances with each generation configuration was not the best way to get a good problem set; it would be better if the instances were as evenly distributed across a conceptual problem space as possible. This is the focus of Chapter 7. The goal was to build on the results of Chapter 6, and produce a set of problem instances that was more suitable for the problem-heuristic modelling. Another driver was that the set of 1440 problem instances was too large to reasonably experiment with, so the secondary goal was to derive a smaller set of problem instances, but which were more suitable.

Chapter 7 developed some procedures to derive better problem instances, for input into the problem-heuristic modelling. The concept is novel, so the research was exploratory. The first investigation utilized a concept of *diversity* of problem instances. A number of diversity metrics were derived to describe the diversity of a set of problem instances. The general idea was to generate an extremely large number of random problem instances, since generation is a cheap process, and then select a *maximally diverse* subset of these. An optimization problem was formulated, and some heuristics were developed to solve this problem. Encouragingly, multiple selection heuristics seemed to result in subsets that had substantial overlap, suggesting that these heuristics were close to optimality, at least with respect to the diversity measure used. To make the problem space even richer for this exercise, the ASRP was

extended to include penalties for non-service. Time constraints prevented these problem instances being used for the problem-heuristic modelling, but a potential research plan is outlined in Chapter 7.

The approach up until this point had been to have a set of problem instances, run two heuristics on these, create a predictive model on which heuristic would perform better based on the problem instance characteristics, and then look at the model to see what about the problem instances is important in making the distinction. The second part of Chapter 7 develops a completely different approach to problem generation, and to the problem-heuristic modelling. Instead, we *construct* a problem instance to exhibit the property we desire, namely a large performance difference between heuristic A and heuristic B. This section develops a completely novel idea. A new combinatorial optimization problem is defined, where a solution is an ASRP problem instance, and the objective function is the performance difference between heuristic A and heuristic B. Then, MLS is used to solve this problem. Although only simple move-types were used, adding or removing an arc from the underlying grid, the results were very strong that this technique works. Problem instances were designed that strongly favoured heuristic A over heuristic B, and then the reverse, favouring B over A. This avenue of research was not pursued further, but there is significant potential with this technique. It also highlighted the utility and flexibility of the MLS framework. Very little modification of the MLS program was required to solve this problem, which was effectively a completely new problem domain. All of the metaheuristic control logic was simply reused from the ASRP experiments, and only a wrapper-class was required to hold an ASRP problem instance as a solution, and a move-type to add or drop an arc.

Chapter 8 moves away from the problem-heuristic modelling concept and explores some of the more advanced heuristic ideas that MLS allows. Our research goal was to develop a modular framework that was not simply a programming convenience, but which allowed the creation of *new* types of metaheuristic; this chapter illustrates some of these new types of heuristics. In some sense, the rest of the thesis is *foundational* work, leading up to this chapter; these techniques justify MLS. The "advanced" concepts were explored briefly, establishing their validity and potential, leaving many areas of further research open.

The first "advanced" MLS technique uses MLS to *design* other MLS heuristics. This is another new problem domain: the MLS Design Problem (MDP). Because the MLS structure is modular, new metaheuristics can be created simply by adding and removing modules, and changing parameter values. This process of changing modules and parameters can be thought of as a search in heuristic space, and metaheuristics are very efficient and effective search procedures. A simple version of this problem was run, using performance on an ASRP problem instance as the objective function. Five test ASRP problem instances were used, and the MDP-created heuristic outperformed the best results from our other existing metaheuristics on some of the problem instances. This result was surprising, given that the first MDP configuration attempted should not be expected to be the best, but still achieved such good results. This type of technique is has significant promise to allow new types of heuristic development. In the normal course of research, a heuristic can only be as good as the researcher who created it, and has no guarantee to be optimized for a given problem domain. With the MDP approach, metaheuristics are *self-adaptive*, becoming iteratively more tailored to the problem at hand.

The second advanced MLS technique, which we call Adaptive Diversification Local Search (ADLS) uses the powerful memory structures in MLS to implement *learning*. There are many ways that learning and memory could be incorporated into an MLS heuristic; for this investigation we designed a multi-phase metaheuristic that procedes with basic local search until it reaches a local optimum, and then executes a diversification phase. This repeats until a total iteration count is reached. The unique feature of ADSL is that at each diversification phase the diversification method is chosen randomly from a list, with probability related to that methods previous success. After each diversification phase the probabilities are updated based on how well it worked. On the same five problem instances as used previously, the ADLS heuristic outperformed all other heuristics on two of the instances, and best in aggregate.

The two new advanced metaheuristics, that are only made possible by MLS, between them found the best solution on all of the five test problems, outperforming the best versions of the other metaheuristics. Five instances is not enough to draw any conclusions about these heuristics, but it is certainly encouraging. These new metaheuristics were also only the first version of each that was attempted, with basic settings and arbitrary parameters; it seems likely that these could be improved further. Chapter 8 demonstrates that MLS allows the creation of *new* types of metaheuristics, which answers one of the research questions for this thesis.

9.2 Experimental design

Upon critical reflection, there were some significant shortcomings with the experimental design for the computational experiments throughout the research. There were three distinct phases of tournaments, where heuristics were applied to problem instances.

The first set of tournaments was in Chapter 3, for the preliminary ASRP investigation. The first purpose of these tournaments was to compare the various ASRP construction heuristics that had been designed, both by themselves and with simple improvement procedures. The second purpose was to trial the ASRP as a computational test problem, analysing simple random instance generation methods, and creating the initial set of problem instance characteristic metrics. The tournaments followed a typical Operations Research investigation pattern, where every heuristic was applied to every instance, and the instances were generated using a full factorial design of the various generation settings. The heuristics were deterministic, so no replications were necessary. The experimental design seems fit for its intended purposes.

The next set of tournaments was in Chapter 6, and the experimental design here had some flaws. The first problem was with the scale of experimentation. The goal was to produce a large dataset, with many heuristics run on many problem instances, to enable modelling the relative performances of heuristics based on problem characteristics. Since this had not been done before, the methodology was itself an exercise in exploration. As it turns out, the volume of experiments was too ambitious, and needed to be abandoned without completing them all after several months of computer processing on multiple machines. Although enough tournaments were completed to enable successful analysis, at least to validate the idea of modelling heuristic performance and problem characteristics, it would have been better to have a more focussed experimental design. The approach was simply to generate a large

number of data points, and then do the analysis, whereas better results would have been achieved by performing a sequence of smaller experimental schedules, each one incorporating the learnings from the previous.

The second problem with the Chapter 6 experiments concerned the lack of replications for the heuristics that involved a random component. The Simulated Annealing configurations of the MLS heuristics involve sampling from a random distribution, and so the result, even applying the same heuristic to the same problem instance will be different each time it is run. Conventional experimental design dictates that *at least* 30-50 replications should be performed, and the mean and variance of these be considered. In retrospect, this should absolutely have occurred in our experimental design, although this would have significantly increased the total number of tournaments. To understand why this was not performed requires a subtle shift in perspective. The goal of the experiments in this chapter was to attempt to model the relationship between a heuristic and the various problem characteristic metrics. In total there were 48 different problem generation "settings" (see Section 6.2.4), representing problem instances with similar characteristics. 30 problem instances were generated at each setting, so the idea was that each heuristic would already be performed with 30 replications at each generation setting, and replications on the individual problem instances would be unnecessary. This reasoning was flawed, and proper experimental design dictates that the replications should have been performed. Indeed, one way that the total volume of experiments could have been reduced was by generation a single problem instance at each generation setting, and then performing multiple replications for the Simulated Annealing-based heuristics.

The only mitigating factor is that the main purpose of Chapter 6 was not to objectively judge the heuristics against each other and determine the best, in which case the experimental design was ineffective, but rather to produce a large dataset for modelling, and **this was achieved**, although haphazardly. The experimentation in Chapter 6 was very much an exploratory, learning, exercise, and the key insight to be gained is that massive experimental schedules should be avoided. A series of smaller experiments that built on each other, with more focus, is preferable.

The final set of tournaments was for Chapter 8, for the advanced MLS applications. The goal of this chapter was to demonstrate some of the more interesting potential uses of MLS. Time constraints dictated that these investigations were performed as a "proof-of-concept", rather than in any depth, and the small scale of the experiments highlights this. Five problem instances were used to demonstrate the new metaheuristic concepts. These five instances were sufficient for demonstration purposes, but it should be noted that this is not a sufficiently comprehensive experimental design to properly evaluate or compare the heuristics; at least 100 instances would be preferable, and the best, mean and variance of the performances reported.

Overall, the experimental designs achieved the goals of the various experimental phases, however they did not do so efficiently or elegantly.

9.3 Contributions and implications

There are many original contributions to the field expressed in this thesis. Indeed, perhaps the strongest criticism is that it attempted to introduce *too many* new ideas, at the expense of in-depth study of any

one of them. However, as described in the previous section, each of these ideas do form a part of a cohesive whole, which forms the foundations for significant new ways of studying combinatorial optimization problems. In this section we summarise the contributions, and sketch briefly how these might be developed further.

9.3.1 The MLS framework

The most significant contribution is the Modular Local Search framework itself. This framework is significant in several ways.

The most obvious advantage that it conveys is the **ease with which metaheuristics can be implemented**. Only the bare minimum of programming needs to be performed to implement new heuristics, for example programming a new admissibility condition. The remainder of the metaheuristic structure and control algorithm stays the same and is reused. Indeed, many new metaheuristics can be expressed as MLS with no programming required at all; the Modular Local Search Markup Language (MLSML) allows heuristics to be specified *declaratively*, rather than *programmatically*. This was demonstrated clearly by the ease with which standard metaheuristics such as Tabu Search, Simulated Annealing, and Variable Neighbourhood Search were implemented, and by how quite sophisticated multi-phase hybrids were created from the component modules of these metaheuristics.

This ease of creating hybrids is made possible by a **new conceptual model** of local search-based metaheuristics. Most prior research treats each type of metaheuristic as a distinct category, and this leads to most other frameworks implementing each metaheuristic as a separate entity. MLS, by contrast, treats all local search-based metaheuristics as simply versions of MLS that have a greater or lesser number of modules in common. Each of the components that all local search techniques have in common are abstracted out and made explicit: the neighbourhood scheme, the admissibility conditions, the neighbourhood reduction process, the search logic, triggers and responses, etc. MLS also makes some aspects of the search logic explicit that are usually not considered, allowing fine grained control of the search process: the move-list size, the examinations maximum, and the candidate list size. Most metaheuristics typically examine the whole neighbourhood at each iteration; MLS allows a subset of the neighbourhood to be examined, and this is another means of controlling the intensity of the search, making tradeoffs on computation time and number of iterations.

The flexibility of MLS is also demonstrated by how easily it is **adapted to new problem domains**. In the course of the thesis it was applied to three quite distinct problems: the ASRP, the ASRP Problem Instance Creation Problem, and the MLS Design Problem. In all these cases only a few supporting classes needed to be programmed, representing a "solution", and the move-types, and all of the control logic was unchanged. MLS can reduce the time and effort needed to implement metaheuristics on new problems, allowing the researcher to focus on experimentation, and on designing new metaheuristic innovations.

The most important contribution of MLS is that it allows **completely new types of advanced metaheuristics**. This was demonstrated in Chapter 8, with an adaptive MLS heuristic that used *learning* to progress through multiple phases and adapt itself to suit the problem instance at hand. Chapter 8 also demonstrated what was arguably the part of the thesis with the most significant implications: using one

MLS heuristic as a control mechanism to *design* other metaheuristics to suit a particular problem instance. The potential of this type of technique is discussed in more depth later.

9.3.1.1 Limitations and tradeoffs of MLS

Although we believe that MLS is a very useful tool for combinatorial optimization research, it is not necessarily the right tool for every purpose; there are some limitation and tradeoffs.

The main point of concern is whether an MLS version of a metaheuristic causes a significant performance degradation over a metaheuristic programmed and optimized individually. Obviously the best way to determine this would be to actually test it using some state-of-the art metaheuristics on standard benchmark problems. However, we are able to reason about this topic and draw some tentative conclusions.

The underlying assumption of this concern is that a framework such as MLS is *necessarily* slower than a specially programmed algorithm. This is not true. First, we make the observation that two different researchers implementing the same algorithm are likely to make quite different programming choices, which will correspond to their skill level and past experience. We would argue that a framework such as MLS, which has been carefully tuned is perhaps likely to run more efficiently overall than an algorithm which is programmed from scratch and suffers from a "first version" set of bugs and inefficiencies. There is a question of whether MLS performs many more operations than a specially programmed algorithm, and again the answer is no. MLS has many more things that it *can do*, but these are not tasks that are performed unless they are required by the algorithm. The actual path through the program takes *exactly the same steps* that would be required for the specially-programmed version. For example, consider a trigger that checks to see if a certain number of iterations have passed. In MLS this is a trigger that is only checked if it is currently active, in a regular program it might be part of a while loop, but in both cases the number of iterations is compared with some threshold after each iteration. This is the same set of tasks.

There are some performance savings that could be made by performing extreme optimization on the code; reducing the object-oriented nature of it, using fixed arrays rather than ArrayList objects, perhaps attempting to use a language such as C, rather than Java. However, these savings would likely be minimal, high-level programming languages such as Java are extremely efficiently optimized in the current era. Also, the level of effort required to program, and then debug such a low-level algorithm would far exceed the benefits. Algorithms that utilize linear algebra and matrix decompositions, such as the Simplex algorithm, have to be programmed in languages such as C to achieve every possible milli-second of computational time-savings, since they perform millions of iterations. Algorithms such as metaheuristics are far more dependent on the specification of the algorithm, regardless of how it is programmed.

And this is where MLS far exceeds individually-programmed heuristics. It is not only much quicker to tweak the heuristics and experiment with different settings, but it enables methods that are definitely not possible with individual heuristics - such as one MLS routine designing another.

There are other limitations of the MLS framework, such as the types of metaheuristics that can be expressed with it; these were discussed in Section 4.8.4.

9.3.2 Problem-heuristic modelling

Chapter 6 validated the hypothesis that it is possible to predict relative heuristic performance based on à priori analysis of problem instance characteristics, at least on problem types for which solution data had previously been collected.

This type of analysis is the first step in what could prove to be a fruitful field of study. Even the prototypical models developed in Chapter 6 could be of significant practical benefit; using the models to choose a heuristic to run would result in better aggregate performance over time than a random heuristic choice, or simply using one of the heuristics always. In case studies from the literature, one or more metaheuristics are chosen to be applied to the problem at hand, usually with little or no justification; where multiple heuristics are run, the one that performs the best in aggregate tends to be selected as the "best". Incorporating this type of modelling could make the choice more sophisticated, by identifying the characteristics of problems where each heuristic excels, and incorporating this model into the solution methodology.

One direction that was not explored in depth in the thesis was attempting to model heuristics and problem instance characteristics more generally. The modelling in Chapter 6 focussed on validating the concept of problem-heuristic modelling, and so focussed on predicting which of two heuristics would perform better. The actual similarity or otherwise of the heuristics was not taken into account. One of the strengths of the MLS framework is that it provides a way to place heuristics in "heuristic space", and qualitatively and, to a certain extent, quantitatively, describe their similarity. It would be interesting to attempt a more broad model that also uses the modules and parameters of the heuristic as input variables to the predictive models. Some research questions that would be interesting to answer is whether we can find more general relationships between heuristic types and problem types. For example, we might find that heuristics with more dramatic diversification techniques tend to do better on sparse problem instances, or that heuristics that explore only a small subset of the possible neighbourhood at each iteration (by controlling the search logic) tend to produce reasonable-quality solutions quite quickly on large problem instances, but that later in the search process heuristics that search more intensively have a tendency to perform better.

Those examples are completely arbitrary guesses, but they do suggest a new type of research that is made possible by combining the two main elements of our research. The MLS framework provides a structure that can be thought of as a "heuristic space", and heuristics can be grouped as similar or not based on the combinations of modules and parameters that they have. The problem-heuristic modelling can be used in combination with this to provide a methodology for systematically exploring these relationships, allowing a more "scientific" approach than is usually possible; making hypotheses like those suggested above, and then testing these hypotheses. It is hard to predict at this early stage where this type of research could go, but it seems to have at least the potential to radically change our understanding of *why* certain heuristics perform well on some problems and not others.

9.3.3 The Arc Subset Routing Problem

The final contribution of this thesis is also the least novel. A new combinatorial optimization problem was introduced, the Arc Subset Routing Problem (ASRP). A number of construction heuristics were developed, including a family of greedy variants, and two based on iteratively modifying the problem instance by removing arcs.

A number of problem instance characteristic metrics were developed and improved. In addition some consideration was given to comparing problem generation methods.

One key contribution related to the ASRP, but also more generally applicable was the concept of selecting a subset of problem instances that have maximum diversity, and the related concept of *generating* problem instances to exhibit desired performance characteristics, as a combinatorial optimization problem. Both of these ideas have the potential to assist a more systematic study of the relationship between problem and heuristic, as discussed in the previous section.

9.4 Further research directions

There are a number of specific research directions that could be followed to extend the concepts introduced in this thesis, and these can broadly be grouped into three categories:

- Extensions to MLS
- Further applications of MLS
- Other research directions, tangential to MLS

9.4.1 Extensions to MLS

The MLS framework, as described in Chapter 4, should be capable of implementing most trajectory-based metaheuristics that currently exist. However, there are a number of extensions that could be made to the framework.

9.4.1.1 Programming structure

The programming structure used in the experiments of this thesis is described in Appendix B. It is noted there that this structure has been a work in progress over the course of the thesis, and some points are made of how it could be slightly restructured to make it coincide more closely with the conceptual structure of MLS.

Significant work could be done to make this structure more robust, and more accessible for other researchers to use and extend it. There are some programming techniques that are designed for the creation of frameworks, and these could be usefully enforced – things such as properly controlled inheritance, interfaces, code documentation, etc.

The code is already relatively easy to use to run experiments, with the MLSML language being used to define new heuristics using existing modules. The standarised structure of MLSML would lend itself rather well to a Graphical User Interface (GUI) for the creation of heuristics, with drag-and-drop capabilities; the program would then create the MLSML code from the graphical design.

A nice extension would also be to enable database support. Currently all the inputs to the MLS engine are through text files, and the outputs are created as text files. These text files must then be parsed and imported into a data analysis software package. It would be useful for analysis if the program was connected to a database, and then analysis could be performed directly on these database tables.

9.4.1.2 Additional modules

During the experiments in this thesis we used a basic set of modules that originally came from the metaheuristics Steepest Ascent, Simulated Annealing, Tabu Search and Variable Neighbourhood Search. These were then recombined in different ways for the hybrids of Section 6.5, and even for the advanced heuristics of Chapter 7. However, there are still many other metaheuristics that can be expressed as MLS heuristics simply by the creation of the appropriate modules. For example, the many variations of thresholding methods that were discussed in Section 5.3, such as Threshold Accepting, Great Deluge and Record-to-Record Travel, each require only the creation of a slightly different admissibility condition module.

Some of the components of MLS that were introduced and "architected" in Chapter 4 were not really demonstrated at all. For example the neighbourhood reduction process is quite a rare feature; it does not appear in most metaheuristics. Making it an explicit feature though, invites the creation of potential modules that can then be tested against other methods.

Only a single variation of fitness function was explored, briefly as part of the advanced heuristic Adaptive Diversification Local Search, and there was no investigation of the effectiveness of this approach. In particular, fitness functions that get slowly modified according to the recent state of the search show alot of potential, as indicated by the success of Guided Local Search approaches. Studies comparing when fitness function modifications outperform neighbourhood modifications (Variable Neighbourhood Search) or admissibility modifications (Tabu Search) would be interesting.

Perhaps the MLS component with the most interesting unexplored potential is the change-current-solution component. This is a key component for many of the Iterated Search metaheuristics discussed in Section 5.2. Having a change-current-solution module available as a response could add a valuable tool to the MLS toolbox, especially since it can be used to provide a diversification effect that is as small (e.g., slightly perturbing the current solutions), as large (e.g., completely randomising the solution), or as targeted (e.g., revisiting a previous elite solution), as required.

The more modules that are added to the MLS library, the more combinations of these are possible, and the more potential there is for interesting hybrids and advanced techniques.

9.4.1.3 Population-based metaheuristics

Metaheuristics can be classified as either trajectory-based or population-based; in this thesis we restricted the scope of consideration to trajectory-based methods. Extending the MLS structure to include population-based methods would require some architectural changes, and there are several ways that this could be done.

A completely population-based version of MLS could be designed; since most population based methods are evolutionary perhaps this could be called **Modular Evolutionary Search** (MES). Instead of a single *current solution* and *target solution*, the heuristic would maintain a population of these. Instead of having a neighbourhood that contains solutions one move away from the current solution, MES would have a neighbourhood containing all the solutions in the next potential generation, based on all the possible "moves", which in this case could include such things as combining two solutions (genetic crossover) or those along the path from one solution to another (as in Scatter Search). From this total "neighbourhood" a subset of solutions would be chosen for the next generation. These are all analogous processes to those of the MLS search scheme, although Genetic Algorithms in particular have a number of other quite sophisticated mechanisms that could also potentially be modelled. In MES, as in MLS, the true power of our approach would be the trigger-response model, where the structure of the search could be modified as the search progresses, in response to certain conditions. It seems reasonable that with the expanded neighbourhood sizes that arise from introducing a population, there would be considerable scope for neighbourhood reduction processes to intelligently reduce these numbers, without simply limiting them as in a maximum move-list size.

What would perhaps be even more interesting would be a hybrid of MES and MLS. Instead of always maintaining a population of solutions, this could be optional. Consider a regular MLS heuristic, perhaps a simple Steepest Ascent configuration. When it reaches a local optimum a trigger is tripped and the response is to call a change-current-solution module that generates a population of solutions by slightly perturbing the current solution in a number of different ways, and also throwing in a set of remembered elite solutions that have previously be visited. The same move-types could be retained, in which case *each* of the solutions in the population would have the move-types applied to them (multiplying the size of the neighbourhood), or some new move-types could be activated that specifically deal with populations of solutions, perhaps by swapping solution features between solutions. After some iterations of this another trigger is tripped, perhaps by a certain number of iterations having occured, or perhaps because a new best-so-far solution is found. At this time another change-current solution module is performed that collapses the population down to a single solution, and the search configuration is also changed to facilitate an intensified search around this solution, for example by changing the move-types and increasing the candidate list size. This hypothetical type of heuristic, where an increase in population is just another of the modules available, and with a new set of move-types, could be extremely powerful, especially when combined with the ability of the search to "learn" what works over time.

9.4.1.4 Parallelization

Parallel metaheuristics are able to take advantage of multiple processors to perform tasks in parallel, rather than sequentially. The modular nature of MLS could make it relatively easy to adapt to parallelization. First, it is worth noting that there is not *necessarily* a qualitative advantage to parallelizing a heuristic; any process that can be performed in parallel can be performed in an equivalent way by a number of sequential processes, at least in theory. The main advantage offered by parallelization is that tasks can be performed simulataneously, and therefore the *elapsed duration* (if not the total processing time) can be reduced dramatically. For some of the more advanced heuristics such

as the MDP of Section 8.2, each neighbour evaluation and iteration can take a long time, and parallelization offers a way to speed this up.

There are several points in the MLS process that would benefit from parallelization. The most obvious is the neighbourhood evaluation. Say that there are 1000 moves in the move-list, and the candidate list size and examinations maximum are set to *unlimited*, so all of these moves must be evaluated as solutions and the best chosen. Each of these neighbour evaluations is independent of the others, so instead of performing these sequentially, they could be performed by a number of processes, up to 1000 if there were that many threads available. The computation-time savings could be considerable, especially if the evaluation of each neighbour were a significant exercise. In the case of the ASRP, each neighbour evaluation is relatively quick, since it simply involves adding up the rewards of the arcs in the new route. However, for the MDP each neighbour evaluation involves performing a complete heuristic execution on the current problem instance, which could take minutes or even hours.

Population-based techniques are the most obvious candidates for parallelization, and indeed these seem to be those in the literature that are most often parallelized, and most commonly for multi-objective problems (for example, [9,13,148,153].

The more interesting potential for parallelization is for the more advanced techniques. Consider a number of MLS heuristics, each operating independently, in parallel, on the same problem instance. If they are able to share some memory structures then this approach becomes very interesting, and the idea of **cooperation** becomes relevant. Bouthillier and Crainic [165] study a similar approach, where multiple parallel metaheuristics cooperate asynchronously by sharing information on elite solutions. In the MLS framework, each of these metaheuristics could be also changing its own structure over time, and perhaps sharing information on which types of *modules* are effective. This type of asynchronous sharing of information is very hard to replicate sequentially, without an even more complicated structure and lots of changing backwards and forwards.

The other side of cooperation, where heuristics act to assist each other as part of a larger problem solving metaheuristic engine, is **competition**, and evolution. Suppose we envisage an MLS configuration as a genome; modules and parameter values are the specific genes. It would be possible to create a population of MLS configurations, and let them all be executed and obtain a "best" solution value in parallel. We can consider a type of Genetic Algorithm where the more successful MLS configurations are "bred" together to create the next generation, and so on. This type of evolution of algorithms is known as Genetic Programming, and MLS provides an ideal structure for metaheuristics to be expressed as genomes (indeed, the MLSML and the MetaheuristicTemplate class described in Appendices B and C are actually used this way in the MDP). This type of large-scale experiment – the running of multiple metaheuristics for each iteration of a higher-level heuristic – is very suited for parallelization, since otherwise the time taken could be prohibitive.

In terms of implementing such a parallel system, the author has recently been investigating a relatively new service from Amazon, called the **Elastic Compute Cloud** (EC2). The concept is that a customer can "hire" a number of computer instances running virtually on Amazon's servers, for cents per hour per instances. These can be set up once and then cloned as many times as required, and this process can

be automated or controlled by an algorithm. The computing instances are able to communicate with each other using standard HTTP protocols, and are able to access shared databases and drive space.

It seems ambitious, but perhaps the ultimate end-point of the development of MLS-based engines is the following idea. A "master" MLS configuration controls the construction of others. Each of these "child" MLS configurations operates on its own EC2 instance, but they share some memory structures. The parent configuration peridodically shuts down poorly performing child heuristics, and sometimes clones others that are promising in order to explore multiple variations. Some of the child configurations are simple MLS routines, others are parallelized population-based techniques themselves, that are able to spawn new EC2 instances as required to examine as many solutions as needed. This whole structure could have multiple points where shared data can be disseminated and shared, for example each of multiple heuristics could be contributing to a shared elite solution list, and then a Scatter Search approach could be utilising this list to search the paths between these elite solutions for promising regions.

Of course such a scheme would be slightly overkill for solving 15×15 instances of the Arc Subset Routing Problem, but there are other problems on which existing heuristics do not perform adequately, and for these perhaps such a system might be justified. Some of these more challenging problems are discussed in the next section.

9.4.2 Further applications of MLS

As well as extending the MLS framework, as described in the previous sections, there is considerable scope for future research to apply MLS to other problems.

9.4.2.1 Benchmark problems

The most immediate application would seem to be to some of the standard problems that are used as benchmarks in the Operations Research literature. In the course of this thesis the Arc Subset Routing Problem was used as a test problem, both due to the development of the research topic, which grew out of this problem initially, and also because it allowed the creation of a large number of problem characteristics, especially when restricted to grid graphs. It would be useful to apply some of the MLS heuristics to benchmark instances of the Travelling Salesman Problem and the Capacitated Arc Routing Problem, to see how they compare. Especially when some of the more advanced ideas are developed, it would be informative to know if they are actually competitive with other state-of-the-art techniques.

9.4.2.2 Advanced heuristics

As demonstrated in Chapter 8, there is potential for the MLS framework to express extremely sophisticated multi-phase metaheuristics, even with just the current structure, without any of the extensions discussed above. Chapter 8 provided some proof-of-concept examples, but there is almost unlimited scope for developing these ideas further. Self-adaptive metaheuristics could be extremely powerful, especially combined with learning structures.

One example that was not examined, but which could prove interesting is loosely based on the *sequential fan* technique that was described by Glover and Laguna [120]. In this approach, at each iteration a number of different MLS configurations would be executed, starting from the current ASRP solution. The configuration that terminates with the best solution becomes the "selected" configuration. The search then moves to an ASRP solution from the solution trajectory of that configuration. One extreme would be to take the best solution that it found (which makes it the best that any configuration found). Another approach would be to move to the first ASRP solution in the search trajectory of the "selected" configuration (so the target solution is actually a neighbour of the current solution using the ASRP move-types). In this way the search would slowly progress, at each small step examining a "fan" of solution trajectories. This approach would likely be very computationally expensive, although it would be an ideal candidate for parallelization. In even more sophisticated variations, each of the candidate configurations could be modified and adaptive, perhaps with some type of learning.

9.4.2.3 Continual optimization and the Eternity II puzzle

The Eternity II puzzle [1] is an edge-matching puzzle that was released in 2007 that has a US$2m prize for the first complete solution, which is as yet unclaimed. It involves placing 256 square puzzle pieces into a 16 by 16 grid, such that adjacent edges have matching patterns. Each piece has its edges on one side marked with different shape/colour combinations. There are 4 corner pieces that are only matched on two sides, 56 non-corner edge pieces that are matched on 3 sides, and 196 inner pieces that are matched on all four sides. Each inner piece has four orientations. This is an extremely hard combinatorial problem, that is not suited to a brute-force approach; there are approximately 1.15×10^{661} possible configurations.

One way of measuring the quality of a solution is to count the number of matches; there are a total of 480 edges that must match for an optimal, and winning, solution. The author briefly attempted this problem in 2008 and achieved a score of 401 using only a basic tabu search heuristic; as of 31 December 2008 it was announced that the best partial solution was 467. All deadlines have passed without a complete solution being found, so it remains unsolved, and in fact unproven that an optimum exists.

This puzzle is described because it presents a new type of problem, one that a self-adaptive MLS approach might be suited for. Typical research in combinatorial optimization follows the pattern of studying a large number of problem instances for a given problem, and heuristic performance is assessed on aggregate performance measures. The goal of metaheuristics for these problems is to obtain a reasonable solution in a reasonable amount of time. The Eternity II puzzle has a different objective: the goal is to find an optimum, regardless of how long it takes. It is in the class of *NP* problems; a solution can be checked for optimality very quickly simply by counting the matches. This offers an ideal testing ground for advanced MLS concepts.

Testing advanced heuristics on 15×15 instances of the ASRP has two problems. First, it is likely that an optimum, or near optimum solution will be found relatively quickly; these problems may not offer a hard enough problem for the extremely sophisticated heuristic methods that are possible with MLS, such as those discussed in Section 9.4.1. Second, there is no benchmark against which the heuristics

can be compared; the optimum is generally unknown. The puzzle seems better on both counts: the problem is extremely hard, and there is an objective measure of progress, and knowledge of when an optimum is found.

We define a **continual optimization problem**[1] to be a problem where the solution method must continue trying to solve the problem, and not settle for any intermediate solution. We further consider two types of continual optimization: **static domain** and **dynamic domain**.

A static domain problem has fixed problem data, and the search is for the best solution possible. The Eternity II puzzle is an example of a static domain continual optimization problem where there actually is an optimum, and this can be checked. If the optimum is found then the search can terminate, so it is not necessarily infinitely continual. The alternative would be where it was not possible to verify the optimum, in this case the search could continue indefinitely in the hope of finding a better solution.

A dynamic domain problem is where the problem data changes over time. In this case there is no "optimum", although there will be optima at any fixed point in time. The goal with this type of problem is to continue improving the search, adapting to changes as they occur. An example of this type of problem is that of optimising airline schedules. Let us suppose that an airline schedule must be optimised via a metaheuristic each day. We further know that the best metaheuristic to solve this scheduling problem has changed over time, as the global economy and travel patterns evolve. One approach to this scenario is to apply the MDP approach, as introduced in Section 8.2. The metaheuristic used to schedule the airline would be the "solution" for a higher-level MLS configuration that guides the design of the heuristic. The interesting aspect would be considering how to assess the "quality" of a given "solution". Its quality should be tested on either real world data, or simulated data with the same properties. However, which data? The premise is that this data changes over time, so using the entire history would not give a clear picture of the current quality, however using only the most recent day would "overtrain" the heuristic. Some type of data selection would need to be applied, so that the testing data is representative of future data, so far as heuristic performance is concerned.

Returning our consideration to the Eternity II puzzle, which we may now class as a *verifiable-optimum static domain continual optimization problem*, we see that the MLS framework has the potential to be effective. It is not hard to design move-types; simple options include rotating a piece, swapping two pieces, and swapping more than two pieces. Note that when swapping more than two pieces the number of possible moves increases quickly; in this case the *move-list size* parameter of MLS will potentially be essential, for example allow swaps of 7 moves, but only allow 100000 randomly chosen swaps to be considered each iteration. This problem could offer a good opportunity to study the trade-offs between large neighbourhood size and slower iterations, and smaller neighbourhood size with faster iterations.

[1] Not to be confused with a "continuous optimization" problem, which is optimization of a problem where the variables have a continuous domain, as opposed to discrete, or combinatorial, optimization.

The usefulness of MLS for continual optimization will lie in its ability to completely change its structure as required, with a range of different diversification options. Consider a version of Adaptive Diversification Local Search, as introduced in Section 8.3. This heuristic would have a large number of modules available to be activated in various combinations, and a series of increasingly influential diversification techniques. A balance would be needed here; the solution should not be disrupted too dramatically until its potential has been fully explored, in case potential paths to the optimum are missed, but eventually it will become obvious that a plateau has been reached and the local region is a "dead end", in which case a diversification should be applied. The most dramatic diversification would be a complete reshuffle, which should occur when all other methods fail to achieve progress.

A variety of learning mechanisms can be in place, both concerning which modules are effective, as in ADLS, but also with solution elements. Consider a memory structure that keeps track of particular combinations of tiles, and the quality of the solutions in which they appear. Clearly some sort of filtering would need to occur to limit the number of combinations of tiles, however this type of memory structure could be used to inform the probability of various moves occuring (in the nature of a dynamic and probabilistic tabu list). Depending on the state of the search it may be desirable to favour keeping groups together that have previously been in solutions, or perhaps at other times the opposite would be preferred as having a greater dversification effect.

Although we discuss the Eternity II puzzle here, this is merely illustrative of the MLS ideas that could apply to any continual optimization problem.

9.4.2.4 Automated trading systems

Many other problem domains could be considered as appropriate targets for advanced MLS heuristics. We introduce the **Automated Trading System Design Problem** (ATSP). The goal is to develop a set of trading rules that can be algorithmically applied to open and close trades on a trading market, such as the foreign exchange (forex) market, which trades currencies. Trading strategies may be classed as either fundamental or technical. **Fundamental** strategies consider the real-world events and factors that should affect the prices of the instruments being traded, for example news releases and analysis of the economic situation. **Technical** strategies are based on the premise that all the necessary information is already captured in the time series of the prices by the activities of other traders; it looks purely at data, and calculates a number of *technical indicators* on which decisions are made. Realistically, both approaches have their strengths, however we consider only technical systems since they are amenable to solution by computer to create *automated* trading systems.

We choose to illustrate some ideas with the ATSP because it has a number of interesting features:

- It is an example of a *dynamic domain* continual optimization problem, as defined in the previous section. Although a system may be designed that is effective on historical data, this is no guarantee that it will be effective in the future. In practice, systems would need to be constantly reoptimised and refreshed. As has already been discussed, the ability of MLS to modify itself over time suits it for continual optimization.

- The evaluation of a particular trading system requires a simulation of the system against historical data; it is not possible to express the ATSP as a mathematical program. Within MLS this simply requires a more sophisticated *fitness function* module. There are many real world problems where the objective function can only be calculated algorithmically, for example a New Scientist article from 2001[187] discusses using Genetic Programming to design electronic circuits, that must be processed through a simulator to test their performance. Potvin et al. [211] study using Genetic Programming to devise trading rules for the Canadian stock market, with encouraging results.

- This is an important problem; a good solution would equate to the potential for profits in the real world.

The same advanced MLS approaches that were discussed previously in this chapter would also be suited for the ATSP, so we do not repeat these. However we do discuss briefly how this problem could be modelled in terms of what constitutes a solution, and what moves are possible. The author was originally intending an investigation of this problem to be included in the thesis, but time constraints did not permit the experimental phase to be started. However, much of the conceptual modelling and the programming of the MLS class structures was completed, and the ATSP is the next research direction that the author intends to pursue, eventually incorporating as many of the advanced features discussed in this chapter as seem reasonable; the ATSP seems like a complicated enough problem to justify some of the extremely sophisticated self-adaption, learning, and cooperative approaches, and it has a very clear definition of success: consistent profitability.

Currency trading occurs against a **currency pair**, for example EUR/USD is the Euro/US dollar currency pair. Every currency order involves buying one currency and selling another, and involves an **open** where the buy or sell order is made for a certain volume, and a **close** where the order is realised and any profit or loss is taken. For example, a 10000 unit buy order might be opened against EUR/USD at a buy price of 1.2921 (so 1 Euro buys 1.2921 USD), and closed at a sell price of 1.2929. The raw profit on this would be 10000*0.0008 = $8. There is a small difference in the buy and sell prices called the **spread** that accounts for the profit of the broker, so any trading system already needs to be better than this spread to be profitable in the long run. A **trading system** is a set of rules under which an order is opened and closed, along with some other safeguards such as a **stop loss**, which sets a limit on the amount of loss that can be made by the trade before the order is closed automatically.

The most granular unit of currency price data is called a **tick**. A tick represents a change in price rather than a fixed unit of time. So there might be multiple ticks in a second, and then there might be no ticks for several seconds. Tick data is often overwhelming, so is usually presented to traders in summarised form, for example as **candlestick charts**. These are look like box-and-whisker charts with bars representing a fixed period of time, for example, 1m, 5m, 15m, 1h, 4h, daily, weekly, or monthly. The top and bottom of the "box" represent the open and close prices for that time period, and the top and bottom of the "whiskers" represent the high and low prices. Traders will usually trade at a certain timeframe, using a particular chart as their main view, glancing at others occasionally to incorporate longer-term trends.

From a modelling perspective the only true way to test a trading system is to simulate it on tick data; many online platforms, such as the MetaTrader platform allow users to create "systems" and then backtest these, but this is done on simulated data rather than actual tick data, using the candlestick bars to generate simulated tick data. For an automated trading system, the concept of time-frames, or charts, should only be used as convenient reference points; all decisions should be made on a tick-to-tick basis.

We briefly describe the ATSP modelled as a problem for MLS; this is only one way of modelling this problem, there are many design choices that will influence the ability of the search to find good systems. The first modelling decision is to narrow the options that the trading system has. Instead of allowing both *buy* and *sell* trades, we allow only *buy* trades, which should allow the trading rules to be more focused. We do this without loss of generality by duplicating each currency pair and reversing the order. Each pair is usually expressed the same way, for example EUR/USD refers to the price of 1 Euro is US dollars. We duplicate and transform this data series to obtain USD/EUR, and a *buy* order on this series is equivalent to a *sell* order on EUR/USD.

A **solution** for the ATSP is composed of the following elements:

- One **order opening strategy**, which defines the conditions under which a *buy* order should be opened on the current currency pair.
- One or more **order closing strategies**, which each define a set of conditions under which an open order should be closed. If *any* of the order closing strategies are satisfied then the order is closed.
- An optional **take profit**, which is a parameter specifiying a threshold for profit; if the amount of profit for a trade reaches this threshold then the order is automatically closed and the profit is taken, regardless of whether the closing strategies are satisfied.
- An optional **stop loss**, which is a parameter specifiying a threshold for loss; if the amount of loss for a trade reaches this threshold then the order is automatically closed and the loss is taken, regarless of whether the closing strategies are satisfied.
- An optional **time limit**, which either closes the order after a certain amount of time has elapsed, or if a certain time event occurs (such as the end of the trading week).

The key elements here are the **order opening strategies** and **order closing strategies**. Both of these types of strategy are essentially the same; they are a set of one or more **order decision components**. An order decision component is a boolean condition that evaluates to either *true* or *false*. For an opening or closing strategy to be satisfied at least one order decision component must evaluate as *true*, and no decision components must evaluate as *false* (all the decision components must be satisfied).

An order decision component is satisfied if the following inequality is true:

$$C1 * V1(T1) + A > C2 * V2(T2)$$

where the decision component has the following elements:

- Two variables V1 and V2. The order decision components are based on data variables that are available. When a system is evaluated it is simulated against a dataset representing a certain period of tick data from the past. This dataset contains the price of the currency pair being traded and any number of derived variables. The simulation progresses through this dataset one row (corresponding to a tick) at a time, evaluating whether to open or close an order according to the rules of the system. The quality of the trading systems will heavily depend on the quality of the derived variables being used as the building blocks; this is discussed later. Note that one variable is always ONE, a variable that alway contains the value 1, allowing the decision component inequality to reference an absolute threshold.

- Two tick references T1 and T2. These are used to determine which values of variables V1 and V2 should be used for the comparison. There is a limited set of tick references that the system may have; some examples include: the current tick, a tick a certain period of time ago (e.g., exactly one hour), a tick corresponding to a candlestick bar (e.g., the close price of the first H4 bar of the week). There are many possible options for tick references; the more options the richer the search space becomes. Tick reference enable, for example, the comparison of the current price against the moving average of the consumer price index (CPI), if this was included as a variable.

- Two coefficients C1 and C2. These are used to scale the values of the variables.

- One constant ‘A’.

Note that the equality is always the same direction, since the opposite inequality can be obtained by switching all the elements and choosing a new constant.

An order opening strategy or order closing strategy is then composed of a number of rules defined by order decision components. The above system offers extreme flexibility in what these rules are, there are an infinite number of decision components and an infinite combination of components and strategies in a trading system; the challenge of the MLS heuristic is to search out good combinations.

The potential quality of the system depends heavily on the range and quality of the variables that are available to the decision components. The best approach is possibly to include all the variables that can be thought of and calculated, and then let MLS use a memory structure to slowly build up a record of the variables that are most often in “good” solutions, and perhaps weight these so that they become more likely to be included over time. Variables will include all the technical indicators that are commonly used in trading analysis, for *all* currency pairs, not just the one currently being traded, since there may be correlation or cointegration between pairs. These correlations and cointegrating coefficients should also be included, along with any other variables that are available representing the economies of the countries in question.

The *solution space* of potential systems is vast, and the challenge will be trying to find good ones. This is where a local search approach that uses iterative improvement could be valuable. Possible move-types inlcude, but are not limited to, the following:

- Adding a previously constructed order decision component from a "pool" of them to an order opening strategy or order closing strategy.
- Changing a variable, coefficient, tick reference, or constant, in an order decision component.
- Swapping variables in an order decision component (possibly taking their tick reference and coefficient with them, or possibly leaving them fixed in the inequality).
- Adding or removing one of the other solution elements: take profit, stop loss, or time limit, or simply changing its value.
- Any combination of the above things.

Moves that involve changing a parameter can either big large or small. There is an almost infinite number of moves possible, and this give a learning-based MLS approach significant scope to develop.

A trading sytem that is good on one currency pair, on one historical time period, will not necessarily be profitable in the future, or on other currency pairs. Some thought needs to be given to how these systems are evaluated so that they are not "overtrained" to the historical data. Perhaps a "problem instance" could be a stretch of time, say 3 months, for a single currency pair. A good system will then be one that is consistently good across multiple problem instances. Or perhaps each time the fitness function is evaluated (via a simulation on a historical dataset) a different problem instance could be used.

A real-world implementation of these systems would add another level of complexity. **Slippage** is a concept that needs to be incorporated in the simulations. This is where the price on which the decision is made is not necessarily the price that the broker is eventually able to settle the trade for, especially if the market is in a high state of flux. Slippage can be included in the simulation.

A real-world trading platform would probably have hundreds of strategies being constantly optimized. A higher-level control algorithm would monitor their performance and guide their optimization, and assign weights to them based on their current performance. In general, many models each making small trades are preferable to a single system making large trades – the profit curve is smoothed.

The MLS framework is, at least conceptually, capable of managing such a complicated system. Careful design of triggers and responses, along with appropriate memory structures, have the potential to be a powerful combination.

9.4.3 Other research directions

One of the ideas that provide continuity in Part III is that of ways to decide which heuristic should be applied to which problem instance. This was discussed somewhat as a motivation in the Introduction. The first method attempted, in Chapter 6, is to produce a large number of instances, and run a range of

heuristics on these, then attempt to model the relationship between problem characteristics and heuristics using multivariate techniques. This approach demonstrates some promise, but lack of diversity in the problem set used prompts an investigation into methods for developing diverse problem sets. Several heuristics were developed and tested in Chapter 7, and they show some promise.

The second method is to apply a variation of the MLS approach to actually *designing* problem instances that cause heuristics to distinguish themselves. The point of this investigation is that problem instances which cause quite different heuristic performance could potentially lead to insights about what about those special instances is similar and what is unique. A proof-of-concept investigation in Section 0 proves that this technique has potential, and also demonstrates the flexibility of the MLS framework on a new problem.

The third method comes at the problem from another angle. The first method attempts the problem like a scientist studying data obtained through observation: certain data is available from the results of previous experiments, and we attempt to model this data and try to understand it, with the hope of applying that model in a valid way to future problem instances. The second method attacks the problem in a more proactive way, actively performing experiments to obtain new data, but then still trying to fit a model with the hope that it will be generally applicable. The third method approaches the problem more directly. Instead of trying to model the heuristic that might be predicted to be best for a particular problem instance or class, this method actually *designs* the heuristic specifically for that instance.

The third method again uses the MLS framework in an original way, this time to *design* MLS heuristics. The modules and parameters that make up a MLS heuristic form the "solution" that is acted on by a higher-level MLS heuristic. "Moves" cause modules to be swapped and parameters varied, over a number of iterations designing the heuristic that is best suited for the instance under consideration. A proof-of-concept implementation of this was designed – the MDP of Section 8.2.

Although none of these methods was explored in great depth during the thesis, they do validate the concept of attempting to systematically understand the relationship of problem type to heuristic performance. We briefly suggest and discuss several possible directions for future research.

9.4.3.1 Problem-heuristic modelling

In Chapter 6 we demonstrated that it was possible to predict which of two heuristics would perform better on given problem instances, at least as an aggregate preference. The next step is understanding *why* this is the case. The fact that predictive models were able to make this classfication implies that there is some systematic effect that they are detecting. It would be an interesting and useful research investigation to attempt to understand this.

One approach that was briefly demonstrated in Section 0 is using MLS to specifically design problem instances that distringuish between the heuristics. Further developing this approach could involve generating a large number of instances that are better for heuristic A than heuristic B, and then a large number that are better for heuristic B than heuristic A, and then *analysing* the difference between these two problem sets.

There is considerable room for further research simply extending the approaches that were explored in Chapters 6 and 7; performing large numbers of experiments and analyses, then modelling and understanding the underlying causes.

Coda

▼ Summary

In this chapter we have described some research ideas that could be undertaken in future to extend and develop the concepts introduced in this thesis. This concludes Part III.

▼ Link

The remaining sections of the thesis are supplementary information: a glossary of MLS terms, a description of the program structures used to implement MLS, and a description and some examples of the MLSML code used to specify heuristics used in the experimentation. The bibliography concludes.

Appendices

APPENDIX A

Glossary of MLS Terms

The following terms have special definitions within the MLS framework and throughout this thesis.

Admissibility condition

One of the components of the MLS search scheme. An admissibility condition determines whether a neighbour solution is eligible to be selected as the target solution for the current iteration. The admissibility conditions are only assessed for those neighbours that are chosen by the search logic to be evaluated. See Section 4.3.4.

Best-so-far

An automatically maintained memory element that is the best solution found so far. There is some discretion to the designer of the heuristic as to whether this is based on the objective function or the fitness function; it will depend on which module is specified. See Section 4.4.3.3 and Section 4.5.5.

Candidate

A neighbour solution that has been selected by the search logic to be evaluated, and meets the admissibility conditions. See Section 4.3.4.

Candidate list

The set of **candidates** that have been evaluated and are admissible. The best solution (highest fitness) on the candidate list is always selected as the target solution for the current iteration. See Section4.3.5.

Change-current-solution

A particular type of **response** module that changes the current solution. At the end of each search iteration the current solution is set to the target solution that was just found. This type of module overrides this default process. See Section 4.4.4.6.

Component

A particular aspect of the MLS framework that fulfils an architectural and functional role. Each component will have specific modules that perform its role. See Section 4.2.

Control system

The collection of MLS components that are related to the "intelligence" of the heuristic, primarily those which are involved in modifying the current state: the triggers and responses, the update-memory functions, etc. See Section 4.4.

Examination order

The order in which *moves* are drawn from the *move-list* in order to be evaluated as solutions (recall that *moves* simply describe the operation to occur without actually performing it), and assessed against the admissibility conditions. See Section 4.3.5.

Examinations maximum

One of the components that controls the scope of the search in each iteration. The maximum number of moves that may be evaluated as solutions and assessed against the admissibility conditions. See Section 4.3.5.

Fitness function

One of the MLS components within the search scheme. The fitness function determines how a solution is evaluated by the admissibility conditions, and may be used in other modules as required. The most common fitness function is simply the **objective function** for the problem. An MLS heuristic may only have one fitness function at a time. See Section 4.3.3.

Generate-initial-solution

A component of MLS that generates the initial solution on which the MLS search process acts. Is treated as a black-box function, and specific modules could be construction heuristics, random generation, or using some previously found solution. See Section 4.4.1.

Heuristic

A term used to refer to any local search approach. Used interchangeably with **metaheuristic** and **local search**, and **MLS configuration**.

Hybrid

Any MLS heuristic that features a combination of modules that were originally associated with different metaheuristic paradigms. A label of convenience rather than a strict classfication.

Initialize-memory

An optional component of the MLS control system. A heuristic might specify initialize-memory modules to initialize any special memory strutures required, for example pre-populating a list of interesting solution features. Automatic counters etc. are initialized by default and do not require a specific initialize-memory module. See Section 4.4.2.

Local search

Any heuristic method that moves from solution to solution.

Memory element

A particular instance of an MLS memory structure, can be a parameter, a list, or some other structure. Memory elements can be read by other modules, and can be modified by some modules such as responses and update-memory functions. See Section 4.5.

Memory parameter

A memory element that takes a single value. Many of the MLS components are stored as memory parameters, such as the candidate list size. See Section 4.5.

Memory structures

A collective term for all the user-specified and automatically maintained memory elements. See Section 4.5.

Metaheuristic

Any local search technique. Used interchangeably with **heuristic**, **local search**, and **MLS configuration**. Often used to refer to local search techniques with some mechanism for intensifying or diversifying the search, especially to escape from or avoid local optima.

MLS

The Modular Local Search framework.

MLS configuration

A particular set of MLS modules that have been specified to be executed as a metaheuristic. Can be thought of as a "family" **of MLS instances**, and might specify some modules that are initially inactive, and a range of parameters settings.

MLS instance

A realization of a MLS configuration, which includes just those modules that are active at the current time, with particular settings for the parameters.

MLSML

Modular Local Search Markup Language. MLSML is a language developed to express MLS heuristics declaratively; it forms the blueprint for an MLS configuration. MLSML is what serves as the input to the MLS program. See Appendix C.

Module

A packaged function of a specific type that fulfils a specific role in an MLS component. These are the building blocks of MLS heuristics. See Section 4.2.

Move

A specific realization of a move-type. If a *move-type* is the swapping of two solution elements, then a *move* might be swapping C and F. See Section 4.3.1.

Move selection order

The order and process by which **moves** are created. Since the total number of moves that are defined might be less that the total number possible (especially if the search space is infinite), this can affect the nature of the search. See Section 4.3.1.3.

Move-list

The set of **moves** that have been explicitly defined. The move-list forms the pool from which moves are selected to be examined by the search logic. It is a (possibly equal) subset of all the possible moves. See Section 4.3.1.

Move-list size

The number of **moves** that are defined and added to the move-list. This may be *unlimited*, or may be limited in order to control the scope of the search. See Section 4.3.1.2.

Move-type

A procedure for transforming one solution into another. If a *move-type* is the swapping of two solution elements, then a *move* might be swapping elements C and F, for example. See Section 4.3.1.

Neighbour

Any solution that can be reached from the current solution be the application of one move.

Neighbourhood reduction process

An optional component of MLS. A neighbourhood reduction module starts with the move-list, and transforms it to another move-list that is a (possibly equal) subset of the original move-list. See Section 4.3.2.

Neighbourhood scheme

The collection of MLS components relating to the moves that are available: the **set of move-types**, the **move-list size**, and the **move selection order**. See Section 4.3.1.

Response

A module that performs a specific action to modify the search or the heuristic. Responses are associated with **triggers**, so that when a trigger is *tripped*, its associated responses are performed. A response might be to perform some other MLS module, such as an update-memory module. See Section 4.4.4.

Search characteristics

Automatically maintained memory parameter metrics relating the most recent iteration of the search iteration process, such as the number of solutions examined, etc. These may be used within other modules, such as triggers. See Section 4.5.4.

Search iteration process

The key MLS process that starts with a current solution and finds a target solution according to the components that make up the **search scheme**. See Section 4.2.

Search logic

The collection of MLS modules that controls which moves are selected to be evaluated, and how many. Consists of the **candidate list size**, the **examinations maximum**, and the **examination order**. See Section 4.3.5.

Search parameters

These are technically "modules", but they take the form of compulsory memory parameters: the **move-list size**, the **examinations maximum** and the **candidate list size**. These MLS components are stored as memory parameters so that they be used and modified like any other parameters during the search. See Section 4.5.1.

Search scheme

The collection of MLS components that controls the seach iteration process. The modules of the search scheme define which neighbours of the current solution will be examined to determine the target solution. See Section 4.3.

Search space

The set of solutions that are reachable by the current heuristic configuration. May be a subset of the **solution space**.

Set of move-types

The currently active move-types that are available to be be performed on the current solution. Defines the neighbourhood. See Section 4.3.1.1.

Solution space

The complete set of solutions to the problem. Not all solutions are necessarily reachable by a given heuristic.

Target solution

The solution that is selected as a result of the search iteration process. The target solution is always the candidate with the highest fitness function value. If there are no candidates then the current solution becomes the target solution. See Section 4.3.

Trigger

A MLS module that checks if a certain condition is satisfied (the **trigger logic**), and if so then any associated responses are performed. Must evaluate to *true* or *false* (*tripped* or not *tripped*). If the trigger is currently *inactive* then it is not evaluated. See Section 4.4.4.

Trigger logic

Also known as the **trigger condition**, this is the function that determines if a trigger is tripped or not. Can refer to memory elements. Must evaluate to *true* or *false* (*tripped* or not *tripped*). See Section 4.4.4.1.

Trigger memory element

A memory element that is specifically present to be used in the evaluation of the trigger logic. May require additional update-memory modules to maintain. See Section 4.4.4.

Update-memory

A module that updates one or more memory elements. This can be after analysis or other functions are performed. A common use of an update-memory module is as a response to a trigger being tripped. If an update-memory module is *active* then it is performed automatically immedidately after the search iteration process, every iteration. If it is *inactive* then it is only performed when specifically called by a response. See Section 4.4.3.

APPENDIX B

Programmatic Structure

In the following sections we describe the general structure of the programming code used to implement MLS. This structure is a continual work-in-progress, and suggestions are made for possible improvements.

B.1 Introduction

The major contribution of this thesis is the introduction of Modular Local Search (MLS), a framework that allows easy and sophisticated hybridization of metaheuristics, especially *multi-phase* hybridization, where the structure and operation of the metaheuristic change significantly during its execution. The previous chapters have presented.

The MLS architecture presented so far is problem-agnostic; it deals simply with "solutions" and "moves". The implementation aspect must also be considered, in which the problem domain is important. We have separated the two aspects. The core MLS engine is problem agnostic – the classes pass around "solutions" and call functions such as `getObjectiveValue`. Then for each problem domain a set of classes need to be written. These follow a standard template and have certain functions they must implement.

The MLS structure has evolved over the course of the research (started out C++, now Java), and so has the programming code. This Appendix documents the current state of the programming structure, but we also make notes of which parts could be improved. The programming interface has grown over a number of years, and has been rewritten several times, but still is in a state of constant improvement. It has been getting steadily more general and robust, but there is still some work to be done; it is a work in progress. There is not a perfect correspondence with the conceptual framework – there are some parts that are lagging with previous ways of doing things, and if a workaround was functionally equivalent it hasn't always been updated yet.

This Appendix describes the general structure of the program, and lists the main classes. There are lots of other supporting classes that are not discussed, and within each class lots of supporting functions

such as *getters* and *setters* for the variables. The full code base contains 90 classes (including all the subclasses for each problem domain) and 16216 lines of code. This has been completely rewritten in Java several times, and replaces an earlier C++ version, which replaced the C version used for the ASRP investigation that was non-object-oriented. It took literally several months just to get Edmonds blossom matching algorithm working properly, days of tracing graph matchings by hand until bugs were discovered. There was no sufficiently detailed algorithmic descriptions of this algorithm – they all dealt at a high level. Then this algoritm was reprogrammed, taking more months, for the new java object-oriented arc and node structure. This motivates having a reusable programming structure like MLS, so that this dosen't need to be done every time.

B.2 Object-oriented programming structure

Object-oriented programming is method of programming that uses "objects", data structures that encapsulate a set of variables and methods. Each object is a member of a "class". There are many resources available that give good introductions to this style of programming, for example, Schach [226].

One design decision was how to handle classes and subclasses. There were two main options. The first is to abstract the classes completely, so for example each type of move is a class of its own, which are subclasses of the Move class. The other way is to have a single Move class, with conditional logic that executes the appropriate function based on the type of move, which is a parameter of the object. We have used a hybrid, we contain all the functions for a particular problem domain within a single subclass for each problem domain. We call this the container pattern. The following code gives an example.

```
private String type;

public void execute(){
    if(type.equals("Type 1"){
        executeType1();
    }
    else if(type.equals("Type 2")){
        executeType2();
    }
}

private void executeType1(){
    // Code for execution of Type 1
}

private void executeType2(){
    // Code for execution of Type 2
}
```

A future improvement to the code would be to create separate subclasses for each move-type (and every other MSL component that uses the container pattern). This would complicate the code, but would make the system more robust and extensible.

B.2.1 Core MLS classes

These classes contain the core functionality of the MLS process, which is the same for any problem domain. Their methods pass and act upon generic classes such as "Solution", even though the actual objects they are passing will belong to subclasses such as "AsrpSolution".

- **MetaheuristicTemplate**. This class is the "blueprint" for a metaheuristic, or MLS configuration. When the metaheuristic specifications are read from the MLSML file, each is stored in a MetaheuristicTemplate object. All of the modules and parameters are stored as String variables (plain text), rather than as the actual objects that they are converted to when a Metaheuristic object is instantiated from a MetaheuristicTemplate object. For example the *admissibility conditions* are stored in an ArrayList of Strings. The MetaheuristicTemplate class contains a method to construct a MetaheuristicTemplate object from an XML Element, which is passed when the class is instantiated.

- **Metaheuristic**. The Metaheuristic class contains the main structure of the procedure. It's main procedure is the `executeMetaheuristic()` function, which sets the current solution by calling a construction heuristic, and then loops through the search iteration process until the heuristic is terminated. It contains a Heuristic object, which performs most of the operations of finding the target solution. After the Heuristic object is iterated, the automatic counters are updated, along with any other update-memory objects that are required, and the best-so-far, and then the triggers are processed.

- **Heuristic**. The heuristic class is actually the *search scheme* of the MLS process. The Heuristic contains objects for all the components of the search scheme: the move-types, the fitness function, the admissibility conditions, the search logic parameters, and the update memory functions. It also contains all the procedures needed to perform the search iteration process. The main control function is the `iterate()` function, which is called by the Metaheuristic that contains it. The first step is that the move-list is generated, then the candidate list is generated from the move-list, and then the target solution is selected.

- **Trigger**. Trigger objects belong to a Heuristic object. The Trigger class follows the *container* pattern described above. When its main `process` method is called if checks to see what type of trigger it is, and then calls the appropriate method to check whether it has been tripped. Most of these methods perform some lookup of the memory parameters, such as the iteration count. If the trigger detects that it has been tripped then it loops through its list of responses, calling and executing each one in turn.

- **Response**. Response objects are contained in a list of responses within a Trigger object. The Response class also follows the container pattern, although this class would perhaps be better structured as a main response class, and a number of sub classes, one for each type of response. When its main method is called it checks to see what type it is (it has a "type" variable), and then the main method calls the appropriate method for this type.

- **Neighbour**. The Neighbour class contains information about a particular neighbour that has been evaluated: the actual Solution object for this neighbour, whether it was found to be admissible, the Move object that created it, the MoveType that created it.

- **Memory**. This class contains all of the memory elements (parameters, lists, etc) that are used by the other classes. Each experiment has its own Memory object and this is referenceable by every class. Because there is endless variety of memory elements, these are not explicitly listed, instead they are stored in an extendable list actually a HashMap, that is dynamically generated when the MetaHeuristic object is created from the memory elements specified in the MetaheuristicTemplate (from the MLSML specification). The Memory class also stores all the automatic counters.

B.2.2 Generic classes and factories

Many of the classes and modules used in the MLS framework are specific to a particular problem domain. The core process is domain-independent, but the individual fitness function evaluations and moves, for example, need to interact with the problem data and need to be developed especially. Some classes, such as the admissibility conditions, can have some modules that are problem independent, and others that depend on the problem data.

The way that this is handled is by using the **factory** design pattern. For each type of class, there is a generic class and a factory class, and then each problem domain also has a subclass. We illustrate this concept with the MoveType component.

There exists a generic MoveType class. This contains all the variables and functions that are required by all MoveType objects: the number of moves remaining to be generated, a generateMove method that returns a Move object when called by the Heuristic object, and some methods that handle randomising the order of the moves.

Each problem domain then has its own MoveType class, for example AsrpMoveType and MdpMoveType. These classes *extend* the generic MoveType class, which means that they are sub-classes and inherit all the variables and methods in the MoveType class. Each domain-specific subclass contains its own logic for generating Move objects when called, for example the AsrpMoveType class can generate ASRP moves (*add*, *drop*, *shortcut*, etc.), whereas the MdpMoveType generatesMDP moves (*add module*, *increase parameter*, etc.). Note that the MDP is introduced in Chapter 8.

Each type of class then has a “factory” class, for example MoveTypeFactory. A simplified version of the code for the factory class is shown below:

```
public class MoveTypeFactory {
    public MoveType generate(String problemType, String name){
        if(problemType.equals("ASRP")){
            MoveType moveType = new AsrpMoveType(name);
            moveType.setProblemType(problemType);
            return moveType;
        }
        else if(problemType.equals("MDP")){
            MoveType moveType = new MdpMoveType(name);
            moveType.setProblemType(problemType);
            return moveType;
        }
        else return null;
    }
}
```

When a Metaheuristic object is created from the MetaheuristicTemplate object, based on the MLSML description, all that is specified is the name of the move-type to be added, for example "Shortcut". The system does not know from this template whether this move-type is an ASRP move-type, or an MDP move-type. The key parameter that is used here is the *problemType* parameter. Every class has a *problemType* parameter, which is simply a String that is set when the class is created, as can be seen above. When a new move-type object needs to be created, a MoveTypeFactory object is created, and its `generate` method is called, passing it the problem type, and the name of the move-type. The factory class then checks to see what the problem type is and creates a MoveType object of the appropriate subclass.

This pattern is used for many of the components of the MLS framework.

B.2.3 Domain-specific MLS classes

As described in the previous section, there are a number of problem-specific classes that are subclasses of generic parent classes. The main MLS logic only cares that it is handling, for example, a MoveType class with a `generate` method, whereas in reality there are a number of MoveType subclasses, each with a different `generate` function.

Most of these domain-specific classes follow the container pattern, they are a single class with a control function that looks up the appropriate method to call based on their "type" parameter. One improvement that could be made to the this programming structure would be to convert these classes to distinct classes. Currently we have an AsrpMoveType class (which extends the generic MoveType class), with a master `generate` function that calls a different function based on the "type" parameter of the AsrpMoveType (e.g., generateAddMove, generateDropMove, etc). Perhaps a better structure would be to have a generic MoveType class, and then a whole series of specific subclasses, one for each move-type, e.g., AsrpAddMoveType, AsrpDropMoveType, etc. This makes the code base slightly more complicated, which is the reason it was not designed like this initially, but it makes it better for extension. By having multiple subclasses, each one has a simpler structure, and is a distinct unit of

code. Other researchers can add new move-type classes to a library of classes without needing the ability to modify a "master" class for that problem domain.

We briefly describe each of the domain-specific MLS classes:

- **AdmissibilityCondition**. Required only to have a single method, `isAdmissible`, that takes as input a Neighbour and the current Solution. This function returns a boolean result of *true* or *false*. The domain-specific subclass follows the container method, however this class is perhaps the best candidate for a change to multiple subclasses as described above, since many admissibility conditions are generic to all problems, whereas the current structure requires these to be replicated for each new domain. An example of a generic admissibility conditions is the Metropolis condition, which uses only a comparison of the fitness values of the current solution and the neighbour, and some memory parameters such as the *temperature*.

- **ConstructiveHeuristic**. This should more properly be named GenerateInitialSolution, to bring it in line with the MLS terminology; it is named like this since the only method of generating initial solutions used was a constructive heuristic. This is clearly very different for each problem domain, and again follows the container pattern, although could be split out into separate classes, one for each construction heuristic. This class is only required to have one method: `constructSolution`, which returns a Solution object.

- **FitnessFunction**. The only method a FitnessFunction subclass must have is a `calculateFitness` method that takes as input a Solution object. This method does not return anything, instead it modifies the Solution object, updating its *fitness* variable. Subclasses follows the container pattern.

- **Instance**. This class contains the problem data. Subclasses will be very different from each other, since each problem domain is different. The only method that is required is an `initialise` method, but this does not necessarily need to do anything. For the ASRP the AsrpInstance subclass contains a Graph object, a Node object for the depot, a variable for the budget, and some supporting methods, for example the code required to instantiate the instance by reading in the text file that contains the instance data.

- **MoveType**. This class was discussed as an example above. The way the MoveType class works is that each iteration the Heuristic class that contains the MoveType calls its `reset` method, which takes as input the current Solution. This function initialises the MoveType object. This will potentially be a different process for each problem domain, and for each move-type, so we illustrate the concept by describing the process for the ASRP move-types.

  For the ASRP move-types (*add*, *drop*, *shortcut*, etc) the reset function calculates how many moves of this move-type are possible. It does this by maintaining a number of lists. For the *add* move-type, each move has two parameters: the index of the node on the route where the addition starts, and the index of the node on the graph that is being added. This sets an upper limit on the number of *add* moves that are possible: if there are p nodes on the route and q nodes in the graph, then there is an upper limit of pq *add* moves. Of course not all of these will

result in valid moves, since for the *add* move-type, *p* and *q* must be adjacent. The `reset` function creates two lists of integers: list1 contains integers from 0 to *p*-1, and list2 contains integers from *q*-1. The `reset` function then randomises the order of these lists, so that node 1 is not always examined before node 2.

When building the move-list, the Heuristic object then repeatedly calls the `generateMove` method of the move-list (the MoveType object is also selected randomly from all the MoveTypes available) until there are no moves remaining in all MoveType objects, or until the move-list size threshold is reached. The generateMove function follows the container pattern, and uses the lists that were randomised above to generate the next Move object. After it is generated, the lists are updated so that they only contain unchosen moves.

- **Move**. A Move subclass needs only contain a method `generateNeighbour`, which generates a Neighbour, following the container class. Move objects are generated by the MoveType objects. For the ASRP, the AsrpMove class also contains a number of indexes that are required to identify starting and ending nodes for the various ASRP move-types. Each Move has as a variable the current Solution object. Each specific `generateNeighbour` function (e.g., `generateAddNeighbour`, `generateDropNeighbour`, etc), first creates a Neighbour object and sets itself as the Move variable for the Neighbour. It then clones the current Solution and sets this new Solution as the Solution variable for the Neighbour. This new Solution is then modified according to the procedure of the move; for an *add* move it inserts an arc into the route twice at a specified point. The fitness function is then calculated for the resulting new solution.

- **Solution**. The Solution subclass will be different for each problem type. Solution subclasses are required to have the following functions: `getFitnessValue`, which returns the fitness value for the solution, `isFeasible`, which checks feasibility, and `getObjectiveValue`, which returns the objective function value (which may be the same as the fitness function value).

  The AsrpSolution class is relatively complicated; it performs many of the problem-specific tasks required to implement the ASRP. In addition to the required functions above, it contains two lists that represent the current solution route: an ordered list of Node objects that represents the nodes visited in the route, and an ordered list of Arc objects that represents the arcs visited. Both lists are necessary, since each is suited for certain calculations. Whenever a change is made to the route, both lists must be updated.

- **UpdateFunction**. This class should properly be named MemoryUpdate, in line with the MLS component naming conventions. These classes follow the container pattern and perform whatever calculations are required, updating any memory elements that are needed. One AsrpUpdateFunction involves updating the tabu status of the any arcs that have been changed from last solution, decrementing the remaining tenure of arcs already on the list, and removing any arcs that have no remaining tabu tenure.

B.2.4 Domain-specific support classes

In addition to the MLS-specific classes described above, each problem domain requires a number of other classes. These will vary depending on the problem type. For the experiments in this thesis, most experiments were done on the Arc Subset Routing Problem (ASRP); even the other problem domains that are explored that use MLS in more advanced ways are based around the ASRP.

We briefly describe the classes required to support ASRP problem instances. Recall that this structure is completely object-oriented. All of the following classes are those that are required to use *graphs*; the other elements of an ASRP problem instance (the budget, depot node, etc), are parameters of the Instance class. The structure of the graph is maintained because each node and arc knows where it is in relation to its neighbours. Note that these structures will apply equally well to any node or arc routing problem, not necessarily based on grid graphs.

- **Node**. Represents a node of the graph. Contains a list of incident Arc objects, and a method `isIncident` that takes an Arc as input and returns a boolean result for whether it is incident on this Node. Also contains a method `getIncidentArcFromAdjacentNode` that takes another Node as input and returns the Arc object that is between these nodes (or *null* if there isn't one). Similarly it contains a list of adjacent Nodes, and a method `isAdjacent` that checks whether an input Node is adjacent.

- **Arc**. Represents an arc of the graph. The Arc class contains similar lists of adjacent Arcs and incident Nodes to the Node class, and similar functions for checking incidence and adjacency. It also contains variables for the reward, penalty and cost associated with the Arc.

- **Graph**. The Graph class is primarily a container for the list of Arcs and list of Nodes. It also contains some special functions, such as a test for connectedness.

A separate set of classes are required to support the *generation* of problem instances, based on grid graphs.

Of particular note are the classes and methods required to perform Edmonds matching algorithm (see Edmonds and Johnson [78]). This algorithm finds a maximum weight perfect matching on a graph, and is the preferred method for making a graph or subgraph Eulerian, a necessary step for finding a minimum cost route through the arcs of the subgraph. Three special classes: Euler, EulerArc and Euler node contain the logic required to implement this algorithm, which required significant development effort to program and debug, resulting in approximately 1800 lines of code.

B.2.5 Environment classes and experimentation

There were numerous other classes required to support the execution of computational experiments. In this section we briefly describe the files and procedure used to execute these experiments.

The inputs and outputs of the program are all text files:

- Input: **MLS_Settings.ini**. This is the first file read by the program, and it contains settings such as the names of the other input files, paths to where these files are and paths to where output

files should be placed, and the problem-type (ASRP, MDP, etc). None of the other files listed below need to have the name given, these are instead specified in this file.

- Input: **metaHeuristics.xml**. Contains the MLSML code that specifies the heuristics being experimented with.
- Input: **ScheduleHeuristics.csv**. Contains a list of names of the heuristics that should be applied to the problem instances. These names must match names in the MLSML file, but not all MLSML entries need to be listed in this file.
- Input: **ScheduleInstances.csv**. Contains a list of the filenames of all the problem instances that should be solved. All the heuristics listed in the previous file are applied to all of these instances.
- Input: ***instanceFilename*.instance**. There will be a number of these instance files, one for each of the entries in the ScheduleInstances file. Each one contains the problem data for the problem instance, starting with a section giving the budget, depot node, and number of arcs and nodes, and then a section listing the nodes, along with their grid row and column, and then a section listing the arcs with which nodes they are incident on, and their reward, penalty, and cost parameters.
- Output: **Result.*Instance*.*Heuristic*.txt**. Contains the results of the application of one heuristic to an instance. Each result produces its own file. The results include the time taken, the reward collected by the best solution, and the cost of the route.
- Output: **Route.*Instance*.*Heuristic*.txt**. Contains the route for the best solution found.

An **Experiment** class handles the execution of the program; it reads in the schedules and then loops through them, creating the Metaheuristics and Instances in turn, and executing them, then outputting the results.

Another important tool was an Excel spreadsheet model that had macros to assist in visualising problem instances and routes. The path to a instance file is entered into one cell of the spreadsheet, and the path to a route is entered into another. Then, buttons can be pressed to execute VBA macros that construct the problem instance visually, and then if desired display the route, pausing after each arc. Double lines are used where the route traverses an arc twice.

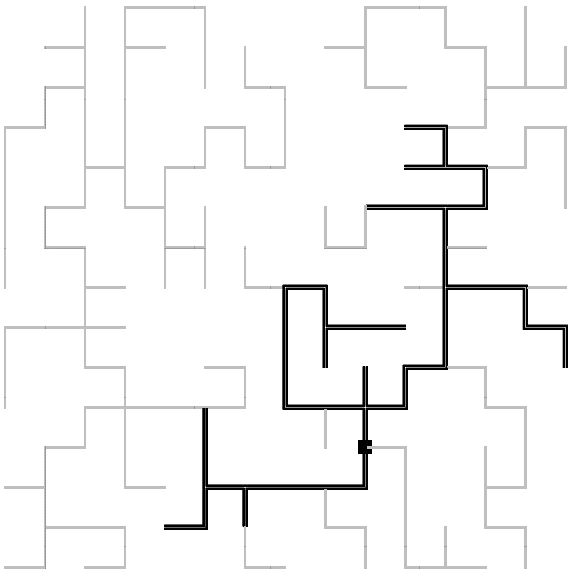


Figure 9.1: An example of a route displayed with the route visualizer

Figure 9.1 gives an example of a route that traversed a subset of arcs twice, and no arcs once, due to the sparse nature of the graph.

B.3 Extension to new problem domains

MLS is designed to be easily applied to new problem domains. In this thesis three distinct problem domains were implemented: the ASRP, the MDP (discussed in Chapter 8) and the APICP (discussed in Chapter 7). The main control logic of the MLS procedure remains completely unchanged; only problem-specific classes need to be created:

- **Admissibility conditions**, if required. Ideally many of these can be reused from existing classes.
- **Construction heuristics**. These will be dependent on the problem domain. If necessary a simple random solution can often be easily constructed.
- **Fitness functions**. The default is simple "Objective", which passes the calculation logic back to the Solution class.
- **Move-types and Moves**. These follow a standard structure, but the specific logic to generate a Move, and Neighbour, will be different for each move-type, as well as each problem domain.
- **Solution**. The Solution class is where the MLS algorithm interacts with the problem data.
- **Update functions and other modules**. If there are any specific module required for the problem or solution techniques these must be developed.
- **Output functions**. These have not been explicitly discussed yet, but each problem domain will need to output different things. For example the ASRP outputs the best route found... non-routing problems will have other output requirements. Again, these follow the factory pattern,

so that the control class (Experiment) only needs to execute the Output method of the Output class, and the subclass function of the appropriate type is executed.

- **Other problem data classes**. These can potentially take the longest time to program. For the ASRP, the Node, Arc, and Graph classes required significant development effort to perfect.

APPENDIX C

Modular Local Search Markup Language (MLSML)

In the following sections we describe the structure of MLSML and provide examples of the MLSML code used to specify the heuristics used in the experimentation. Only a single example of each type of heuristic is presented, to illustrate the usage of MLSML.

C.1 Structure of MLSML

One of the design goals of the MLS framework is that new hybrid metaheuristics can be created simply by specifying the combination of modules and parameters that should be used. We introduce a new markup language, Modular Local Search Markup Language (MLSML) as the way that these combinations of modules and parameters are specified.

Markup languages are a way of annotating text such that separate elements are structurally identifiable. See Coombs et al. [59] for an introduction to markup systems. Modern markup languages, as well as MLSML, tend to be modelled on XML (eXtensible Markup Language), which is a specific set of rules for developing new markup langauges (see the World Wide Web Consortium XML homepage for more details [2]). The most common markup language is HTML. Essentially, information is surrounded by "tags" that label the information type. For example

```
<moveType>Shortcut</moveType>
```

is an example of a piece of MLSML data. The tag `<moveType>` announces that the text following is the name of a move-type. The tag `</moveType>` "closes" this tag, announcing the end of this piece of data.

We briefly describe the structure of an MLSML file. Note that this structure represents only the most recent iteration of MLSML; this strucure has constantly been in development, and it may be expected to expand for future research. Examples of the MLSML specifications for the some of the MLS configurations used during this thesis are given in subsequent sections..

The tags used in MLSML are described below. These are listed in the order in which they generally occur in an MLSML specification file, although the order is not important; the MLSML parser is able to

extract the tags in any order. Each <tag> should be accompanied by a closing </tag>, although these are not explicitly listed.

- `<metaHeuristics>`
  This is the top level tag for the MLSML specification file. An XML-based file should have a single root "node" or tag.

- `<metaHeuristic>`
  The container tag for each individual MLS configuration.

- `<name>`
  The name of the heuristic. This should match the name in the schedule file, and is used in outputs to identify the heuristic, but otherwise has no effect.

- `<admissibilityConditions>`
  A container tag for the other individual `<admissibilityCondition>` tags. This tag is optional; multiple `<admissibilityCondition>` tags can be directly beneath the `<metaHeuristic>` tag and the MLSML is still valid.

- `<admissibilityCondition>`
  Contains the name of an admissibility condition that is active for this heuristic. This name must match that programmed into the AdmissiblityCondition classes.

- `<fitnessFunction>`
  Contains the name of the active fitness function for this heuristic. This name must match that programmed into the FitnessFunction classes.

- `<neighbourhoodScheme>`
  A container tag for the `<moveType>` tags. This tag is not optional.

- `<moveType>`
  Contains the name of a move-type. This name must match that programmed into the MoveType classes. Multiple move-types are usually specified.

- `<memoryParameters>`
  A container tag for the `<memoryParameter>` tags. This tag is not optional if there are memory parameters specified (if present, they must be contained in this tag).

- `<memoryParameter>`
  Represents a memory parameter. Contains various child tags that represent the attributes of the memory parameter: `<parameterActive>`, `<parameterName>`, `<parameterValue>`, and optionally, `<increaseType>`, `<increaseValue>`, `<minValue>`, `<maxValue>`.

- `<parameterActive>`
  Contains either "true" or "false" indicating whether the memory parameter is initially active.

- `<parameterName>`
  Contains the name of this memory parameter. This name may be anything desired, but if modules reference this parameter is will be by name so these should match.

- `<parameterValue>`
  The initial value of the memory parameter. Often memory parameter values are modified during the course of the search.

- `<increaseType>`
  An optional parameter that is used in self-adaptive MLS heuristics where memory parameters are modified as "moves". Defines the type of change that is made to the parameter when an increase or decrease is called; either "Add" (the increase value is added to the current parameter value) or "Multiply" (the increase value is multiplied with the current parameter value).

- `<increaseValue>`
  An optional parameter that specifies how much the current memory parameter value should be increased by (either additively or multiplicatively).

- `<minValue>`
  An optional parameter than specifies the minimum value to which this memory parameter may be reduced .

- `<maxValue>`
  An optional parameter that specifies the maximum value to which this memory parameter may be increased.

- `<updateFunctions>`
  A container tag for the `<updateFunction>` tags (they must be contained in this tag, if present).

- `<updateFunction>`
  Contains the name of this memory-update function. This name must match that programmed into the UpdateMemory classes.

- `<triggers>`
  A container tag for the `<trigger>` tags.

- `<trigger>`
  Represents a trigger. Contains a number of child tags representing attributes of the trigger, and responses: `<active>`, `<triggerName>`, `<triggerType>`, `<threshold>`, `<triggerParameter>`, `<response>`.

- `<active>`
  Defines whether the trigger is initially active (*true*), or inactive (*false*).

- `<triggerName>`
  The name of the trigger. The name is used to reference this trigger internally, for example when a response is to deactivate a particular trigger, the parameter is the trigger name.

- `<triggerType>`
  The type of the trigger. This determines which trigger function is executed, and must match the *name* in the Trigger classes.

- `<threshold>`
  Optional. Specifies the memory parameter value for the trigger threshold. Not all triggers are comparisons against a threshold. An alternative formulation would be to simply have the thereshold as a `<memoryParameter>` item.

- `<triggerParameter>`
  Optional. If the trigger refers to another module, the name of the module is given here. For example, one trigger condition is if a number of iterations have occured since a particular trigger was last tripped. This name of the trigger is given here, and can be name of this trigger itself.

- `<response>`
  Represents a response that is executed when the trigger is tripped. Multiple `<response>` tags may be children of each `<trigger>`. Contains the following tags: `<responseName>`, `<responseType>`, and optionally, `<responseParameter>`.

- `<responseName>`
  The name of this response. This is purely a label for convenience and has no functional effect.

- `<responseType>`
  The type of response. Must match one of the response type in the Response classes.

- `<responseParameter>`
  If the response takes a parameter then it is specified here. For example if the response is to activate a trigger, then this would give the name of the trigger to be activated.

C.2 Examples of MLSML specifications

In the following sections we provide examples of the MLSML code used to specify the heuristics used in the experimentation. Only a single example of each type of heuristic is presented, to illustrate the usage of MLSML.

C.2.1 Steepest Ascent

```
<metaHeuristic>
    <name>
        SteepestAscent
    </name>
    <admissibilityCondition>
```

```
            Improving fitness and feasible OR infeasible but decreasing cost
        </admissibilityCondition>
        <fitnessFunction>
            Objective
        </fitnessFunction>
        <neighbourhoodScheme>
            <moveType>nAdd</moveType>
            <moveType>nDrop</moveType>
            <moveType>nShortcut</moveType>
            <moveType>nDetour</moveType>
        </neighbourhoodScheme>
        <memoryParameters>
            <memoryParameter>
                <parameterActive>true</parameterActive>
                <parameterName>NLookahead</parameterName>
                <parameterValue>4</parameterValue>
            </memoryParameter>
            <memoryParameter>
                <parameterActive>true</parameterActive>
                <parameterName>movelistSize</parameterName>
                <parameterValue>100000</parameterValue>
            </memoryParameter>
            <memoryParameter>
                <parameterActive>true</parameterActive>
                <parameterName>admissibleSize</parameterName>
                <parameterValue>100000</parameterValue>
            </memoryParameter>
            <memoryParameter>
                <parameterActive>true</parameterActive>
                <parameterName>maxNeighbours</parameterName>
                <parameterValue>100000</parameterValue>
            </memoryParameter>
        </memoryParameters>
        <triggers>
            <trigger>
                <active>true</active>
                <triggerName>Local optimum</triggerName>
                <triggerType>Local optimum</triggerType>
                <response>
                    <responseName>Termination</responseName>
                    <responseType>Termination</responseType>
                </response>
            </trigger>
        </triggers>
    </metaHeuristic>
```

C.2.2 Simulated Annealing

```
    <metaHeuristic>
        <name>
            SimulatedAnnealing
```

```
</name>
<admissibilityCondition>
    Annealing probability
</admissibilityCondition>
<fitnessFunction>
    Objective
</fitnessFunction>
<neighbourhoodScheme>
    <moveType>Add</moveType>
    <moveType>Drop</moveType>
    <moveType>Shortcut</moveType>
    <moveType>Detour</moveType>
</neighbourhoodScheme>
<memoryParameters>
    <memoryParameter>
        <parameterActive>true</parameterActive>
        <parameterName>movelistSize</parameterName>
        <parameterValue>100000</parameterValue>
    </memoryParameter>
    <memoryParameter>
        <parameterActive>true</parameterActive>
        <parameterName>admissibleSize</parameterName>
        <parameterValue>1</parameterValue>
    </memoryParameter>
    <memoryParameter>
        <parameterActive>true</parameterActive>
        <parameterName>maxNeighbours</parameterName>
        <parameterValue>100000</parameterValue>
    </memoryParameter>
    <memoryParameter>
        <parameterActive>true</parameterActive>
        <parameterName>annealingTemperature</parameterName>
        <parameterValue>1500</parameterValue>
    </memoryParameter>
    <memoryParameter>
        <parameterActive>true</parameterActive>
        <parameterName>coolingRate</parameterName>
        <parameterValue>0.99</parameterValue>
    </memoryParameter>
</memoryParameters>
<triggers>
    <trigger>
        <active>true</active>
        <triggerName>Iterations since last trigger</triggerName>
        <triggerType>Iterations since last trigger</triggerType>
        <threshold>80</threshold>
        <triggerParameter>Iterations since last trigger</triggerParameter>
        <response>
            <responseName>Reduce annealing temperature</responseName>
            <responseType>Reduce annealing temperature</responseType>
```

```
            </response>
        </trigger>
        <trigger>
            <active>true</active>
            <triggerName>Temperature threshold</triggerName>
            <triggerType>Temperature threshold</triggerType>
            <threshold>0.001</threshold>
            <response>
                <responseName>Termination</responseName>
                <responseType>Termination</responseType>
            </response>
        </trigger>
    </triggers>
</metaHeuristic>
```

C.2.3 Tabu Search

```
<metaHeuristic>
    <name>
        TabuSearch
    </name>
    <admissibilityCondition>
        Tabu arcs with aspiration
    </admissibilityCondition>
    <fitnessFunction>
        Objective
    </fitnessFunction>
    <neighbourhoodScheme>
        <moveType>Add</moveType>
        <moveType>Drop</moveType>
        <moveType>Shortcut</moveType>
        <moveType>Detour</moveType>
    </neighbourhoodScheme>
    <memoryParameters>
        <memoryParameter>
            <parameterActive>true</parameterActive>
            <parameterName>movelistSize</parameterName>
            <parameterValue>100000</parameterValue>
        </memoryParameter>
        <memoryParameter>
            <parameterActive>true</parameterActive>
            <parameterName>admissibleSize</parameterName>
            <parameterValue>100000</parameterValue>
        </memoryParameter>
        <memoryParameter>
            <parameterActive>true</parameterActive>
            <parameterName>maxNeighbours</parameterName>
            <parameterValue>100000</parameterValue>
        </memoryParameter>
        <memoryParameter>
            <parameterActive>true</parameterActive>
            <parameterName>tabuTenure</parameterName>
            <parameterValue>20</parameterValue>
        </memoryParameter>
    </memoryParameters>
    <updateFunctions>
        <updateFunction>Update tabu arcs</updateFunction>
    </updateFunctions>
    <triggers>
        <trigger>
            <active>true</active>
            <triggerName>Iteration count</triggerName>
            <triggerType>Iteration count</triggerType>
            <threshold>1000</threshold>
            <response>
```

```
                <responseName>Termination</responseName>
                <responseType>Termination</responseType>
            </response>
        </trigger>
    </triggers>
</metaHeuristic>
```

C.2.4 Variable Neighbourhood Search

```
<metaHeuristic>
    <name>
        VariableNeighbourhoodSearch
    </name>
    <admissibilityCondition>
        Improving fitness and feasible OR infeasible but decreasing cost
    </admissibilityCondition>
    <fitnessFunction>
        Objective
    </fitnessFunction>
    <neighbourhoodScheme>
        <moveType>Add</moveType>
        <moveType>Drop</moveType>
        <moveType>Shortcut</moveType>
        <moveType>Detour</moveType>
    </neighbourhoodScheme>
    <memoryParameters>
        <memoryParameter>
            <parameterActive>true</parameterActive>
            <parameterName>movelistSize</parameterName>
            <parameterValue>100000</parameterValue>
        </memoryParameter>
        <memoryParameter>
            <parameterActive>true</parameterActive>
            <parameterName>admissibleSize</parameterName>
            <parameterValue>100000</parameterValue>
        </memoryParameter>
        <memoryParameter>
            <parameterActive>true</parameterActive>
            <parameterName>maxNeighbours</parameterName>
            <parameterValue>100000</parameterValue>
        </memoryParameter>
        <memoryParameter>
            <parameterActive>true</parameterActive>
            <parameterName>NLookahead</parameterName>
            <parameterValue>8</parameterValue>
        </memoryParameter>
    </memoryParameters>
    <triggers>
        <trigger>
            <active>true</active>
            <triggerName>Local optimum</triggerName>
```

```
            <triggerType>Local optimum</triggerType>
            <response>
                <responseName>Change moveset to extended</responseName>
                <responseType>Change moveset</responseType>
                <responseParameter>Extended</responseParameter>
            </response>
            <response>
                <responseName>Deactivate trigger</responseName>
                <responseType>Deactivate trigger</responseType>
                <responseParameter>Local optimum</responseParameter>
            </response>
            <response>
                <responseName>Activate trigger</responseName>
                <responseType>Activate trigger</responseType>
                <responseParameter>Iterations since last trigger</responseParameter>
            </response>
        </trigger>
        <trigger>
            <active>false</active>
            <triggerName>Iterations since last trigger</triggerName>
            <triggerType>Iterations since last trigger</triggerType>
            <threshold>1</threshold>
            <triggerParameter>Iterations since last trigger</triggerParameter>
            <response>
                <responseName>Change moveset to basic</responseName>
                <responseType>Change moveset</responseType>
                <responseParameter>Basic</responseParameter>
            </response>
            <response>
                <responseName>Deactivate trigger</responseName>
                <responseType>Deactivate trigger</responseType>
                <responseParameter>Iterations since last trigger</responseParameter>
            </response>
            <response>
                <responseName>Activate trigger</responseName>
                <responseType>Activate trigger</responseType>
                <responseParameter>Local optimum</responseParameter>
            </response>
        </trigger>
        <trigger>
            <active>true</active>
            <triggerName>Iteration count</triggerName>
            <triggerType>Iteration count</triggerType>
            <threshold>1000</threshold>
            <response>
                <responseName>Termination</responseName>
                <responseType>Termination</responseType>
            </response>
        </trigger>
    </triggers>
```

```
</metaHeuristic>
```

C.2.5 ASRP Design

```
<metaHeuristic>
    <name>
        ASRPDesign
    </name>
    <admissibilityCondition>
        Connected and improving
    </admissibilityCondition>
    <fitnessFunction>
        Objective
    </fitnessFunction>
    <neighbourhoodScheme>
        <moveType>Swap arcs</moveType>
    </neighbourhoodScheme>
    <memoryParameters>
        <memoryParameter>
            <parameterActive>true</parameterActive>
            <parameterName>movelistSize</parameterName>
            <parameterValue>5000</parameterValue>
        </memoryParameter>
        <memoryParameter>
            <parameterActive>true</parameterActive>
            <parameterName>admissibleSize</parameterName>
            <parameterValue>1</parameterValue>
        </memoryParameter>
        <memoryParameter>
            <parameterActive>true</parameterActive>
            <parameterName>maxNeighbours</parameterName>
            <parameterValue>100</parameterValue>
        </memoryParameter>
    </memoryParameters>
    <triggers>
        <trigger>
            <active>true</active>
            <triggerName>Iteration count</triggerName>
            <triggerType>Iteration count</triggerType>
            <threshold>500</threshold>
            <response>
                <responseName>Termination</responseName>
                <responseType>Termination</responseType>
            </response>
        </trigger>
    </triggers>
</metaHeuristic>
```

C.2.6 MLS Design Control

```
<metaHeuristic>
    <name>
        MLSDesignControl
    </name>
    <admissibilityCondition>
        All admissible
    </admissibilityCondition>
    <fitnessFunction>
        Objective
    </fitnessFunction>
    <neighbourhoodScheme>
        <moveType>Add module</moveType>
        <moveType>Remove module</moveType>
        <moveType>Increase parameter</moveType>
        <moveType>Decrease parameter</moveType>
    </neighbourhoodScheme>
    <memoryParameters>
        <memoryParameter>
            <parameterActive>true</parameterActive>
            <parameterName>movelistSize</parameterName>
            <parameterValue>100</parameterValue>
        </memoryParameter>
        <memoryParameter>
            <parameterActive>true</parameterActive>
            <parameterName>admissibleSize</parameterName>
            <parameterValue>5</parameterValue>
        </memoryParameter>
        <memoryParameter>
            <parameterActive>true</parameterActive>
            <parameterName>maxNeighbours</parameterName>
            <parameterValue>300</parameterValue>
        </memoryParameter>
    </memoryParameters>
    <triggers>
        <trigger>
            <active>true</active>
            <triggerName>Local optimum</triggerName>
            <triggerType>Local optimum</triggerType>
            <response>
                <responseName>Termination</responseName>
                <responseType>Termination</responseType>
            </response>
        </trigger>
        <trigger>
            <active>true</active>
            <triggerName>Iteration count</triggerName>
            <triggerType>Iteration count</triggerType>
            <threshold>80</threshold>
```

```
            <response>
                <responseName>Termination</responseName>
                <responseType>Termination</responseType>
            </response>
        </trigger>
    </triggers>
</metaHeuristic>
```

C.2.7 MLS Design – ASRP Template

```
<metaHeuristic>
    <name>
        ASRPTemplate
    </name>
    <admissibilityCondition>
        All admissible
    </admissibilityCondition>
    <fitnessFunction>
        Objective
    </fitnessFunction>
    <neighbourhoodScheme>
        <moveType>Add</moveType>
        <moveType>Drop</moveType>
        <moveType>Shortcut</moveType>
        <moveType>Detour</moveType>
    </neighbourhoodScheme>
    <memoryParameters>
        <memoryParameter>
            <parameterActive>true</parameterActive>
            <parameterName>movelistSize</parameterName>
            <parameterValue>1000</parameterValue>
        </memoryParameter>
        <memoryParameter>
            <parameterActive>true</parameterActive>
            <parameterName>admissibleSize</parameterName>
            <parameterValue>1</parameterValue>
            <increaseType>Add</increaseType>
            <increaseValue>3</increaseValue>
            <minValue>1</minValue>
            <maxValue>1000</maxValue>
        </memoryParameter>
        <memoryParameter>
            <parameterActive>true</parameterActive>
            <parameterName>maxNeighbours</parameterName>
            <parameterValue>100</parameterValue>
            <increaseType>Add</increaseType>
            <increaseValue>5</increaseValue>
            <minValue>6</minValue>
            <maxValue>1000</maxValue>
        </memoryParameter>
        <memoryParameter>
```

```
            <parameterActive>false</parameterActive>
            <parameterName>annealingTemperature</parameterName>
            <parameterValue>1000</parameterValue>
            <increaseType>Multiply</increaseType>
            <increaseValue>1.2</increaseValue>
            <minValue>100</minValue>
            <maxValue>2500</maxValue>
        </memoryParameter>
        <memoryParameter>
            <parameterActive>false</parameterActive>
            <parameterName>coolingRate</parameterName>
            <parameterValue>0.75</parameterValue>
            <increaseType>Multiply</increaseType>
            <increaseValue>1.1</increaseValue>
            <minValue>0.1</minValue>
            <maxValue>0.99</maxValue>
        </memoryParameter>
        <memoryParameter>
            <parameterActive>false</parameterActive>
            <parameterName>tabuTenure</parameterName>
            <parameterValue>7</parameterValue>
            <increaseType>Add</increaseType>
            <increaseValue>1</increaseValue>
            <minValue>3</minValue>
            <maxValue>30</maxValue>
        </memoryParameter>
        <memoryParameter>
            <parameterActive>false</parameterActive>
            <parameterName>NLookahead</parameterName>
            <parameterValue>4</parameterValue>
            <increaseType>Add</increaseType>
            <increaseValue>1</increaseValue>
            <minValue>2</minValue>
            <maxValue>10</maxValue>
        </memoryParameter>
    </memoryParameters>
    <triggers>
        <trigger>
            <active>false</active>
            <triggerName>Simulated annealing temperature reduction</triggerName>
            <triggerType>Iterations since last trigger</triggerType>
            <threshold>12</threshold>
            <triggerParameter>
                Simulated annealing temperature reduction
            </triggerParameter>
            <response>
                <responseName>Reduce annealing temperature</responseName>
                <responseType>Reduce annealing temperature</responseType>
            </response>
        </trigger>
```

```
<trigger>
    <active>false</active>
    <triggerName>
        Simulated annealing temp threshold termination
    </triggerName>
    <triggerType>Temperature threshold</triggerType>
    <threshold>0.001</threshold>
    <response>
        <responseName>Termination</responseName>
        <responseType>Termination</responseType>
    </response>
</trigger>
<trigger>
    <active>true</active>
    <triggerName>Default termination</triggerName>
    <triggerType>Iteration count</triggerType>
    <threshold>1000</threshold>
    <response>
        <responseName>Termination</responseName>
        <responseType>Termination</responseType>
    </response>
</trigger>
<trigger>
    <active>false</active>
    <triggerName>VNS phase 1 trigger</triggerName>
    <triggerType>Local optimum</triggerType>
    <response>
        <responseName>Change moveset to extended</responseName>
        <responseType>Change moveset</responseType>
        <responseParameter>Extended</responseParameter>
    </response>
    <response>
        <responseName>Deactivate trigger</responseName>
        <responseType>Deactivate trigger</responseType>
        <responseParameter>VNS phase 1 trigger</responseParameter>
    </response>
    <response>
        <responseName>Activate trigger</responseName>
        <responseType>Activate trigger</responseType>
        <responseParameter>VNS phase 2 trigger</responseParameter>
    </response>
</trigger>
<trigger>
    <active>false</active>
    <triggerName>VNS phase 2 trigger</triggerName>
    <triggerType>Iterations since last trigger</triggerType>
    <threshold>1</threshold>
    <triggerParameter>VNS phase 2 trigger</triggerParameter>
    <response>
        <responseName>Change moveset to basic</responseName>
```

```
                <responseType>Change moveset</responseType>
                <responseParameter>Basic</responseParameter>
            </response>
            <response>
                <responseName>Deactivate trigger</responseName>
                <responseType>Deactivate trigger</responseType>
                <responseParameter>VNS phase 2 trigger</responseParameter>
            </response>
            <response>
                <responseName>Activate trigger</responseName>
                <responseType>Activate trigger</responseType>
                <responseParameter>VNS phase 1 trigger</responseParameter>
            </response>
        </trigger>
    </triggers>
</metaHeuristic
```

C.2.8 Adaptive Diversification Local Search

```
<metaHeuristic>
    <name>
        ADLS
    </name>
    <admissibilityCondition>
        Improving fitness and feasible
    </admissibilityCondition>
    <fitnessFunction>
        Objective
    </fitnessFunction>
    <neighbourhoodScheme>
        <moveType>Add</moveType>
        <moveType>Drop</moveType>
        <moveType>Shortcut</moveType>
        <moveType>Detour</moveType>
    </neighbourhoodScheme>
    <updateFunctions>
        <updateFunction>Update tabu arcs</updateFunction>
    </updateFunctions>
    <memoryParameters>
        <memoryParameter>
            <parameterActive>true</parameterActive>
            <parameterName>NLookahead</parameterName>
            <parameterValue>4</parameterValue>
        </memoryParameter>
        <memoryParameter>
            <parameterActive>true</parameterActive>
            <parameterName>movelistSize</parameterName>
            <parameterValue>100000</parameterValue>
        </memoryParameter>
        <memoryParameter>
            <parameterActive>true</parameterActive>
```

```
            <parameterName>admissibleSize</parameterName>
            <parameterValue>100</parameterValue>
        </memoryParameter>
        <memoryParameter>
            <parameterActive>true</parameterActive>
            <parameterName>maxNeighbours</parameterName>
            <parameterValue>100000</parameterValue>
        </memoryParameter>
        <memoryParameter>
            <parameterActive>true</parameterActive>
            <parameterName>diversificationAlpha</parameterName>
            <parameterValue>6</parameterValue>
        </memoryParameter>
        <memoryParameter>
            <parameterActive>true</parameterActive>
            <parameterName>annealingTemperature</parameterName>
            <parameterValue>500</parameterValue>
        </memoryParameter>
        <memoryParameter>
            <parameterActive>true</parameterActive>
            <parameterName>tabuTenure</parameterName>
            <parameterValue>15</parameterValue>
        </memoryParameter>
    </memoryParameters>
    <triggers>
        <trigger>
            <active>true</active>
            <triggerName>Local optimum</triggerName>
            <triggerType>Local optimum</triggerType>
            <response>
                <responseName>Start diversification phase</responseName>
                <responseType>Start diversification phase</responseType>
            </response>
        </trigger>
        <trigger>
            <active>false</active>
            <triggerName>Iterations since last trigger</triggerName>
            <triggerType>Iterations since last trigger</triggerType>
            <threshold>3</threshold>
            <triggerParameter>Local optimum</triggerParameter>
            <response>
                <responseName>End diversification phase</responseName>
                <responseType>End diversification phase</responseType>
            </response>
        </trigger>
        <trigger>
            <active>true</active>
            <triggerName>Iteration count</triggerName>
            <triggerType>Iteration count</triggerType>
            <threshold>250</threshold>
```

```
            <response>
                <responseName>Termination</responseName>
                <responseType>Termination</responseType>
            </response>
        </trigger>
    </triggers>
</metaHeuristic>
```

APPENDIX D

Discussion of Possible Extensions to the ASRP

In this appendix we explore some variations that could be applied to extend the ASRP. This discussion was an early part of the research, when the development of MLS had not yet been started, and one of the possible research directions was further development of the ASRP. It has been moved to an appendix because it is no longer directly relevant to the main research investigation, however may provide some ideas for future research.

D.1 Introduction

Subset selection has been extensively covered in the node routing literature, but as far as the author is aware has yet to be systematically explored in the field of arc routing. We attempt to fill that void. We begin by a discussion of what aspects of a routing problem could be modelled within a subset selection framework, and then attempt to formalize the definition of some key problems within this field, which we call *subset selection arc routing* (SSAR).

We first define an **arc routing problem** (ARP) to be a problem where there exists a graph, or network, G, such that one or more vehicles are required to traverse the arcs of G, subject to some constraints.

Given this definition, an Arc Subset Routing Problem (ASRP) will therefore be defined as an arc routing problem, such that in a solution, only a subset of the arcs of G need be traversed. There are many different possible variations of this basic structure, which will duly be considered, but one of the most fundamental questions to be explained is *why* the subset selection aspect needs to be introduced.

This may best be explained with the introduction of two concepts: rewards and costs. Both concepts are central to existing routing models. The concept of cost is usually associated, in arc routing literature, with the length of an arc; a related objective is to minimize either the distance travelled over the whole solution, or the time taken to do so. Reward is introduced to make some arcs (or customers, in node routing literature) more attractive than others. Subset selection problems naturally arise in two cases: where there is a limited cost (time or distance) budget, and the maximum reward must be collected within the confines of this budget, or where there is a minimum reward that must be collected, and a minimum cost must be incurred during the collection. We call the former case a *set cost, max reward*

problem, and the latter case a *set reward, min cost* problem. The former problem is actually a *capacitated* problem. Introducing capacities traditionally gives rise to multiple routes to satisfy demand (as in the Vehicle Routing Problem (VRP)); we consider such a problem where only one of these routes may be completed. Perhaps this implies that judicious truncation of certain vehicle routing heuristics could provide efficient solution methods.

Interestingly, these two types of problems are actually the same problem; one can be transformed into the other. We consider that there are two quantities associated with each arc: cost and reward. To transform a set reward, min cost problem into a set cost, max reward problem, we simply let *reward* ← *-cost* and *cost* ← *-reward*. Minimizing the cost now translates to minimizing the negative reward, i.e., maximizing the reward, and having a set reward translates to having a set cost. Therefore, we could consider only one of the variations, with no loss of generality. However, there are some possible variations which make more sense one way or the other; for example, time variant rewards make more intuitive sense than time variant negative costs. So, we will consider both problem formulations.

Section D.1.1 defines a basic version of an arc subset routing problem, which is used as a point of comparison throughout the remainder of this chapter. Sections 0 through D.5 explore possible variations that could be applied to extend the basic problem. It is important to note that these sections are in the nature of a brain-storming exercise; most of these variations are not incorporated in the ASRP problem that is used throughout the remainder of the thesis. Section 3.1 explores various mathematical programming formulations for the ASRP.

D.1.1 The basic problem

We define a basic problem, and then consider what modifications we could apply to make the problem more interesting. In some sense, it doesn't matter *which* problem we define to be our basic problem; whether problem A is the basic problem and problem B the variation, or whether problem B is the basic problem and problem A the variation is immaterial.

We define the **Basic Arc Subset Routing Problem** (BASRP) on a network, *G*, such that for each arc on the network there is an associated cost for traversing the arc, and a reward for traversing the arc. We have a single vehicle, stationed at a pre-located depot, which is to service some subset of the arcs on the network, forming a route. The objective is to maximize the reward collected, while not incurring more cost than some pre-specified limit, or budget.

D.2 Variations on the Basic ASRP

D.2.1 Multiple-vehicle variations

If we allow multiple vehicles within our framework, then the field widens considerably. We can define subset-selective versions of all the vehicle routing problems in the node routing literature.

D.2.1.1 The Multi- Selective Arc Routing Problem (MASRP)

We begin by defining the closest analogue with the Selective Arc Routing Problem, the **Multi- Arc Subset Routing Problem** (MASRP). In this variation there exists a fleet of vehicles, of size *k*, with which to do our reward collection. A solution will, therefore, consist of a set of *k* routes. For this problem, we assume that all the vehicles are domiciled at the same depot, although a generalization would be the case where each vehicle has its own depot (which may or may not be the same as another vehicle's depot). In the simplest case of the MASRP, the vehicle fleet would be homogeneous, i.e. all vehicles have the same cost budget, as well as cost and reward structure. Variations would be where the vehicles have different cost budgets (influencing route size) and/or reward and cost structures (influencing route selection).

We made the analogy earlier between the BASRP and the VRP, where the BASRP may be considered to select one of the routes created in a VRP solution. Similarly, the MASRP may be considered to take *k* of the routes created in a VRP solution.

D.2.1.2 The capacitated MASRP

In some sense the BASRP and the MASRP are both capacitated problems, in that there is a restriction on the cost that may be incurred by a single vehicle. We now introduce a variation which is not only cost-restricted, but also reward-restricted; this problem is a generalization of the MASRP. In this problem, which we call the **Capacitated Multi- Arc Subset Routing Problem** (CMASRP), a vehicle may incur no more cost than its cost budget, and may collect no more reward than its reward capacity. There are two ways of modelling this additional restriction: in the first model, the restriction forces the routes to be shorter; a vehicle may not traverse an arc if it does not have the capacity to collect its reward. In the second model, a vehicle may *traverse* an arc without collecting its reward, and so may collect reward until its capacity is reached and then return to the depot without collecting any more (but still incurring cost, which must stay within the budget). This latter model introduces the concept of *service variation*, which we will consider in greater detail subsequently, but which has parallels in vehicle routing literature, i.e., where *split service* is allowed. Split service is where a vehicle may collect *some* of the reward from an arc (service variation), leaving the remainder for another vehicle (or leaving it to be uncollected). Allowing split service would help to fill the vehicles to complete capacity, eliminating wasted capacity. We will see later that split service is not exactly the same thing as service variation, since when service variation is used to collect a fraction of the reward a corresponding fraction of the cost is incurred.

D.2.2 Penalties and requirements

There are many possible variations on the BASRP which arise from slightly different objectives and additional constraints.

D.2.2.1 Non-selection penalties

It is a natural extension, especially in a customer service framework, to incur a penalty for not servicing an arc; this penalty could correspond to a loss of customer goodwill. Then, we could introduce another

constraint that does not allow such penalties to exceed a certain limit, or we could modify the cost constraint to include penalties in the total cost evaluation.

After introducing penalties for non-inclusion, it is only a small step to make some arcs *compulsory*, and some arcs optional; a compulsory arc would have an arbitrarily large non-inclusion penalty. Another way of modelling compulsory arcs would be to add a constraint such that all the compulsory arcs are *required* to be included:

$$\sum_{(v_i, v_j) \in A^C} x_{ij} \geq 1$$

where A^C is the set of compulsory edges. Similarly, *preferences* for the inclusion/non-inclusion of arcs can be specified by the adjustment of penalties, costs, and rewards.

D.2.2.2 Pickup and delivery

We can modify the reward structure to include pickups and deliveries; this is really defining two types of service. There are many ways of modelling this. One method is to have each arc having a "demand". If this demand is positive, then the arc requires a delivery; if the demand is negative, then the arc has surplus product to be "picked up". A vehicle would then have a "stock level", and enough product to be delivered to the delivery customers must be present in the vehicle before delivery. The simplest case would be where there is one homogeneous product, which can be picked up from one customer and delivered to another. A more complicated case involves the existence of several different product types, and a careful schedule must be maintained, so that each customer gets the type of product he requires, and the vehicle's capacity for product is not exceeded.

Variations on this problem abound; the depot can either be merely the starting place, or an unlimited source of extra product. Of course, this problem immediately lends itself to the addition of penalties for unsatisfactory service.

D.2.3 Depot location

There are two options for depot location: the depot can be pre-specified, or the depot location can be chosen as part of the solution; either way is simple to model. The more interesting case is where the location of the depot is a separate problem.

D.3 Reward structures

Reward structures can be simple or complicated. Varying the reward structure is one of the aspects of subset selection arc routing that offers the most interesting avenues for study. Such reward structures can be fitted to many real world applications, or can be highly artificial to provide interesting problems. We refer to "reward structures", although, in fact, we mean "reward and cost structures"; both will have similar impacts on the type of problem. Unless stated otherwise, in the problems described below we use a simple cost structure.

D.3.1 Static reward structures

The simplest reward structure is where all arcs have the same reward. Assuming costs are the same this problem is trivial, since all arcs are equally attractive. If costs vary, then the cost will be the deciding factor.

We now define a *static* reward structure to be such that the reward for a given arc remains the same until it is collected, and then it is reduced. Typically, it will be reduced to zero; this corresponds to a prize-collection problem, i.e. once the reward is collected it is gone. Variations could be that the reward is reduced by a certain fraction, or by a certain amount; thus, more reward can be collected from an arc by traversing it repeatedly. Usually the rewards would be diminishing, but a special case would be when the reward does *not* decrease, and may be collected in full multiple times. These are all **static-reward problems** (because the reward does not vary by itself), the first sort we call **static single-reward problems** and the second sort **static multiple-reward problems**.

D.3.2 Dynamic reward structures

We define a dynamic reward structure to be one where the reward for an arc changes without the arc being traversed. There are endless possible variations of dynamic reward structures; we consider some of the more obvious ones. We broadly group them into **time-dependent reward structures** and **state-dependent reward structures**. We define an arc's *reward function* to be the way it changes. Time-dependent structures will incorporate a function of time, and state-dependent structures will incorporate a function of the network state.

D.3.3 Time-dependent reward structures

Arcs which have time-dependent rewards are *time-variant*; the reward that is collected for the traversal of an arc depends on *when* the arc is traversed. This involves the introduction and modelling of a new concept, time, and has several effects on the way such problems will be investigated and studied. The first effect is that it will matter in which direction the route is traversed; in most applications this makes no difference, but the concept is not completely new, since in problems where the network has directed arcs (arcs that can only be traversed in one direction) this is also the case. Another effect will be that simple insertions (adding an arc somewhere in the middle of a route) have the potential to make comparatively huge changes to the objective function evaluation, since all the traversal times and dependent rewards must then be recalculated. With respect to modelling time, we must also make the decision whether time and cost will be the same thing, and therefore define the cost of traversing an arc to be the same as the time to traverse it, or whether time and cost will be separate. If time and cost are separate, then we have, in effect, two different types of cost. We must then decide if time also has a budget, or if it merely affects the reward structure. In our view, it serves no purpose to have both 'time' and 'cost' separate, since we believe that there will always be some new cost structure which is a combination of 'time' and 'cost', and which results in effectively identical problems. So, in the following pages, we assume that the time to traverse an arc and the cost of doing so are synonymous.

D.3.3.1 Linear reward functions

Given that we have a time-dependent reward structure, there are many available options. The simplest is where the reward function for an arc is a linear function of time, either increasing or decreasing. An increasing reward function creates a dichotomy. We want to collect the rewards when they are at their highest, but we have a time budget that we must not exceed. This is an interesting problem, especially if the coefficient of increase is different for each arc. If the coefficient is the same for every arc, then the problem is really no different; we would still choose to traverse the same arc as if rewards were static, since arc A would have increased by the same amount as arc B. We define the **Arc Subset Routing Problem with Increasing Rewards** (ASRPIR). This is the same as the Selective Arc Routing Problem we defined in Section D.1.1, but with the addition of linearly increasing rewards for the arcs. Each of the arcs potentially has a different coefficient of increase; the ASRPIR is a dynamic problem, and hence will result in unpredictable outcomes.

We may also consider the case where the rewards are *decreasing* linearly, and define the **Arc Subset Routing Problem with Decreasing Rewards** (ASRPDR). Again, each of the arcs will have a different negative coefficient of increase, and again, this is a dynamic problem and the behaviour of heuristics cannot be predicted.

A generalization which includes both the ASRPIR and the ASRPDR is where the reward function for an arc can either be increasing or decreasing. We define the **Arc Subset Routing Problem with Linear-Dynamic Rewards** (ASRPLDR). Each of the previous two problems is a special case; the ASRPIR has the requirement that the coefficient of increase is positive, hence,

$$\text{let } \alpha_{ij} \text{ be the coefficient of increase for arc } v_{ij}, \text{ then } \alpha_{ij} \in \Re^{+},$$

where $\Re^{+}$ is the set of positive real numbers. The ASRPDR similarly has the requirement that the coefficient of increase is negative, hence,

$$\text{let } \alpha_{ij} \text{ be the coefficient of increase for arc } v_{ij}, \text{ then } \alpha_{ij} \in \Re^{-},$$

where $\Re^{-}$ is the set of negative real numbers. The ASRPLDR is simply a generalization of the two, which will also include the possibility that an arc may have a static coefficient of increase, hence,

$$\text{let } \alpha_{ij} \text{ be the coefficient of increase for arc } v_{ij}, \text{ then } \alpha_{ij} \in \Re,$$

where $\Re$ is the set of real numbers. We can see that the static-reward problems discussed above are just special cases of the ASRPLDR where all the coefficients of increase are zero. For the three problems considered above, we also have the option of single-rewards or multiple-rewards. Single rewards are gone when they are collected, multiple rewards reappear, possibly with a changed reward function. We call the starting value and the reward function of an arc its *initial* state, and if it is regenerated, it is *reinitialized* to a new starting value and reward function (which may be the same as the old one). We list briefly some of the possible options for multiple linear rewards. There are two aspects to vary: the new *starting value* and the new *reward function*. These can then be combined in any chosen manner. First, the starting value options:

- The arc is reinitialized to its original starting value.
- The arc is reinitialized to zero.
- The arc is reinitialized to some other starting value, perhaps some fraction of the initial starting value, perhaps some fraction of the reward level of the arc at collection, or perhaps some absolute amount less.

Now we consider the options for the reward function. For the ASRPLDR, a different reward function corresponds to a different coefficient of increase:

- The coefficient of increase remains the same. This is the natural scenario for applications such as snow falling.
- The coefficient of increase is set to zero. If this is the case for all the arcs, and the starting value is also reinitialized to zero, then we have the single reward case.
- The coefficient of increase changes, perhaps increases/decreases by some fraction, or absolute amount, perhaps changes sign.

Each combination of these options results in a unique problem with different characteristics.

D.3.3.2 Non-linear reward functions

Non-linear reward functions. The next obvious extension is to allow the reward function to be *non-linear*. There are endless possibilities here, but we attempt to categorize them:

- The reward function could be increasing or decreasing, but instead of linearly, the rate of increase could be accelerating/decelerating, the actual function would still be a function of time, but it might be an exponential or parabolic function instead.
    - The reward function could have peaks, or troughs. This implies that there is an optimum time period to traverse an arc, and shares many features with time-windows, which are discussed below. Sine functions would be the obvious choice for this option.
    - Any other artificial function: polynomial functions, trigonometric functions, exponential functions, combinations, etc.

Step-wise reward functions. We define *step-wise reward functions* to be functions where the type of function changes at some point, or more than one point. Functions which change upon collection are one realization of this, and we have discussed these briefly. Here we consider functions which change either at a certain point in time, or when the reward for an arc reaches a certain level. We list some of the possibilities below:

- We define a *reward cap* to be a limit to how far the reward for an arc may change. The first possibility is where the reward function becomes static when the reward cap is reached. For example, if teamed with a linearly increasing function, we might have a case where the reward

for an arc starts out at, say, 10 then increases at 1 unit per unit time until it reaches 25, and then stays at 25.

- The reward cap can also signal a change in the reward function, for example, the reward function may cause the reward to accelerate up to a certain point, then linearly increase to another point, and then become static. Or perhaps the reward could increase linearly up to a point, and then decrease linearly down to zero (from there it could change again, or stay at zero).
- Instead of having a reward cap, we could have a step-wise function where the reward is static, but changes at certain times. For example, the reward could increase by one unit after every ten units of time. This type of function could be teamed with any of the other types; within one 'step' the function can vary as desired.

Time windows. These share many similarities with step-wise functions, but rate a separate section because they have such an important role in vehicle routing problems. We model time windows by having a stepwise function where before and after the 'time window' the reward is less. Time windows are especially interesting when teamed with penalties. These penalties may be of two types: negative rewards, in which case it would seem prudent *not* to traverse the arc at all (unless there is a requirement to traverse the arc, or the penalty for not traversing the arc is even worse than that for missing the time window). The other type is a time penalty, whereby the vehicle is forced to wait while the customer processes an out-of-window traversal.

Weather / acts of God. We now consider some more advanced forms of the model, which involve complicated effects. We include them for completeness. We list some alternatives below, but this list is by no means exhaustive.

There may be an 'agent' or 'agents' of the model which travel around the network modifying the reward structures of the arcs within their influence. These *weather agents* would be independent of the current state of the graph and location of the vehicle. They could travel around the graph either randomly, or according to certain rules (algorithmically). The influences they have on the network reward structures could either disrupt the reward levels permanently (for example if a weather agent causes the reward for an arc to increase tenfold then when it passes the reward will still be tenfold, and keep being modified from there by its function), or temporarily (for example the reward for an arc could be multiplied by ten for duration of the weather agents influence, and then resort to what it would have been if the weather agent had not influenced it). The weather agents could similarly modify the cost structures, making arcs more or less expensive to traverse.

There is an important concept which we introduce here and discuss more thoroughly in the section on competition: information. So far, we have assumed that the vehicle is omniscient; it knows everything about the problem. However, this is not necessarily true. Perhaps the vehicle knows only the initial reward values, or it might also know the reward functions. It might not know about the weather, or it might simply know the probabilities of its behaviour. In the section on competition we consider the implications of this information, and situations when the vehicle can 'buy' additional information. It is

worth noting here, though, that in dynamic problems, we separate the vehicle and ourselves; we are not so much creating routes for the vehicle as providing him with strategies and rules with which to design his own routes, and to modify them if necessary.

The other option for time-dependent reward structures, which is similar to weather, is that of *acts of god*. These are where the reward structures for certain arcs (perhaps those within a set radius of a certain point) are modified somehow, at either a random point in time, or a random point in space (or both). If the point in time and space is known beforehand, then we can model the situation with a step-wise function. In terms of information, the vehicle might know that such an event will occur, but not know when, or where, or it might have no idea and be forced to modify its route at the time.

D.3.4 State-dependent reward structures

State-dependent reward structures have the same complications as time-dependent reward structures – we must keep track of route direction, and everything must be recalculated when a change is made – but that is the nature of dynamic problems and is unavoidable. State-dependent reward structures vary the rewards of arcs depending upon the current state of the network, rather than time. The sections below explore some of the options.

D.3.4.1 Group-dependent rewards

We can imagine an application such that only one arc in a group of arcs *needs* to be traversed, and then the others become not as important, for example if a vehicle was clearing streets of snow, then as long as there is one road into and out of a suburb, that is sufficient. We model this by introducing the concept of *groups*. As soon as one of the arcs in the group is traversed, then the rewards of the others are modified. Perhaps there could also be some modification in the rewards of other groups. We could either have the requirement that arcs may only belong to one group, or allow them to belong to more than one group. The modification could be to increase *or* increase the rewards of other arcs in the group. Not all arcs need be in a group, for example maybe only the main arterial routes would be included.

D.3.4.2 Precedence constraints

Another concept which has seen some interest in the node-routing literature is that of *precedence constraints*. This can be modelled in three ways: constraints based on sets and all of one set being traversed before any of the next set, penalties given for traversing an arc out of turn, or using state-dependent reward structures. It is this last option we concentrate on. The first set of arcs to be traversed would have high rewards, and the other arcs low rewards (or zero rewards) until all the class 1 arcs are traversed, and then the class 2 arcs would have increased rewards, etc. Penalties could also be teamed with this formulation.

D.3.5 Reward structures that are both time- and state- dependent

We can also have reward functions that are functions of both time and state. These would simply be combinations of the problem aspects considered above.

D.4 Service variation

We define a problem to allow *service variation* if there is more than one way a vehicle may traverse an arc, thereby incurring less cost and/or greater reward.

D.4.1 Allowing traversal without service

This is the simplest realization of service variation. We separate the concepts of *traversal* and *service* that, until now, we had treated as synonymous. This means that a vehicle may simply traverse an arc, collecting no reward, or also service the arc, collecting the reward. The cost for the latter option would naturally be more than for the former. We could model this in two ways:

- We could have two different costs: the *cost of traversal* and the *cost of service*. When a vehicle merely traverses an arc it incurs the cost of traversal, and when it also services the arc it incurs the cost of service instead.
- We could again have two different costs: the *cost of traversal* and the *cost of service*. However, since a vehicle which services an arc also traverses it, in this formulation the cost of traversal is incurred for both cases, and the cost of service is an *additional* cost, which is incurred only when the vehicle services the arc.

The latter method seems to be more versatile, and can model all the same situations as the former method (by making the cost of service the old cost of service minus the cost of traversal), so we recommend using the second method.

We define the **Arc Subset Routing Problem with 2-Service Variation** (ASRP2SV) to be the Selective Arc Routing Problem, with the addition of the above option: that a vehicle may choose to service an arc, or merely traverse it without service. We introduce two new parameters: for arc v_{ij}, the cost of traversal is c_{ij}^T and the cost of service is c_{ij}^S. The total cost for servicing arc v_{ij} is then the sum of c_{ij}^T and c_{ij}^S. Of course, if the vehicle does not service the arc, then it does not collect the reward. We also introduce another binary variable. For the BASRP, x_{ij} is 1 if arc v_{ij} is traversed, and 0 otherwise. For the ASRP2SV, we keep this variable, but also define s_{ij} equal to 1 if arc v_{ij} is serviced, and 0 otherwise. Of course, an arc cannot be serviced without being traversed, so we also need the following constraint .

$$s_{ij} \leq x_{ij} \qquad (v_i \in A, v_j \in \delta(i))$$

where s_{ij} and x_{ij} are both binary variables.

D.4.2 Discrete service variation

The next step is to allow a vehicle to service an arc at a fraction of full service. The previous section considered the case where a vehicle could either traverse an arc at zero service (just traversal) or full service, let us code this problem as $S = 2$, since there are two options.

D.4.2.1 Half-service

We now consider the case where $S = 3$, so there are three options: zero service (just traversal), full-service, and half-service.

Given that we are allowing half-service, we must now decide *what is* half service? We regard 'servicing' an arc to be synonymous with collecting its reward. Let us continue with that idea and say that *half*-servicing an arc is collecting half its reward, giving rise to two questions:

- How will we deal with costs? Should the cost for half-servicing an arc be simply half the cost for fully servicing the arc, or should we model the costs in another way?
- How will we deal with the remaining reward? We reconsider some of the ideas for dynamic reward structures to see if they apply.

D.4.2.2 Costs of half-service

The easiest way is simply to say that the cost for half-service is half the cost of full service. The other way of doing it is to make the cost some proportion of the full cost, or some absolute amount less than the full cost. Any of these methods is easy to model. If we think of applications, then perhaps there should be some fixed cost component to servicing the arc (analogous to the time it takes to set up the servicing equipment), and then the additional amount could be halved, so the cost for half-servicing an arc v_{ij} would be

$$\frac{c_{ij}^{S} - \lambda}{2} + \lambda$$

where λ is the fixed cost of setting up service.

It seems reasonable that instead of having an absolute service cost, rather the service cost should be a function of the reward. This would make collection consistent. Say we half-service an arc, collecting half the reward and paying half the service cost. If we then return to that arc and collect the remaining reward, we are collecting the same amount of reward, but are instead paying twice the cost (the full service cost). To remedy this we can either make service cost a function of reward (leaving traversal cost constant for each arc), or subtract the service cost already paid from the service cost for that arc. If we choose the latter option, then for dynamic reward structures it is possible that the reward would have regenerated up to the point it was at the previous collection, but we would only have to pay half the cost of last time – another inconsistency. The only consistent option is to make service cost a function (not necessarily linear) of reward.

Again, there are two options. We can make service cost a function of the full reward for the arc, and then use whatever rule we want to choose a fraction of this, as above. Or, we can make the cost a function of the reward collected, and only calculate this one amount.

D.4.2.3 Remaining rewards after half-service

After we have half-serviced an arc, and collected half its reward, there is still half the reward remaining. We have several options as to how to treat this remaining reward.

- The reward is reduced to zero, and may not be collected again. This is analogous to the *single reward* cases that we considered in Section D.3.
- The reward remains at the point it was when collection occurred, even if there was a dynamic reward structure operating. This reward is collectable again.
- The reward is reinitialized to its starting value, and any dynamic reward structure continues operating.
- The remaining reward is available for recollection, and any dynamic reward structure continues operating.
- A different reward structure is introduced, either on the remaining reward, or on a reinitialized, or zeroed, reward.

D.4.2.4 Fractional service

Half-service is the case where $S = 3$; there are three options for service. We now generalize this idea to the fractional-service case. For any problem we simply specify S. For example, if $S = n$, then there are n+1 options for service.

The costs are, similarly, modelled the same way as for half-service. Either the service cost is some function of the reward collected, or it is a fraction of the service cost for full service (which is a function of the full reward).

D.4.3 Continuous service variation

In the previous section we considered discrete service levels. Another possibility is *continuous service variation*, which allows continuously variable service levels. The vehicle may then choose at what fraction to service an arc – 0.456 if desired. If teamed with cost functions based on reward, then this presents no additional difficulties. It is effectively equivalent to discrete service variation with S very large, say 1000.

D.5 Competition

We define competition to be where there are two, or more, vehicles that are independent. This is not the same case as the multiple vehicle problems because there the vehicles are adding reward to a common pool, with competition they are each trying to maximize their own reward, perhaps each at the expense of the others.

The number of potential problems is vast, and is limited only by the imagination. We consider some that seem fundamental, and leave an in depth exploration to the future. When we say competition we are really considering three scenarios:

- Competition – the vehicles are purely competitive, with strategies to advance their own wealth at the expense of their opponents. Objectives include maximizing their own wealth and maximizing the difference between their own and their opponent's wealth.
- Cooperation – the vehicles are still independent, but work together to achieve the best result for both. This scenario is closest to multiple-vehicle routing. Objective is to maximize both vehicles wealth.
- Coopetition – involves aspects of both competition and cooperation. Vehicles are trying to maximize their own wealth, but sometimes the best way to do this is to cooperate. Still purely selfish, but also rational.

Of course, all the problem aspects that have been considered so far may be included in competitive problems. The problems are especially interesting with dynamic reward structures. In the case of multiple vehicles we have teams; each team has a specified number of vehicles and works together as in the non-competitive problems.

It is useful to think of a competitive problem instance as a *game*. Not strictly in the game-theoretic sense, but in terms of having an opponent and strategies to 'beat' them.

D.5.1 Basic competitive problem

We start with a simple competitive problem, with two competing vehicles and static reward structures. Let us define the **Competitive Arc Subset Routing Problem** (CASRP) to be the selective arc routing problem with the addition of another, competing, vehicle. Each vehicle has its own depot node, and all information is known to each vehicle regarding its opponent and the problem.

Competitive problems are inherently dynamic; we do not design a route a priori, we instead give the vehicles *strategies* which guide their progress step by step. Similarly, we do not have a single objective function, rather each competitor has its own objective (and they may be different). Because *strategies* rather than *heuristics* are the tool used to decide routes for competitive problems, we define a new space, *strategy space* which is the set of all strategies. The object of future study could be to study strategy space and the mapping between strategy space and competitive problem space. Perhaps some heuristic which guides the search in strategy space could be useful (following a hyper-heuristic or genetic programming idea).

D.5.2 Information

As mentioned in an earlier section, *information* has a vital role to play in competitive problems.

D.5.2.1 Types of information

Information may be broadly grouped into two types: information about the problem and information about the opponent. We list some of the things that are included in each type:

Information about the problem:

- Knowledge of the initial network state: arc rewards, costs, penalties.
- Knowledge of subsequent network states. Is the vehicle aware of changes in the network state (acts of god, etc), or must he extrapolate? Does the vehicle know the reward function, from which he can calculate rewards, or just the starting value?
- Knowledge of itself. This is usually assumed, but is not necessarily so. Factors include budget and capacity.

Information about the opponent:

- Knowledge of opponent's location. This is a fundamental item of information.
- Knowledge of opponent's strategy. From this much can be extrapolated.
- Knowledge of opponent. Does the opponent have the same budget and capacity? Does the opponent have the same reward, cost structures?
- Knowledge of opponent's knowledge. This may seem to be getting a bit involved, but is a valid point. "Does he know that I know where he is?", "Does he know where I am?"… etc.

D.5.2.2 Access to information

We need not assume that knowledge is constant. As a vehicle traverses the network it gains knowledge about the arcs it traverses. We list some other ways that information access could be modelled:

- Visual knowledge. Perhaps a vehicle could be given some 'distance' that it can 'see' and each arc have some 'distance' associated with it. Then a vehicle could gain knowledge about the arcs in its 'radius', and also the location of the opponent (if he were within range).
- Knowledge trade. Perhaps certain pieces of information could be traded with the opponent. This presupposes the existence of a framework for communication between the competitors; perhaps a competitor could 'offer' a trade of certain pieces of information and the opponent would then 'accept' or 'decline'. If accepted then both competitors 'pool of knowledge' would be updated. The opponent might decline if he already has the information that is offered, or might accept even so if he didn't want his opponent to know he knows… all this would depend on sufficiently complicated strategies.
- Knowledge purchase. Knowledge could be 'purchased' from some third party, at the price of either reward or the use of some of the cost budget. Such knowledge could either be added to the 'pool' instantly, or have some delay (while the third party goes away and finds it). Perhaps there are only certain points on the network where knowledge can be purchased…
- Other knowledge acquisition. Perhaps if the vehicle returns to his depot (and maybe pays a fee) he can be 'updated'.

D.5.3 Variations

We now briefly list some other potential about what could be included in a competitive problem.

- Strategies need not be constant for the whole 'game'. Perhaps there could be a 'meta-strategy' which guides the choice of strategy used based on knowledge of the problem and opponent. It is possible that certain strategies work well against certain opposing strategies, in which case finding out what strategy the opponent is using could be quite important.
- 'Dumb' third parties. There could be other vehicles at work in the network. Perhaps they 'repair' arcs (increasing reward/decreasing cost). Perhaps they also compete for reward (though using a transparent strategy). Perhaps they must be found and chased in order to 'buy' information from.
- Learning. It would be interesting to supply the 'vehicle controller' with the capacity to 'learn' and modify strategies from game to game. An obvious example of this would be the decision of which strategy to pit against a known opponent strategy… this could be learned through trial and error over several games. It would then be interesting to pit a 'learned' version against an early version of the same controller. We would expect the smart version to win most times.

Bibliography

1. Eternity II Puzzle Wikipedia Page (*http://en.wikipedia.org/wiki/Eternity_II_puzzle*).

2. Extensible Markup Language (XML) (*http://www.w3.org/XML/*).

3. B. S. Alprin (1975) A simulation approach to solve the snow and ice removal problem in an urban area. *MSc dissertation*, The University of Tulsa.

4. R. Alvarez-Valdés, E. Benavent, V. Campos, A. M. E. Corberán, J. M. Tamarit, and V. Valls (1993) ARC, a computerized system for urban garbage collection, *TOP* **1**(1), 89-105.

5. A. Amberg, W. Domschke, and S. Voß (2000) Multiple center capacitated arc routing problems: a tabu search algorithm using capacitated trees, *European Journal of Operational Research* **124**(2), 360-376.

6. Andreatta, A. A., Carvalho, S. E. R., and Ribeiro, C. C. (2002) A framework for the development of local search heuristics for combinatorial optimization problems, 59-79. In: S. Voß and D. L. Woodruff (eds) *Optimization software class libraries.* Kluwer.

7. C. Archetti, D. Feillet, A. Hertz, and M. G. Speranza (2010) The undirected capacitated arc routing problem with profits, *Computers & Operations Research* **37**(11), 1860-1869.

8. E. Arkin, J. Mitchell, and G. Narasimhan (1998) Resource-constrained geometric network optimization, *Proceedings of ACM Symposium on Computational Geometry*, 307-316.

9. J. E. C. Arroyo and V. A. Armentano (2005) Genetic local search for multi-objective flowshop scheduling problems, *European Journal of Operational Research* **167**(3), 717-738.

10. A. A. Assad, W. L. Pearn, and B. L. Golden (1987) The capacitated Chinese postman problem: lower bounds and solvable cases, *American Journal of Mathematics and Management Science* **7**, 63-88.

11. B. Awerbuch, Y. Azar, A. Blum, and S. Vempala (1998) New approximation guarantees for minimum-weight *k*-trees and prize-collecting salesmen, *SIAM Journal on Computing* **28**(1), 254-262.

12. N. Azizi and S. Zolfaghari (2004) Adaptive temperature control for simulated annealing: a comparative study, *Computers & Operations Research* **31**(14), 2439-2451.

13. V. Bachelet and E.-G. Talbi (2000) A parallel co-evolutionary metaheuristic, *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, 628-635.

14. R. Bai, E. K. Burke, and G. Kendall (2008) Heuristic, meta-heuristic and hyper-heuristic approaches for fresh produce inventory control and shelf space allocation, *Journal of the Operational Research Society* **59**, 1387–1397.

15. E. Balas (1995) The prize collecting travelling salesman problem: II. polyhedral results, *Networks* **25**, 199-216.

16. E. Balas (1989) The prize collecting travelling salesman problem, *Networks* **19**, 621-636.

17. R. Battiti and F. Mascia (2010) Reactive and dynamic local search for max-clique: engineering effective building blocks, *Computers & Operations Research* **37**(3), 534-542.

18. J. Baxter (1981) Local optima avoidance in depot location, *Journal of the Operational Research Society* **32**(9), 815-819.

19. I. R. Beale (2002) Subset selection routing: modelling and heuristics. *PhD thesis*, Massey University.

20. J. E. Beasley and E. M. Nascimento (1996) The vehicle routing-allocation problem: a unifying framework, *TOP* **4**(1), 65-86.

21. E. L. Beltrami and L. D. Bodin (1974) Networks and vehicle routing for municipal waste collection, *Networks* **4**, 65-94.

22. E. Benavent, V. Campos, A. Corberán, and E. Mota (1990) The capacitated arc routing problem. A heuristic algorithm, *QÜESTIIÓ* **14**, 107-122.

23. E. Benavent, A. Corberán, E. Pińana, I. Plana, and J. M. Sanchis (2005) New heuristic algorithms for the windy rural postman problem, *Computers & Operations Research* **32**(12), 3111-3128.

24. E. Benavent and D. Soler (1999) The directed rural postman problem with turn penalties, *Transportation Science* **33**(4), 408-418.

25. P. Beullens, L. Muyldermans, D. Cattrysse, and D. Van Oudheusden (2003) A guided local search heuristic for the capacitated arc routing problem, *European Journal of Operational Research* **147**(3), 629-643.

26. D. Bienstock , M. X. Goemans, D. Simchi-Levi, and D. Williamson (1993) A note on the prize collecting salesman problem, *Mathematical Programming* **59**, 413-420.

27. Binato, S., Hery, W. J., Loewenstern, D., and Resende, M. G. C. (2002) A GRASP for job shop scheduling, 59-79. In: C. C. Ribeiro and P. Hansen (eds) *Essays and surveys in metaheuristics.* Kluwer Academic Publishers.

28. L. D. Bodin, G. Fagin, R. Welebny, and J. Greenberg (1989) The design of a computerized sanitation vehicle routing and scheduling system for the town of Oyster Bay, New York, *Computers & Operations Research* **16**, 45-54.

29. L. D. Bodin and S. J. Kursh (1978) A computer-assisted system for the routing and scheduling of street sweepers, *Operations Research* **26**(4), 525-537.

30. L. D. Bodin and S. J. Kursh (1979) A detailed description of a computer system for the routing and scheduling of street sweepers, *Computers & Operations Research* **6**, 181-198.

31. G. E. P. Box and N. R. Draper (1987) *Empirical model-building and response surfaces*. Wiley.

32. J. L. Bresina (1996) Heuristic-biased stochastic sampling, *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, 271-278.

33. R. J. Brideau and T. M. Cavalier (1994) The maximum collection problem with time-dependent rewards, *Presented at TIMS International Conference, Alaska*.

34. P. Brucker (1980) The Chinese postman problem for mixed graphs, *Proceedings of the International Workshop on Graph Theoretic Concepts in Computer Science*, 354-366.

35. E. K. Burke, G. Kendall, and E. Soubeiga (2003) A tabu-search hyperheuristic for timetabling and rostering, *Journal of Heuristics* **9**(6), 451-470.

36. Burke, E., Kendall, G., Newall, J., Hart, E., Ross, P., and Schulenburg, S. (2003) Hyper-heuristics: an emerging direction in modern search technology, 457-474. In: F. Glover and G. A. Kochenberger (eds) *Handbook of metaheuristics.* Springer.

37. S. Butt and D. Ryan (1999) An optimal solution procedure for the multiple tour maximum collection problem using column generation, *Computers & Operations Research* **26**(4), 427-441.

38. S. E. Butt and T. M. Cavalier (1994) A heuristic for the multiple tour maximum collection problem, *Computers & Operations Research* **21**(1), 101-111.

39. S. E. Butt and D. Ryan (1996) Using column generation to solve the multiple tour maximum collection problem, *Proceedings of the 32nd Annual ORSNZ Conference*, 143-148.

40. A. Bölte and U. W. Thonemann (1996) Optimizing simulated annealing schedules with genetic programming, *European Journal of Operational Research* **92**(2), 402-416.

41. E. A. Cabral , M. Gendreau, G. Ghiani, and G. Laporte (2004) Solving the hierarchical Chinese postman problem as a rural postman problem, *European Journal of Operational Research* **155**(1), 44-50.

42. V. Cerny (1985) Thermodynamical approach to the traveling salesman problem: an efficient simulation algorithm, *Journal of Optimization Theory and Applications* **45**(1), 41-51.

43. I. M. Chao, B. L. Golden, and E. A. Wasil (1996) A fast and effective heuristic for the orienteering problem, *European Journal of Operational Research* **88**, 475-489.

44. I. M. Chao, B. L. Golden, and E. A. Wasil (1996) The team orienteering problem, *European Journal of Operational Research* **88**, 464-474.

45. L. Chapleau, J. A. Ferland, G. Lapalme, and J. M. Rousseau (1984) A parallel insert method for the capacitated arc routing problem, *Operations Research Letters* **3**(2), 95-99.

46. I. Charon and O. Hudry (2001) The noising methods: a generalization of some metaheuristics, *European Journal of Operational Research* **135**(1), 86-101.

47. N. Christofides (1973) The optimal traversal of a graph, *Omega* **1**, 719-732.

48. Christofides, N., Benavent, E., Campos, V., Corberan, A., and Mota, E. (1984) An optimal method for the mixed postman problem. In: P. Thoft-Christensen (eds) *System Modelling and Optimization.* Springer.

49. N. Christofides, V. Campos, A. Corberán, and E. Mota (1981) An algorithm for the rural postman problem, *Imperial College Report IC.O.R.81.5, London*.

50. N. Christofides, V. Campos, A. Corberán, and E. Mota (1986) An algorithm for the rural postman problem on a directed graph, *Math. Programming Study* **26**, 155-166.

51. R. M. Clark and J. I. Gillean (1975) Analyses of solid waste management operations in Cleveland, Ohio, *Interfaces* **6**(1, part 2), 32-42.

52. R. M. Clark and C. H. Lee Jr. (1976) Systems planning for solid waste collection, *Computers & Operations Research* **3**, 157-173.

53. G. Clarke and J. W. Wright (1964) Scheduling of vehicles from a central depot to a number of delivery points, *Operations Research* **12**, 568-581.

54. J. Clossey, G. Laporte, and P. Soriano (2001) Solving arc routing problems with turn penalties, *Journal of the Operational Research Society* **52**, 433–439.

55. R. K. Congram (2000) Polynomially searchable exponential neighbourhoods for sequencing problems in combinatorial optimisation. *PhD thesis*, University of Southampton.

56. D. T. Connolly (1990) An improved annealing scheme for the QAP, *European Journal of Operational Research* **46**, 93-100.

57. S. Consoli, K. Darby-Dowman, N. Mladenovic, and J. A. M. Pérez (2009) Greedy randomized adaptive search and variable neighbourhood search for the minimum labelling spanning tree problem, *European Journal of Operational Research* **196**(2), 440-449.

58. T. M. Cook and B. S. Alprin (1976) Snow and ice removal in an urban environment, *Management Science* **23**(3), 227-234.

59. J. H. Coombs, A. H. Reenear, and S. J. DeRose (1987) Markup systems and the future of scholarly text processing, *Communications of the ACM* **30**(11), 933-947.

60. A. Corberán, A. Letchford, and J. M. Sanchis (2001) A cutting plane algorithm for the general routing problem, *Mathematical Programming* **90**, 291-316.

61. A. Corberán, R. Martí, and J. M. Sanchis (2002) A GRASP heuristic for the mixed Chinese postman problem, *European Journal of Operational Research* **142**, 70-80.

62. A. Corberán, A. Romero, and J. M. Sanchis (1999) The general routing problem on a mixed graph, *Technical Paper, Department of Statistics and OR, University of Valencia, Spain*.

63. A. Corberán and J. M. Sanchis (1994) A polyhedral approach to the rural postman problem, *European Journal of Operational Research* **79**, 95-114.

64. A. Corberán and J. M. Sanchis (1998) The general routing problem polyhedron: facets from the RPP and GTSP polyhedra, *European Journal of Operational Research* **108**(3), 538-550.

65. J. Current, H. Pirkul, and E. Rolland (1994) Efficient algorithms for solving the shortest covering path problem, *Transportation Science* **28**(4), 317-327.

66. J. R. Current and D. A. Schilling (1994) The median tour and maximal covering tour problems: formulations and heuristics, *European Journal of Operational Research* **73**, 114-126.

67. J. R. Current and D. A. Schilling (1989) The covering salesman problem, *Transportation Science* **23**(3), 208-213.

68. M. De Souza and P. Martins (2008) Skewed VNS enclosing second order algorithm for the degree constrained minimum spanning tree problem, *European Journal of Operational Research* **191**(3), 677-690.

69. M. Dell'Amico, F. Maffioli, and P. Värbrand (1995) On prize-collecting tours and the asymmetric travelling salesman problem, *International Transactions in Operational Research* **2**(3), 297-308.

70. L. Di Gaspero and A. Schaerf (2003) EasyLocal++: an object-oriented framework for flexible design of local search algorithms, *Software Practice and Experience* **33**(8), 733-765.

71. M. Diaby and R. Ramesh (1995) The distribution problem with carrier service: a dual based penalty approach, *ORSA Journal on Computing* **7**, 24-35.

72. X. Dong, H. Huang, and P. Chen (2009) An iterated local search algorithm for the permutation flowshop problem with total flowtime criterion, *Computers & Operations Research* **36**(5), 1664-1669.

73. M. Dorigo and C. Blum (2005) Ant colony optimization theory: a survey, *Theoretical Computer Science* **344**, 243-278.

74. M. Dror and A. Langevin (1997) A generalized travelling salesman problem approach to the directed clustered rural postman problem, *Transportation Science* **31**(2), 187-192.

75. G. Dueck (1990) New optimization heuristics: the great deluge algorithm and the record-to-record travel, *IBM Tech Report #89.06.011*.

76. G. Dueck (1993) New optimization heuristics: the great deluge algorithm and the record-to-record travel, *Journal of Computational Physics* **104**, 86-92.

77. G. Dueck and T. Scheuer (1990) Threshold accepting: a general purpose optimization algorithm appearing superior to simulated annealing, *Journal of Computational Physics* **90**, 161-175.

78. J. Edmonds and E. L. Johnson (1973) Matching, Euler tours and the Chinese postman problem, *Mathematical Programming* **5**, 88-124.

79. J. Egeblad, B. K. Nielsen, and A. Odgaard (2007) Fast neighborhood search for two- and three-dimensional nesting problems, *European Journal of Operational Research* **183**(3), 1249-1266.

80. R. W. Eglese (1990) Simulated annealing: a tool for operational research, *European Journal of Operational Research* **46**, 271-281.

81. Eglese, R. W. and Letchford, A. N. (2000) Polyhedral theory for arc routing problems, 199-230. In: M. Dror (eds) *ARC Routing: Theory, Solutions and Applications.* Kluwer Academic Publishers.

82. R. W. Eglese and L. Y. O. Li (1992) Efficient routing for winter gritting, *Journal of the Operational Research Society* **43**(11), 1031-1034.

83. R. W. Eglese and H. Murdock (1991) Routing road sweepers in a rural area, *Journal of the Operational Research Society* **42**(4), 281-288.

84. H. A. Eiselt , M. Gendreau, and G. Laporte (1995) Arc routing problems, part I: the Chinese postman problem, *Operations Research* **43**(2), 231-242.

85. H. A. Eiselt , M. Gendreau, and G. Laporte (1995) Arc routing problems, part II: the rural postman problem, *Operations Research* **43**(3), 399-414.

86. E. Erkut and J. Zhang (1996) The maximum collection problem with time-dependent rewards, *Naval Research Logistics* **43**, 749-763.

87. J. R. Evans and E. Minieka (1992) *Optimization algorithms for networks and graphs*. Marcel Dekker.

88. O. Faroe, D. Pisinger, and M. Zachariasen (2003) Guided local search for final placement in VLSI design, *Journal of Heuristics* **9**(3), 269-295.

89. O. Faroe, D. Pisinger, and M. Zachariasen (2003) Guided local search for the three-dimensional bin-packing problem, *INFORMS Journal on Computing* **15** (3), 267-283.

90. D. Feillet, P. Dejax, and M. Gendreau (2005) The profitable arc tour problem: solution with a branch-and-price algorithm, *Transportation Science* **39**(4), 539-552.

91. D. Feillet, P. Dejax, and M. Gendreau (2005) Traveling salesman problems with profits, *Transportation Science* **39**(2), 188-205.

92. T. A. Feo and M. G. C. Resende (1989) A probabilistic heuristic for a computationally difficult set covering problem, *Operations Research Letters* **8**, 67-71.

93. T. A. Feo and M. G. C. Resende (1995) Greedy randomized adaptive search procedures, *Journal of Global Optimization* **2**, 1-27.

94. E. Fernandez , O. Meza, and R. e. Al. Garfinkel (2003) On the undirected rural postman problem: tight bounds based on a new formulation, *Operations Research* **51**(2), 281-291.

95. Festa, P. and Resende, M. G. C. (2002) GRASP: an annotated bibliography, 325-367. In: C. C. Ribeiro and P. Hansen (eds) *Essays and surveys on metaheuristics.* Kluwer Academic Publishers.

96. Fink, A. and Voß, S. (2002) HotFrame: a heuristic optimization framework, 81-154. In: S. Voß and D. L. Woodruff (eds) *Optimization software class libraries.* Kluwer.

97. Fink, A., Voß, S., and Woodruff, D. L. (2003) Metaheuristic class libraries, 515-535. In: F. Glover and G. A. Kochenberger (eds) *Handbook of metaheuristics.* Springer.

98. M. Fischetti , J. Gonzŕlez, and P. Toth (1998) Solving the orienteering problem through branch-and-cut, *INFORMS Journal on Computing* **10**(2), 133-148.

99. M. Fischetti , H. Hamacher, K. Jörnsten, and F. Maffioli (1994) Weighted *k*-cardinality trees: complexity and polyhedral structure, *Networks* **24**, 11-21.

100. K. Fleszar and K. S. Hindi (2004) Solving the resource-constrained project scheduling problem by a variable neighbourhood search, *European Journal of Operational Research* **155**(2), 402-413.

101. G. Fleury, P. Lacomme, C. Prins, and W. Ramdane-Chérif (2005) Improving robustness of solutions to arc routing problems, *Journal of the Operational Research Society* **56**, 526–538.

102. L. R. Ford and D. R. Fulkerson (1962) *Flows in networks*. Princeton University Press.

103. G. N. Frederickson (1979) Approximation algorithms for some postman problems, *Journal of the Association of Computing Machinery* **26**, 538-554.

104. P. W. Frizzell and J. W. Giffin (1995) The split delivery vehicle scheduling problem with time windows and grid network distances, *Computers & Operations Research* **22**(6), 655-667.

105. C. G. Garcia, D. Pérez-Brito, V. Campos, and R. Marti (2006) Variable neighborhood search for the linear ordering problem, *Computers & Operations Research* **33**(12), 3549-3565.

106. R. S. Garfinkel and I. R. Webb (1999) On crossings, the crossing postman problem, and the rural postman problem, *Networks* **34**(3), 173-180.

107. L. F. Gelders and D. G. Cattrysse (1991) Public waste collection: a case study, *Belgian Journal of Operations Research, Statistics, and Computer Science* **31**, 3-15.

108. M. Gendreau , G. Laporte, and F. Semet (1997) The covering tour problem, *Operations Research* **45**(4), 568-576.

109. M. Gendreau , G. Laporte, and F. Semet (1998) A branch-and-cut algorithm for the undirected selective travelling salesman problem, *Networks* **32**, 263-273.

110. M. Gendreau , A. Hertz, and G. Laporte (1994) A tabu search heuristic for the vehicle routing problem, *Management Science* **40**(10), 1276-1289.

111. M. Gendreau , G. Laporte, and J.-Y. Potvin (1999) Metaheuristics for the vehicle routing problem, *Les Cahiers du GERAD G-98-52*.

112. D. H. Gensch (1978) An industrial application of the travelling salesman's subtour problem, *AIIE Transactions* **10**, 362-370.

113. G. Ghiani and G. Laporte (2000) A branch-and-cut algorithm for the undirected rural postman problem, *Mathematical Programming* **87**(3), 467-481.

114. G. Ghiani and G. Improta (2000) An efficient transformation of the generalized vehicle routing problem, *European Journal of Operational Research* **122**(1), 11-17.

115. G. Ghiani, R. Musmanno, G. Paletta, and C. Triki (2005) A heuristic for the periodic rural postman problem, *Computers & Operations Research* **32**(2), 219-228.

116. D. Ghosh and G. Sierksma (2002) Complete local search with memory, *Journal of Heuristics* **8**(6), 571-584.

117. F. Glover (1986) Future paths for integer programming and links to artificial intelligence, *Computers & Operations Research* **13**(5), 533-549.

118. F. Glover (1989) Tabu search - part I, *ORSA Journal on Computing* **1**, 190-206.

119. F. Glover (1990) Tabu search: a tutorial, *Interfaces* **20**(4), 74-94.

120. F. Glover and M. Laguna (1997) *Tabu search*. Kluwer Academic Publishers.

121. M. X. Goemans and D. Williamson (1995) A general approximation technique for constrained forest problems, *SIAM Journal on Computing* **24**, 296-317.

122. Golden, B. L. and Assad, A. A. (1988) Vehicle Routing: Methods and Studies, 319-343. In: M. Fischetti and P. Toth (eds) *An additive appoach for the optimal solution of the prize collecting travelling salesman problem.* Elsevier Science.

123. B. L. Golden, J. S. DeArmon, and E. K. Baker (1983) Computational experiments with algorithms for a class of routing problems, *Computers & Operations Research* **10**(1), 47-59.

124. B. L. Golden, L. Levy, and R. Vohra (1987) The orienteering problem, *Naval Research Logistics* **34**, 307-318.

125. B. L. Golden, Q. Wang, and L. Liu (1988) A multifaceted heuristic for the orienteering problem, *Naval Research Logistics* **35**, 359-366.

126. B. Golden, L. Levy, and R. Dahl (1981) Two generalizations of the travelling Salesman problem, *Omega, The International Journal of Management Science* **9**(4), 439-441.

127. B. L. Golden and R. T. Wong (1981) Capacitated arc routing problems, *Networks* **11**, 305-315.

128. P. Greistorfer (2003) A tabu scatter search metaheuristic for the arc routing problem, *Computers & Industrial Engineering* **44**(2), 249-266.

129. A. Grosso, A. R. M. J. U. Jamali, and M. Locatelli (2009) Finding maximin latin hypercube designs by iterated local search heuristics, *European Journal of Operational Research* **197**(2), 541-547.

130. M. Grötschel and Z. Win (1992) A cutting plane algorithm for the windy postman problem, *Mathematical Programming* **55**, 339-358.

131. M. Guan (1962) Graphic programming using odd or even points, *Chinese Mathematics* **1**, 237-277.

132. M. Guan (1984) On the windy postman problem, *Discrete Applied Mathematics* **9**, 41-46.

133. A. Hamacher and C. Moll (1995) The Euclidean travelling salesman selection problem, *Report no 95-199, Zentrum für Paralleles Rechnen, Universität zu Köln*.

134. P. Hansen and N. Mladenovic (2001) Variable neighborhood search: principles and applications, *European Journal of Operational Research* **130**(3), 449-467.

135. Hansen, P. and Mladenovic, N. (2003) Variable neighbourhood search, 145-184. In: F. Glover and G. A. Kochenberger (eds) *Handbook of metaheuristics.* Springer.

136. P. Hansen, C. Oguz, and N. Mladenovic (2008) Variable neighborhood search for minimum cost berth allocation, *European Journal of Operational Research* **191**(3), 636-649.

137. E. Haslam and J. R. Wright (1991) Application of routing technologies to rural snow and ice control, *Transportation Research Record* **1304**, 202-211.

138. M. Hayes and J. M. Norman (1984) Dynamic programming in orienteering: route choice and the siting of controls, *Journal of the Operational Research Society* **35**(9), 791-796.

139. A. Hertz, G. Laporte, and P. N. Hugo (1999) Improvement procedures for the undirected rural postman problem, *INFORMS Journal on Computing* **11**(1), 53-62.

140. A. Hertz, G. Laporte, and M. Mittaz (2000) A tabu search heuristic for the capacitated arc routing problem, *Operations Research* **48**(1), 129-135.

141. A. Hertz and M. Mittaz (2001) A variable neighborhood descent algorithm for the undirected capacitated arc routing problem, *Transportation Science* **35**(4), 425-434.

142. A. Hertz and M. Mittaz (2001) A variable neighbourhood descent algorithm for the undirected capacitated arc routing problem, *Transportation Science* **35**(4), 425-434.

143. M. Hifi and M. Michrafy (2006) A reactive local search-based algorithm for the disjunctively constrained knapsack problem, *Journal of the Operational Research Society* **57**, 718–726.

144. J. N. Hooker (1994) Needed: an empirical science of algorithms, *Operations Research* **42**(2), 201-212.

145. J. N. Hooker (1995) Testing heuristics: we have it all wrong, *Journal of Heuristics* **1**, 33-42.

146. H. H. Hoos and T. Stützle (2005) *Stochastic local search: foundations and applications*. Morgan Kaufmann Publishers.

147. T. C. Hu, A. B. Kahng, and C.-W. A. Tsao (1995) Old bachelor acceptance: a new class of non-monotone threshold accepting methods, *ORSA Journal on Computing* **7**(4), 417-425.

148. C. L. Huntley and D. E. Brown (1996) Parallel genetic algorithms with local search, *Computers & Operations Research* **23**(6), 559-571.

149. H. Ishibuchi, S. Misaki, and H. Tanaka (1995) Modified simulated annealing algorithms for the flow shop sequencing problem, *European Journal of Operational Research* **81**(2), 388-398.

150. R. James (2000) A framework for search heuristics, *Presented at the 35th Annual ORSNZ Conference* .

151. M. R. Johnston and S. Chukova (2007) Rural postmen, rewards and grid networks, *The 42nd Annual Conference of the Operational Research Society of New Zealand*.

152. M. R. Johnston (1999) Dynamic routing with competition: foundations and strategies. *PhD thesis*, Massey University.

153. N. Jozefowiez, F. Semet, and E.-G. Talbi (2009) An evolutionary algorithm for the vehicle routing problem with route balancing, *European Journal of Operational Research* **195**(3), 761-769.

154. M. G. Kantor and M. B. Rosenwein (1992) The orienteering problem with time windows, *Journal of the Operational Research Society* **43**(6), 629-635.

155. G. V. Kass (1980) An exploratory technique for investigating large quantities of categorical data, *Applied Statistics* **29**, 119-127.

156. S. Kataoka and S. Morito (1988) An algorithm for single constraint maximum collection problem, *Journal of the Operations Research Society of Japan* **31**(4), 515-530.

157. Katayama, Kengo, and H. Narihisa (2001) Performance of simulated annealing-based heuristic for the unconstrained binary quadratic programming problem, *European Journal of Operational Research* **134**(1), 103-119.

158. C. P. Keller (1989) Algorithms to solve the orienteering problem: a comparison, *European Journal of Operational Research* **41**, 224-231.

159. C. P. Keller and M. F. Goodchild (1988) The multiobjective vending problem: a generalization of the travelling salesman problem, *Environment and Planning B: Planning and Design* **15**, 447-460.

160. S. Kirkpatrick, C. D. Jr. Gelatt, and M. P. Vecchi (1983) Optimization by simulated annealing, *Science* **220**(4598), 671-680.

161. P. Kouvelis and W.-C. Chiang (1992) A simulated annealing procedure for single row layout problems in flexible manufacturing systems, *International Journal of Production Research* **30**(4), 717-732.

162. G. Laporte and S. Martello (1990) The selective travelling salesman problem, *Discrete Applied Mathematics* **26**, 193-207.

163. G. Laporte (1997) Modeling and solving several classes of arc routing problems as traveling salesman problems, *Computers & Operations Research* **24**(11), 1057-1061.

164. G. Laporte, M. Gendreau, J.-Y. Potvin, and F. Semet (2000) Classical and modern heuristics for the vehicle routing problem, *International Transactions in Operations Research* **7**, 285-3000.

165. A. Le Bouthillier and T. G. Crainic (2005) A cooperative parallel meta-heuristic for the vehicle routing problem with time windows, *Computers & Operations Research* **32**(7), 1685-1708.

166. D. S. Lee, V. S. Vassiliadis, and J. M. J. M. Park (2004) A novel threshold accepting meta-heuristic for the job-shop scheduling problem, *Computers & Operations Research* **31**(13), 2199-2213.

167. A. C. Leifer and M. B. Rosenwein (1994) Strong linear programming relaxations for the orienteering problem, *European Journal of Operational Research* **73**, 517-523.

168. P. F. Lemieux and L. Campagna (1984) The snow ploughing problem solved by a graph theory algorithm, *Civil Engineering Systems* **1**, 337-341.

169. J. K. Lenstra and A. H. J. Rinnooy Kan (1976) On general routing problems, *Networks* **6**, 273-280.

170. A. N. Letchford and R. W. Eglese (1998) The rural postman problem with deadline classes, *European Journal of Operational Research* **105**(3), 390-400.

171. A. N. Letchford (1997) New inequalities for the general routing problem, *European Journal of Operational Research* **96**(2), 317-322.

172. L. Levy (1987) The walking line of travel problem: an application of arc routing and partitioning. *PhD dissertation*, University of Maryland.

173. Levy, L. and Bodin, L. D. (1988) Scheduling the postal carriers for the United States postal service: an application of arc partitioning and routing, 359-394. In: B. L. Golden and A. A. Assad (eds) *Vehicle routing: methods and studies.* North-Holland.

174. L. Y. O. Li and Z. Fu (2002) The school bus routing problem: a case study, *Journal of the Operational Research Society* **53**, 552–558.

175. C. K. Y. Lin, K. B. Haley, and C. Sparks (1995) A comparative study of both standard and adaptive versions of threshold accepting and simulated annealing algorithms in three scheduling problems, *European Journal of Operational Research* **83**(2), 330-346.

176. Lourenço, H. R., Martin, O. C., and Stützle, T. (2003) Iterated local search, 321-353. In: F. Glover and G. A. Kochenberger (eds) *Handbook of metaheuristics.* Springer.

177. M. Lundy and A. Mees (1986) Convergence of an annealing algorithm, *Mathematical Programming* **34**(1), 111-124.

178. E. M. Macambira and C. C. de Souza (2000) The edge-weighted clique problem: valid inequalities, facets and polyhedral computations, *European Journal of Operational Research* **123**(2), 346-371.

179. E. M. Macambira (2002) An application of tabu search heuristic for the maximum edge-weighted subgraph problem, *Annals of Operations Research* **117**, 175-190.

180. C. Malandraki and M. S. Daskin (1992) Time dependent vehicle routing problems: formulation, properties and heuristic algorithms, *Transportation Science* **26**, 185-200.

181. C. Malandraki and M. S. Daskin (1993) The maximum benefit Chinese postman problem and the maximum benefit travelling salesman problem, *European Journal of Operational Research* **65**, 218-234.

182. C. Malandraki and R. B. Dial (1996) A restricted dynamic programming heuristic algorithm for the time dependent travelling salesman problem, *European Journal of Operational Research* **90**, 45-55.

183. J. W. Male and J. C. Liebman (1978) Districting and routing for solid waste collection, *Journal of the Environmental Engineering Division* **104**(1), 1-14.

184. O. Martin, S. W. Otto, and E. W. Felten (1991) Large-step Markov chains for the traveling salesman problem, *Complex Systems* **5**(3), 299-326.

185. O. Martin, S. W. Otto, and E. W. Felten (1992) Large-step Markov chains for the TSP incorporating local search heuristics, *Operations Research Letters* **11**, 219-224.

186. Martí, R. (2003) Multi-start methods, 355-368. In: F. Glover and G. A. Kochenberger (eds) *Handbook of metaheuristics.* Springer.

187. R. Matthews (2001) The ideas machine, *New Scientist*.

188. N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller (1953) Equation of state calculations by fast computing machines, *Journal of Chemical Physics* **21**(6), 1087-1092.

189. L. Michel and P. Van Hentenryck (1999) LOCALIZER: a modeling language for local search, *INFORMS Journal on Computing* **11**(1), 1-14.

190. L. Michel and P. Van Hentenryck (2001) Localizer++: an open library for local search, *Technical Report, CS-01-03, Brown University*.

191. H. H. Millar (1996) Planning fish scouting activity in industrial fishing, *Fisheries Research* **25**, 63-75.

192. H. H. Millar and M. Kiragu (1997) A time-based formulation and upper bounding scheme for the selective travelling salesperson problem, *Journal of the Operational Research Society* **48**, 511-518.

193. J. Mittenthal and C. E. Noon (1992) An insert/delete heuristic for the travelling salesman subset-tour problem with one additional constraint, *Journal of the Operational Research Society* **43**(3), 277-283.

194. N. Mladenovic and P. Hansen (1997) Variable neighbourhood search, *Computers & Operations Research* **24**, 1097-1100.

195. S. Mohan, M. Gendreau, and J.-M. Rousseau (2010) Heuristics for the stochastic Eulerian tour problem, *European Journal of Operational Research* **203**(1), 107-117.

196. P. Moscato and J. F. Fontanari (1990) Convergence and finite-time behaviour of simulated annealing, *Advances in Applied Probability* **18**, 747-771.

197. M. C. Mourão and M. T. Almeida (2000) Lower-bounding and heuristic methods for a refuse collection vehicle routing problem, *European Journal of Operational Research* **121**(2), 420-434.

198. Y. Nobert and J. C. Picard (1996) An optimal algorithm for the mixed Chinese postman problem, *Networks* **27**, 95-108.

199. J. W. Ohlmann, J. C. Bean, and S. G. Henderson (2004) Convergence in probability of compressed annealing, *Mathematics of Operations Research* **29**(4), 837-860.

200. J. W. Ohlmann and B. W. Thomas (2007) A compressed-annealing heuristic for the traveling salesman problem with time windows, *INFORMS Journal on Computing* **19**(1), 80-90.

201. A. Olivera and O. Viera (2007) Adaptive memory programming for the vehicle routing problem with multiple trips, *Computers & Operations Research* **34**(1), 28-47.

202. C. S. Orloff (1974) A fundamental problem in vehicle routing, *Networks* **4**, 35-64.

203. C. H. Papadimitriou (1976) On the complexity of edge traversing, *Journal of the Association of Computing Machinery* **23**, 544-554.

204. W. L. Pearn (1991) Augment-insert algorithms for the capacitated arc routing problem, *Computers & Operations Research* **18**, 189-198.

205. W. L. Pearn and J. B. Chou (1999) Improved solutions for the Chinese postman problem on mixed networks, *Computers & Operations Research* **26**(8), 819-827.

206. W. L. Pearn and T. C. Wu (1995) Algorithms for the rural postman problem, *Computers & Operations Research* **22**(8), 819-828.

207. W. L. Pearn (1989) Approximate solutions for the capacitated arc routing problem, *Computers & Operations Research* **16**(6), 589-600.

208. W. L. Pearn and C. M. Liu (1995) Algorithms for the Chinese postman problem on mixed networks, *Computers & Operations Research* **22**(5), 479-489.

209. K. Pearson (1901) On lines and planes of closest fit to systems of points in space, *Philosophical Magazine* **2**(6), 559-572.

210. M. Pirlot (1996) General local search methods, *European Journal of Operational Research* **92**, 493-511.

211. J.-Y. Potvin, P. Soriano, and M. Vallée (2004) Generating trading rules on the stock markets with genetic programming, *Computers & Operations Research* **31**(7), 1033-1047.

212. M. Prais and C. C. Ribeiro (2000) Parameter variation in GRASP procedures, *Investigación Operativa* **9**, 1-20.

213. M. Prais and C. C. Ribeiro (2000) Reactive GRASP: an application to a matrix decomposition problem in TDMA traffic assignment, *INFORMS Journal on Computing* **12**(3), 164(13).

214. J. Puchinger and G. R. Raidl (2008) Bringing order into the neighborhoods: relaxation guided variable neighborhood search, *Journal of Heuristics* **14**(5), 457-472.

215. W. Pullan (2008) Approximating the maximum vertex/edge weighted clique using local search, *Journal of Heuristics* **14**(2), 117-134.

216. B. Raghavachari and J. Veerasamy (1998) Approximation algorithms for the mixed postman problem, *Proceedings of 6th Integer Programming and Combinatorial Optimization*, 169-179.

217. R. Ramesh and K. M. Brown (1991) An efficient four-phase heuristic for the generalized orienteering problem, *Computers & Operations Research* **18**(2), 151-165.

218. R. Ramesh, Y. S. Yoon, and M. H. Karwan (1992) An optimal algorithm for the orienteering tour problem, *ORSA Journal on Computing* **4**(2), 155-165.

219. Resende, M. G. C. and Ribeiro, C. C. (2003) Greedy randomized adaptive search procedures, 219-249. In: F. Glover and G. A. Kochenberger (eds) *Handbook of metaheuristics.* Springer.

220. F. Ricca and B. Simeone (2008) Local search algorithms for political districting, *European Journal of Operational Research* **189**(3), 1409-1426.

221. A. M. Rodrigues and J. S. Ferreira (2001) Solving the rural postman problem by memetic algorithms, *MIC'2001 - 4th Metaheuristics International Conference*, 679-683.

222. S. Ropke and D. Pisinger (2006) An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows, *Transportation Science* **40**(4), 455-472.

223. B. D. Rosa, G. Improta, G. Ghiani, and R. Musmanno (2002) The arc routing and scheduling problem with transshipment, *Transportation Science* **36**(3), 301-313.

224. S. Roy and J.-M. Rousseau (1989) The capacitated Canadian postman problem, *INFOR* **27**(1), 58-73.

225. M. P. Scaparra and R. L. Church (2005) A GRASP and path relinking heuristic for rural road network development, *Journal of Heuristics* **11**(1), 89-108.

226. S. Schach (2006) *Object-oriented and classical software engineering*. McGraw-Hill.

227. P. R. Sokkappa (1990) The cost-constrained travelling salesman problem. *PhD thesis*, Lawrence Livermore National Laboratory, University of California.

228. F. J. Solis and R. J. B. Wets (1981) Minimization by random search techniques, *Mathematics of Operations Research* **6**(1), 19-30.

229. H. I. Stern and M. Dror (1979) Routing electric meter readers, *Computers & Operations Research* **6**, 209-223.

230. R. Stricker (1970) Public sector vehicle routing: the Chinese postman problem. *MSc. Dissertation*, Massachusetts Institute of Technology.

231. T. Stützle (1998) Local search algorithms for combinatorial problems - analysis, improvements, and new applications. *PhD thesis*, Darmstadt University of Technology.

232. T. Stützle (2006) Iterated local search for the quadratic assignment problem, *European Journal of Operational Research* **174**(3), 1519-1539.

233. L. Tang and X. Wang (2008) An iterated local search heuristic for the capacitated prize-collecting travelling salesman problem, *Journal of the Operational Research Society* **59**, 590-599.

234. C. D. Tarantilis, C. T. Kiranoudis, and V. S. Vassiliadis (2003) A list based threshold accepting metaheuristic for the heterogeneous fixed fleet vehicle routing problem, *Journal of the Operational Research Society* **54**, 65–71.

235. C. D. Tarantilis, C. T. Kiranoudis, and V. S. Vassiliadis (2004) A threshold accepting metaheuristic for the heterogeneous fixed fleet vehicle routing problem, *European Journal of Operational Research* **152**(1), 148-158.

236. T. Tsiligirides (1984) Heuristic methods applied to orienteering, *Journal of the Operational Research Society* **35**(9), 797-809.

237. S. Tsubakitani and J. R. Evans (1998) An empirical study of a new metaheuristic for the traveling salesman problem, *European Journal of Operational Research* **104**(1), 113-128.

238. W. B. Tucker and G. M. Clohan (1979) Computer simulation of urban snow removal, *Trans. Research Board Special Research Report No 185*.

239. W. Turner and E. Hougland (1975) The optimal routing of solid waste collection, *AIIE Transactions* **7**, 427-431.

240. D. Uroševic, J. Brimberg, and N. Mladenovic (2004) Variable neighborhood decomposition search for the edge weighted k-cardinality tree problem, *Computers & Operations Research* **31**(8), 1205-1213.

241. R. J. M. Vaessens, E. H. L. Aarts, and J. K. Lenstra (1998) A local search template, *Computers & Operations Research* **25**(11), 969-979.

242. J. M. Varanelli and J. P. Cohoon (1999) A fast method for generalized starting temperature determination in homogeneous two-stage simulated annealing systems, *Computers & Operations Research* **26**(5), 481-503.

243. J. L. G. V. Velarde and R. Marti (2008) Adaptive memory programing for the robust capacitated international sourcing problem, *Computers & Operations Research* **35**(3), 797-806.

244. T. Volgenant and R. Jonker (1987) On some generalizations of the travelling salesman problem, *Journal of the Operational Research Society* **38**(11), 1073-1079.

245. C. Voudouris and E. Tsang (1999) Guided local search and its application to the traveling salesman problem, *European Journal of Operational Research* **113**(2), 469-499.

246. Q. Wang, X. Sun, B. L. Golden, and J. Jia (1995) Using artificial neural networks to solve the orienteering problem, *Annals of Operations Research* **61**, 111-120.

247. T.-Y. Wang and K.-B. Wu (1999) A parameter set design procedure for the simulated annealing algorithm under the computational time constraint, *Computers & Operations Research* **26**(7), 665-678.

248. D. P. Williamson (1993) On the design of approximation algorithms for a class of graph problems. *PhD thesis*, Massachusetts Institute of Technology.

249. Z. Win (1987) Contributions to routing problems. *Doctoral dissertation*, Universität Augsburg.

250. Z. Win (1989) On the windy postman problem on Eulerian graphs, *Mathematical Programming* **44**, 97-112.

251. C. F. J. Wu and M. Hamada (2000) *Experiments: planning, analysis, and parameter design optimization*. Wiley.

252. J. Wunderlich, M. Collette, L. Levy, and L. D. Bodin (1992) Scheduling meter readers for southern California gas company, *Interfaces* **22**(3), 22-30.

253. R. Wyskida and J. Gupta (1973) IE's improve city's solid waste collection, *Industrial Engineering* **46**, 12-15.

254. E. Zachariadis, C. D. Tarantilis, and C. T. Kiranoudis (2010) An adaptive memory methodology for the vehicle routing problem with simultaneous pick-ups and deliveries, *European Journal of Operational Research* **202**(2), 401-411.