TOWARDS EXPLAINING BLACKBOX MODELS USING GENETIC NETWORK PROGRAMMING

A thesis presented in partial fulfilment of the requirements for the degree of
Master of Information Science
in
Computer Science
at Massey University, Albany,
New Zealand.

Haotian Zhu

2024

Contents

List of Figures

List of Algorithms

Abstract

With the emergence of deep learning systems capable of learning intricate robot manoeuvres, control, team coordination, and planning both from data and through interaction with the environment, numerous complex and non-linear problems have found solutions. However, these systems often function as 'black boxes,' lacking the ability to provide human-interpretable solutions. This study addresses the interpretability challenge in the field of Explainable AI by employing a 'black box' system as a target model and subsequently transforming it into a computational graph, equivalent to a Genetic Network Program (GNP). Furthermore, we present a methodology for refining and reducing the size of the GNP solution. Lastly, we also test if we could utilize the black box system as a guide to the fitness function in a GNP architecture. To illustrate its efficacy, we use a multi-goal path-finding problem from the OpenAI Gym framework. The experimental results demonstrate the efficacy of the converted and refined GNP solution in successfully addressing the taxi problem across 500 environments, constituting a comprehensive dataset. Notably, the refined GNP solution exhibits no redundant or unnecessary nodes. Despite the research's focused scope, centering on a single agent with multiple goals, the algorithms introduced in this study lay the groundwork for the development of more sophisticated and interpretable algorithms. These advancements are poised to tackle more intricate challenges in the future.

Acknowledgements

I would like to extend my deepest gratitude to my supervisor, Dr. Napoleon Reyes, for his invaluable guidance, patience, and support throughout this research journey. He put in a lot of effort during the research. He made slides, provided the research direction, and had meetings with me every week, each lasting at least one and a half hours. His insights and expertise have been fundamental to the completion of this thesis. I am profoundly grateful to my family for their unwavering love, understanding, and encouragement. Their support has been my strength. Moreover, I would also like to extend my thanks to my colleague, for their support and encouragement. Balancing full-time study with my responsibilities as a full-time developer was a challenging journey.

Chapter 1

Introduction

1.1 Overview of the Current State of Technology

1.1.1 Explainable AI

Explainable AI (XAI) refers to methods and techniques in artificial intelligence (AI) that make the AI systems understandable to humans. These methods aim to let human users understand and trust the results and output created by machine learning algorithms[1].

From the 1970s to the 1990s, symbolic reasoning systems like MYCIN, GUIDON, SOPHIE, and PROTOS could represent, reason about, and explain their reasoning for diagnostic, instructional, or machine-learning (explanation-based learning) purposes[2]. From the 1980s to the early 1990s, truth maintenance systems (TMS) enhanced the capabilities of causal-reasoning, rule-based, and logic-based inference systems. Many researchers started investigating whether it is possible to meaningfully extract the non-hand-coded rules generated by opaque trained neural networks[3] [4].

Nowadays, some approaches have been developed to make complex models such as machine learning and evolutionary algorithms more explainable and interpretable[5]. Other approaches explain some specific prediction made by a (nonlinear) black-box model, a goal referred to as "local interpretability" [6] [7]. This lack of transparency can be a serious problem in critical applications like healthcare, finance, and legal systems, where understanding the reasoning behind a decision is just as important as the decision itself.

1.1.2 Evolutionary Algorithm

Metaheuristic algorithms are special methods used in various areas like economics, engineering, and management to solve complex problems[8]. Many of these algorithms are inspired by natural selection, swarm behavior, and the laws of physics. There are two main types of metaheuristic algorithms (Fig. 1.1): single-solution and population-based metaheuristics.

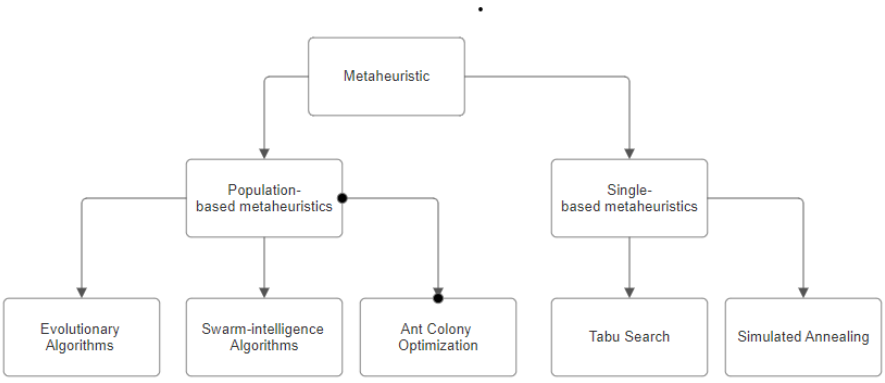


Figure 1.1: Classification of metaheuristic Algorithms

Single-solution metaheuristics: These focus on improving a single solution through local search. However, they can sometimes get trapped in local optima[9]. Examples include simulated annealing and tabu search.

Population-based metaheuristics: These work with multiple possible solutions at the beginning of the algorithm. This approach helps maintain chromosome diversity and prevents the evolution from getting stuck in local optima. These methods are generally better for finding the best solution across a complex set of possible solutions. Some well-known examples are Genetic Algorithms, Particle Swarm Optimization, and Ant Colony Optimization.

Among the population-based metaheuristics, Holland proposed the Genetic Algorithm (GA) in 1975[10]. Inspired by the Darwinian biological evolution process, the GA evolution begins with a population of completely random individuals. The individuals are composed of genes, which are represented in binary or other representations. The fitness of the entire population is evaluated in each generation, and multiple individuals are randomly selected from the current population (based on their fitness) to produce a new population of life through natural selection and mutation, which becomes the

current population in the next iteration of the algorithm.

In 1992, Koza conducted a comprehensive study on Genetic Programming (GP) [11]. Unlike GA, GP uses a tree-type data structure as the chromosome, each chromosome is a computer program, which can represent a solution for a given problem domain. The amount of computation required for GP is so large (dealing with a large number of candidate computer programs) that it could only be used to solve simple problems back in the 1990s. In recent years, with the development of GP and the exponential increase in the computational power of central processors, GP started to produce a large number of remarkable results.

In 2002, Genetic Network Programming (GNP) [12] was introduced. As a special type of GP, it uses a graph-type data structure instead of a tree-type data structure. Compared to GP, GNP is more efficient in solving complex problems in a changing environment. Subsequently, many other flavors of GNP have been developed to enhance its efficiency. These include GNP chromosome structure modification[13], features of mutation and crossover modification[14], and combinations of GNP with other machine learning algorithms such as reinforcement learning, Ant Colony algorithms, and fuzzy logic.

In recent years, GNP has been widely used in many real-life applications, including robot navigation, optimization of financial portfolios, image and signal processing techniques, and data routing for network design. The usage of these techniques is growing in many other fields along with the development of GNP, such as environmental modeling, transportation planning, and optimization of energy systems.

1.2 Scope of Research and Problem Domain

1.2.1 Scope of research

With the development of A.I., many complex and non-linear problems have been solved. However, these solutions, while very powerful, are not human-interpretable; they operate like “black boxes” with millions of neurons and billions of connection weights. The scope of this research is to interpret these ”black boxes” by converting them into a computational graph, which is interpretable and reusable. The proposed approach assumes that we have a set of judgment nodes and processing nodes that can be utilized in a computational graph, tailored to the problem domain. We have only tested the proposed method on the taxi problem, but other similar problems can be tackled as

well (e.g., tile world, maze problem, etc.).

On the other hand, the multi-goal path-finding problem is a classic research problem in GNP. An improved GNP is needed for this kind of question, and the solution should be optimal in real life. The scope of this research includes using GNP and other methodologies to find the optimal solution for a single-agent multi-goal problem in a stochastic environment. The research uses the taxi problem as a case study to conduct a comprehensive analysis. This study defines a methodology for converting a "black box" system into a computational graph representation and solving the multi-goal path-finding problem with an optimal GNP solution in general.

1.2.2 Problem Domain: The Taxi Problem

The Taxi problem [15] is a popular toy problem used to demonstrate and test algorithms. It involves an environment where a taxi must pick up a passenger at one location and drop them off at another within a grid world. The goal is for the taxi to find the shortest, collision-free path from its current location to the pick-up/drop-off location.

Actions:

Basically, The taxi has 6 kind of movements:

1. Move Up (MU)
2. Move Down(MD)
3. Turn Left (TL)
4. Turn Right (TR)
5. Pick Up (PU)
6. Drop Off (DO)

- **Environment Simulation**

  The research uses the 'Taxi-v3' environment provided by the OpenAI Gym library as the simulation platform for the algorithm. The environment described is a 5x5 two-dimensional grid world, featuring walls, obstacles, and four distinct locations marked as R (Red), G (Green), Y (Yellow), and B (Blue), as depicted in (Fig. 1.2).

The taxi must navigate any of the 25 grid squares and solve a total of 500 unique environmental scenarios with six deterministic actions: move south, north, east, west, pick up, and drop off the passenger.

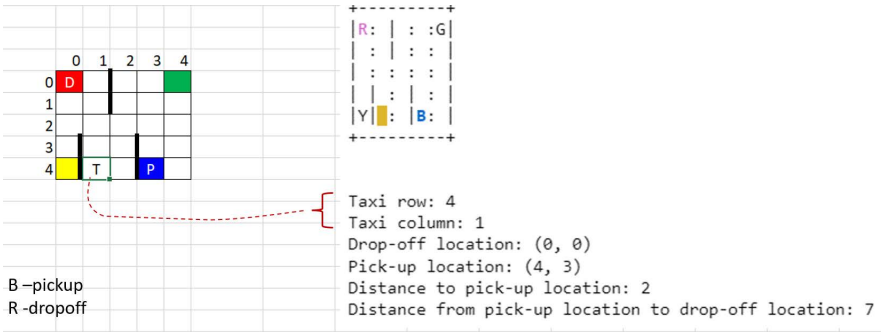


Figure 1.2: Taxi Problem[15]

.

The implementation part of this study is developed using Python with the PYGAD library. The GNP will guide the taxi to find the optimal solutions for different environments.

1.3 Research Objectives

The main objective of this research is to convert a “black box” solution into a computational graph solution. This research focuses on providing a general, innovative approach to achieve this. Additionally, this research develops a novel enhanced version of Genetic Network Programming (GNP) based on the “black box” solution to address the challenge of single-agent, multi-goal path planning in stochastic environments. The goal is to establish a generalized approach for effectively solving this specific type of path planning issue with GNP. Furthermore, the research provides unique techniques aimed at ensuring that the produced GNP solution is optimal for this type of problem. The following outlines the specific objectives:

- Convert a “black box” solution into a computational graph solution.
- Design a solution to simulate the ”black box” system for the specified problem domain.
- Test and validate the generalizability of the designed solution across different scenarios.

- Design the GNP chromosome based on the optimal solution.
- Solve the multi-goal path-finding challenges with our proposed Divide-and-Conquer approach.
- Accelerate the GNP evolution using the optimal solution.
- Develop an improved GNP capable of solving these problems effectively.
- Create a technique to refine the result and generate the optimal GNP solution.
- Conduct tests to assess and validate the generalizability of the solution.
- Define and propose a general comprehensive approach for addressing multi-goal path-finding challenges using GNP.

1.4 Significance of the Research

This thesis can be distinctly distinguished from other GNP works based on the main approach that we have taken (from top to bottom): we started by creating a hand-crafted solution as a stand-in for the "black box" system (top), which guides the evolution of the best GNP individual (bottom). The "black box" system serves as the perfect model and may come in the form of a deep neural network or other advanced algorithms that do not have any explanation capabilities. On the other hand, the target solution is a computational graph that is elegant, highly interpretable, and reusable. Our proposed method details how we can utilize the "black box" system in constructing a chromosome structure, fitness function, and modified crossover and mutation operators for our novel GNP variant. Moreover, there are some challenging multi-agent problems, such as the TileWorld problem, that cannot be solved 100 percent using GNP[14], so resorting to using other advanced algorithms and then converting them to interpretable solutions is an interesting avenue of research to pursue. In this research, the GNP solution will be refined to generate the perfect GNP solution.

This research has the following innovations: First, this study proposes and defines the "Top-Bottom" approach to convert an unexplained but very powerful "black box" solution into a computational graph solution. In previous studies, various methods have been explored to address this problem, but there has been no systematic and effective approach specifically using Genetic Network Programming. Second, this research presents a novel Genetic Network Programming variant that employs an innovative fitness evaluation method, customized chromosome structure, and evolution operators

to accelerate GNP evolution. This enhancement enables quicker decision-making for agents in changing environments. Third, we have developed a new methodology that refines solutions to ensure the generated solution is perfect for the problem domain that we used as a testbed. The methodology can guarantee the generated result is perfect, which is crucial for path planning in real-life scenarios.

1.5 Research Methodology

The research was conducted through a combination of literature review and software development. The methodology and procedure are structured into the following steps:

- Develop a hand-crafted solution as the "black box" system for the research.
- Study and understand the input and output of the "black box" system.
- Conduct a comparative study of the original GA, GP, and GNP, highlighting the differences among these approaches.
- Develop a prototype Genetic Network Programming software to address a simple problem, with a focus on understanding each component of GNP.
- Investigate current multi-goal path planning challenges using GA, GP, or GNP, exploring the methods employed in existing research.
- Break down the taxi problem into smaller, more manageable sub-problems.
- Develop basic GNP software capable of solving these sub-problems in the taxi scenario.
- Analyze how the "black box" system assists the basic GNP in solving the taxi problem.
- Conduct research on fitness functions, particularly those relevant to path planning and multi-goal path-planning issues.
- Enhance the GNP with a unique fitness function and integrate the "black box" system to effectively solve the taxi problem.
- Design a comprehensive test plan to evaluate the proposed solution.
- Observe and analyze the pattern of solutions, verifying them with software implementation.

- Develop an approach specifically designed to refine the solution.
- Conduct extensive testing of the developed approach.
- Summarize the entire research process, including the methodologies and procedures employed.

1.6 Structure of the Thesis

This thesis is structured according to nine main parts, including the Introduction.

- **Chapter 2: Background**

  This chapter introduces the background of GA, GP, and GNP in detail. It summarizes the advantages and disadvantages of GA, GP, and GNP.

- **Chapter 3: Literature Review**

  This chapter provides a comprehensive review of existing literature on different variants of GNP and related works for "black box" interpretation.

- **Chapter 4: Methods**

  This section explains the detailed 'Top-Bottom' approach, which is an approach to convert the 'black box' to the GNP solution. It also explains the hand-crafted solution, which simulates the 'black box' system and serves as the ground truth of this research.

- **Chapter 5: Proposed GNP Variant**

  A detailed exploration of GNP implementation is presented here, including the structure of chromosomes, the development of a fitness function, and the customization of crossover and mutation processes.

- **Chapter 6: Solution Refinement**

  This chapter explains the proposed methodology for refining the GNP solution to an optimal solution.

- **Chapter 7: Experiment**

  This part of the thesis discusses the methods and results of testing and validating our proposed GNP and methodology, ensuring its reliability and accuracy.

- **Chapter 8: Discussion**

  This chapter discusses the results of the experiment and covers any missed details from the previous chapters.

- **Chapter 9: Summary and Future Work**

  The final chapter provides a summary of the findings and contributions of the thesis. It also outlines potential areas for future research and development in this field.

Chapter 2

Background and Literature Review

To better understand the context of this research, it is important to understand some knowledge and background related to this research. In this section, GA, GP, and GNP will be explained and compared with each other. This section not only explains how these evolutionary algorithms work but also summarizes the advantages and disadvantages of GA, GP, and GNP. This section is the foundation of our study.

2.1 Genetic Algorithm

Inspired by the biological evolution process, an evolutionary algorithm called the Genetic Algorithm (GA) was invented by Holland in 1975. It is used to find the optimal solution for a given problem domain. Similar to natural selection, the optimal solution can be evolved from a population of candidate solutions. The evolution process is controlled by parent selection, mutation, and crossover.

2.1.1 Life-Cycle

The figure (Fig. 2.1) shows the evolution life cycle of the GA. There are five phases in the GA life cycle: initial population, fitness function, parent selection, mutation, and crossover. For a certain problem domain, a population is initialized and the population

is composed of a set of chromosomes. The fitness function will evaluate every single chromosome, and for each chromosome, it will return a fitness score. In the parent selection phase, a set of chromosomes will be selected based on the fitness score for the crossover phase. In the crossover phase, a pair of chromosomes will be paired for swapping their genes to generate new offspring. In the mutation phase, one of the genes in the chromosome will be mutated with a certain mutation rate. The new generation is generated after these five phases, and the new generation will be used as the new population for the next evolution life cycle. After several generations, the solution will be found.

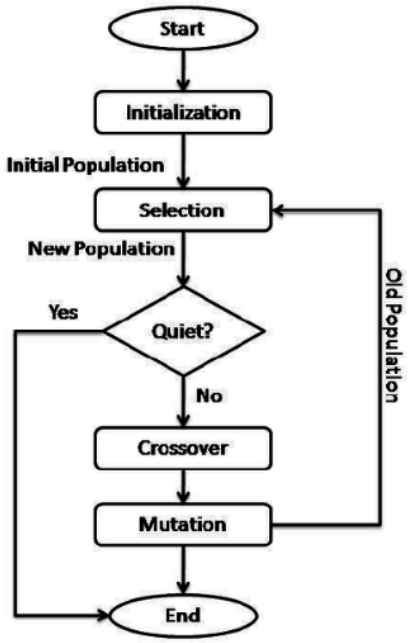


Figure 2.1: GA evolution life cycle

2.1.2 Chromosome Structure

The initial population consists of a set of chromosomes or individuals. Each chromosome represents a solution in the search space for a given problem. The solution is encoded in a certain bit string format, and the encoding scheme depends on the problem domain. For example, in a binary encoding scheme, each chromosome (Fig. 2.2) is a fixed-length vector, and each gene is a one-bit string (either 0 or 1). At the beginning of the GA evolution, the chromosomes in the population are randomly initialized.

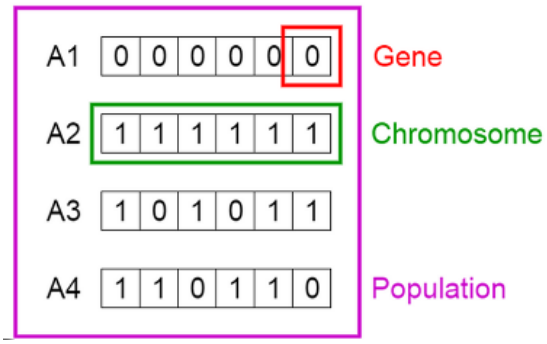


Figure 2.2: GA chromosome structure

2.1.3 Evaluation and selection

- **Fitness**

  The fitness function plays an important role in evaluation; it is used to evaluate the quality of each chromosome. The fitness is a maximization function: the better the quality of the chromosome, the higher the fitness score it has. Depending on the problem domain, different problems use different fitness functions. For example, Manhattan distance and Euclidean distance are widely used in path-finding problems.

- **Selection**

  Parent selection, also known as elitism, selects the chromosomes with the highest fitness scores. The selected chromosomes in the current generation will be used in the next generation. The common parent selection techniques are tournament, rank, roulette wheel, and stochastic universal sampling.

2.1.4 Evolution Operation

- **Crossover**

  In GA, the idea of crossover is to select more than one chromosome from the parent selection step as parents and swap parts of their genes with certain crossover probabilities to produce one or more offspring. The well-known crossover operators are single-point, two-point, k-point, uniform, order, precedence preserving crossover, partially matched, shuffle, and reduced surrogate. Here is an example (Fig. 2.3) of single-point crossover: Given two selected chromosomes, parent A1 and parent A2, some of their genes are exchanged randomly with crossover probability, and two new chromosomes, offspring A5 and offspring A6, are produced.

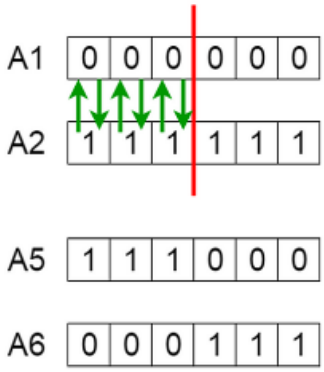


Figure 2.3: GA Crossover

- **Mutation** This concept involves adding random genes to the offspring in a population. This is done to ensure diversity in the population from one generation to the next, which helps prevent the population from becoming too similar too quickly. Here (Fig. 2.4) is what happens after mutation.

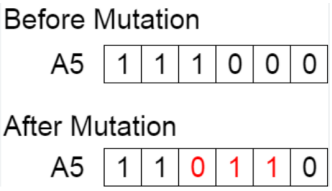


Figure 2.4: GA Mutation

2.2 Genetic Programming

2.2.1 Life-Cycle

GP has the same evolution life cycle, which also contains an initial population, fitness function, parent selection, mutation, and crossover. The main difference between GP and GA is the chromosome structure in the initial population. The chromosome structure of GA is a vector of bit strings, which is randomly initialized according to the problem domain. However, the chromosome structure of GP is a randomly generated computer program. The chromosome for GA is raw data, while the chromosome for GP is processed data.

2.2.2 Chromosome Structure

The representation of the GP chromosome structure is expressed as a syntax tree. The figure (Fig. 2.5) shows the GP chromosome structure. The syntax tree is composed of nodes and links. The nodes are if-else logic functions that indicate the instructions to execute, and the links are terminals that indicate the actions for each instruction. The start node for the chromosome is the tree root, and the terminals are at the deepest level of the tree.

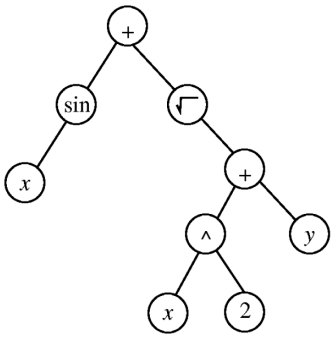


Figure 2.5: GP Chromosome

In Genetic Programming, the chromosome is structured as a tree. This tree comprises a single root node, along with multiple non-terminal and terminal nodes. The non-terminal nodes in GP serve as functions, which can include arithmetic functions, Boolean functions, and conditional statements. The terminal nodes are either inputs to the GP program or are used for specific processing tasks that are relevant to the problem at hand.

2.2.3 Evaluation and Selection

GP utilizes the same techniques for evaluation and selection as GA. The fitness function will evaluate each chromosome and return a fitness score. Chromosomes with higher fitness scores have a higher chance of being chosen.

2.2.4 Evolution Operations

GP employs a variety of genetic operators. These typically include:

- **Crossover:** Sub-trees are exchanged between parent programs.

The crossover (Fig. 2.6) applies to two selected chromosomes to generate two new offspring. During the crossover, a node is randomly picked from both selected chromosomes, and they will swap their picked nodes and their children.

- **Mutation:** Random alterations are made to parts of the tree.

  Similar to GA, the key idea of the GP mutation (Fig. 2.6) is to avoid premature convergence. The algorithm will randomly generate a new sub-tree and replace the randomly selected node with the new sub-tree.

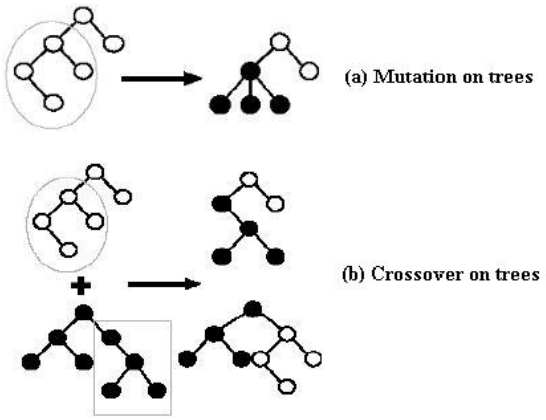


Figure 2.6: GP evolution operations

Compared to GA, GP is another kind of GA. The main difference is that the chromosome for GA is a vector of bit strings (raw data), while the chromosome for GP is a tree-type computer program (processed data). GA has some limitations, while the performance of GP is better in solving some types of problems, such as problems where there is no given ideal solution and problems with a stochastic environment where the situation is constantly changing. For the robot navigation problem, the agent needs to find the path and avoid obstacles. The tree-type chromosome with if-else logic nodes is good for decision-making, as the agent can judge the situation and take the correct action.

The tree-type chromosome has some advantages when solving decision-making issues; however, it still has some limitations. The limitation of GP is the inefficient search for optimal solutions. This inefficiency is primarily due to the "bloat" of GP. In GP, the term "bloat" describes the propensity for the solution search space to grow unnecessarily large. The reason for this expansion is that the GP tree structures can become increasingly deep and complicated over time, presenting the algorithm with a vast number of potential solutions to consider. Because of this, finding the best solutions in GP becomes less effective, especially as the tree's maximum depth grows.

2.3 Genetic Network Programming

As the foundation of our research, Genetic Network Programming (GNP) was introduced as a powerful method for solving more complex decision-making problems in a dynamic environment. GNP is another flavor of GA. The chromosome structure of GNP is also a computer program; however, this computer program is a non-terminal directed graph. To better understand how this design relates to our research, let's delve into the distinct representation and operational mechanisms of GNP as detailed in this paper.

2.3.1 Chromosome Structure

GNP adopts a network structure for its chromosomes. This structure is characterized by two primary types of nodes with time delay: judgment nodes and processing nodes. These nodes in GNP are analogous to the non-terminal and terminal nodes in GP, respectively. The network structure in GNP offers a distinctive approach to problem-solving compared to the tree structure used in GP.

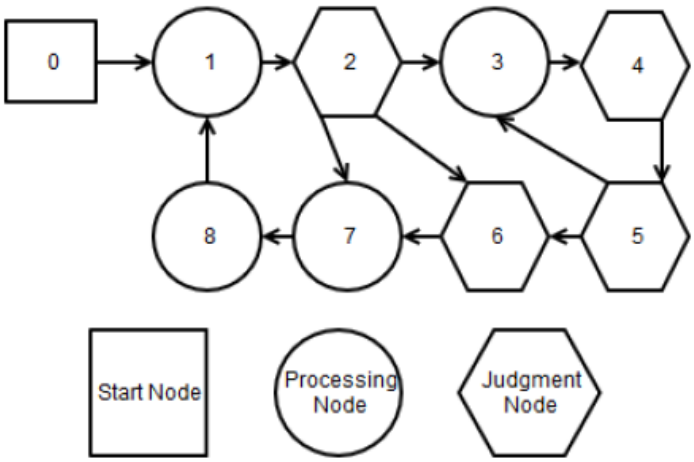


Figure 2.7: GNP Chromosome

Processing nodes are responsible for executing specific tasks or functions, while judgment nodes are used to make decisions based on certain conditions. The time delay is the period from the end of processing in one node to the start in another. During the graph transaction, within a given time frame in a problem domain, judgment nodes decide which node to activate next based on specific conditions, while processing nodes perform certain actions or computations.

2.3.2 Function Library

The function library is a library that contains judgment functions and processing functions. The judgment function responds to decision-making, and the processing function responds to actions. When the network transitions to a certain judgment/processing node, the corresponding judgment/processing function will be called to instruct the agent to judge the current situation or perform a certain action.

2.3.3 Chromosome Transaction

In general, within a given time frame, the graph transaction starts from the start node, which is usually a judgment node. The judgment node will be called based on the current environment, and the judgment will return the information to determine the next node. The time delay for the current node will be deducted from the time frame, and the transaction now moves to node 2. The transaction will stop when the given time is reached.

2.3.4 Evaluation and Selection

GNP use fitness function to evaluate the performance of the chromosome. After the chromosome transaction, the fitness function will assign a fitness score for this chromosome, the value of the fitness score usually depends on the chromosome performance, the feedback from the environment, and the time usage. GNP can use any kind of fitness equation for different kinds of problems. The chromosome with relative hog fitness score will be picked by the selection function, the same as the GA and GP, GNP use the same parent selection techniques.

2.3.5 Evolution Operator

2.3.6 Evaluation and Selection

GNP uses a fitness function to evaluate the performance of the chromosome. After the chromosome transaction, the fitness function will assign a fitness score to this chromosome. The value of the fitness score usually depends on the chromosome's performance, feedback from the environment, and time usage. GNP can use any kind of fitness equation for different kinds of problems. The chromosome with a relatively

high fitness score will be picked by the selection function. Similar to GA and GP, GNP uses the same parent selection techniques.

2.3.7 Evolution Operator

GNP utilizes operators like mutation and crossover but in the context of its network structure. This includes changing links between nodes or altering the functions of nodes themselves. For instance, one mutation type in GNP involves randomly selecting a point in the network and either altering the connection at that node or replacing the node entirely with a new, randomly generated one. This demonstrates the unique network-based approach of GNP compared to the tree-based structure of GP.

- **Crossover**

  As shown in Figure (Fig. 2.8), given two graph chromosomes, Parent 1 and Parent 2, they generate two new offspring, Offspring 1 and Offspring 2. Both parents and their offspring have the same amount and type of nodes. In contrast, node 1 and node 9 for Parent 1 and Parent 2 have different connections. In Parent 1, node 1 is connected to node 2, whereas in Parent 2, node 1 is connected to nodes 4 and 6. Parent 1 and Parent 2 only swap the connections to generate their offspring. In GNP, the crossover operator only swaps the connections between two selected chromosomes randomly with a certain crossover probability.

- **Mutation** The mutation in GNP randomly changes the connections for one of the nodes in the selected chromosome, this will generate one new offspring. In Figure (Fig. 2.9) below, to mutate the randomly selected chromosome, one of the nodes in this chromosome will be selected randomly, and its connection will be changed to connect to another node with mutation probability. The mutated individual will be saved for the next generation.

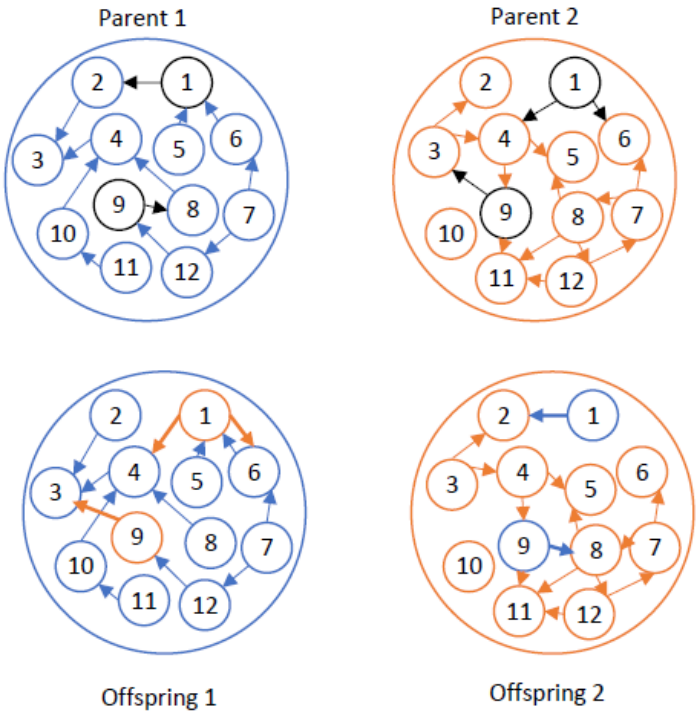


Figure 2.8: GNP Crossover

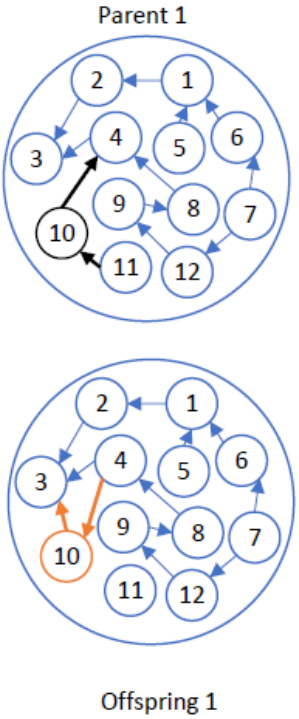


Figure 2.9: GNP Mutation

2.4 Literature Review

In this section, we will conduct a literature review of the relevant GNP variants, including the improved GNP and GNP combined with other algorithms. The scope of the review for this chapter emphasizes how these GNP variants work and where they have been enhanced to improve GNP efficiency. This includes changes to the chromosome structure, modified evolution operators, and the combination of GNP with other algorithms. Through reviewing the literature, we will learn and determine how we want to improve our GNP variant.

2.4.1 Improved GNP Operators and Structure

Although GNP is very powerful compared to GA and GP, it still has some limitations. Recently, there have been significant improvements in GNP, mainly aimed at making it more efficient and effective at solving problems. The concept of "transition by necessity" in GNP was proposed in this paper [14]. In this context, only some judgment functions (necessary judgments) are made to determine actions, where the process of evolution determines the necessity of these judgments. This means that not all judgment functions need to be considered in every scenario, only those deemed necessary through evolutionary processes.

The traditional GNP does not consider "transition by necessity," which leads to two major drawbacks: invalid evolution and negative evolution. Invalid evolution occurs when the genetic operators modify parts of the graph that are not actively used or necessary, while negative evolution indicates a situation where, as the evolutionary process progresses, the fitness scores of the populations decrease instead of improving. These issues can lead to deteriorated evolutionary efficiency.

To resolve these problems, the paper proposes novel evolution strategies with simplified genetic operators. For the crossover, the simplified crossover is restricted to the region of transited branches, where it intersects between crossover individual A and crossover individual B. For the mutation, the simplified mutation is restricted to the region of the mutation individual. These new operators are designed to reduce the negative impacts of invalid and negative evolution, reducing unnecessary transactions in GNP and improving the performance of GNP.

Apart from "transition by necessity," another important feature in GNP is the "Reusability of nodes," which plays a vital role in the GNP graph. This paper [16] addresses this issue and introduces a new method to improve GNP performance.

The "Reusability of nodes" refers to the frequency of node usage during evolution; the more important the node, the higher its usage. In traditional GNP, all nodes are treated equally, regardless of their contribution to successful outcomes, leading to invalid evolution and negative evolution. To increase the usage of important nodes, the paper introduces adaptive Genetic Network Programming. By identifying and focusing on frequently reused, effective nodes, adaptive GNP can enhance evolution efficiency. Adaptive GNP has customized operators that dynamically adjust crossover and mutation probabilities based on the reusability of nodes within the GNP graph. The reusability is quantified by counting the frequency of node usage, influencing the evolutionary pressure for each variable (branch) of the individual GNP. The adaptation process involves assessing each variable's contribution to the solution's fitness, allowing the GNP to self-adjust its evolution strategy.

To enhance GNP and make it more efficient, some studies have modified the mechanism of evolution operators. They have implemented "transition by necessity" evolution and increased the reusability of important nodes. Other research has focused on altering the chromosome structure itself.

In traditional GNP, the judgment nodes, responsible for decision-making, are not fully utilized, and the performance of the evolution highly depends on the usage of these judgment nodes. To address this issue, Genetic Network Programming with Control Nodes (GNPCN) is proposed [17]. The control nodes, similar to start nodes in the original GNP, act as checkpoints, helping nodes transact more efficiently and ensuring all parts of the chromosome are effectively used.

GNPCN is achieved by transferring the activated node to a control node after a certain number of processing nodes are activated. This method ensures that node transactions do not just randomly jump between nodes. Instead, it follows a structured path or branch, where a certain amount of processing is done before moving to the next stage (control node). GNPCN structures the chromosome in breadth and depth; the breadth refers to the number of control nodes, and the depth refers to the number of activated processing nodes. GNPCN evolves both the breadth and depth of the chromosome before returning to a control node, meaning more structured paths or branches will be involved during chromosome evolution.

2.4.2 GNP Combined with Other Algorithms

2.4.2.1 GNP with EDAs

Estimation of Distribution Algorithms (EDAs) [18] are a type of evolutionary computation. Unlike traditional evolutionary algorithms that rely on random methods to mimic biological evolution for creating new populations, this algorithm utilizes a probabilistic model. This model is created using statistical or machine learning techniques to approximate the current population's probability distribution, and then it is used to generate new populations. The life cycle of EDAs can be summarized in the following steps: Firstly, it creates an initial population. Secondly, it selects some elites. Thirdly, it develops a probabilistic model based on the selected individuals. Fourthly, this model is then utilized to create a new population. Finally, the process returns to the second step and repeats these steps until predefined stopping criteria are met.

Evolution operators such as crossover and mutation randomly change and swap genes to evolve the chromosome. However, each GNP gene in the chromosomes is highly dependent on each other, and these operators sometimes might break the useful branch and decrease the evolution efficiency. To overcome this issue, Genetic Network Programming with Estimation of Distribution Algorithms (GNP-EDAs) [19] combines GNP and EDAs, replacing the crossover and mutation with the probabilistic model, which can generate a useful new population by calculating the connection probabilities between different genes.

The probabilistic model is constructed in this way: First, consider a method for estimating the likelihood of connections among distinct nodes within a network. During the t-th iteration, this approach calculates the probability $P_t(i,k,j)$ that a link emanates from the k-th branch of node i to node j. The formula for this probability is given by the ratio of the sum of connection strengths from node i to j over the sum of all connection strengths from node i to any node. Mathematically, it is represented as:

$$P_t(i,k,j) = \frac{\sum_{n \in N} \delta_t^n(i,k,j)}{\sum_{j \in A(i,k)} \sum_{n \in N} \delta_t^n(i,k,j)}$$

Second, both the connection information and transition information between different genes are considered. In a chromosome, not every node might be involved in the problem-solving process. A node transition is determined by choosing the necessary nodes.

Thus, in this method, the transition information is factored into the probabilistic model. The probability that a connection from the k-th branch of node i will lead to node j is dependent on how often such a transition occurs. The more frequent the transition, the more valuable the connection is deemed. The formula for this method is given by:

$$P_t(i,k,j) = \frac{\sum_{n \in N} \delta_t^n(i,k,j) + \eta q_t^n(i,k,j)}{\sum_{j \in A(i,k)} \sum_{n \in N} \delta_t^n(i,k,j) + \eta q_t^n(i,k,j)}$$

where $\delta_t^n(i,k,j)$ represents the connection strength from node i to j at the n-th node in the t-th generation, and $q_t^n(i,k,j)$ accounts for the transition frequency from node i to j.

With the implementation of the probabilistic model, the GNP evolution can generate more useful populations. However, GNP-EDAs have some obvious drawbacks: In EDAs, the distributions of node connections are computed to guide the search process. This leads to the propensity of EDAs to converge on local optima, which means that they can get stuck in what seems like the best solution but is not (local optimum). Another issue is that when calculating these probabilities, the method does not consider how good (or fit) each solution is; it treats the best and the tenth-best solutions the same. Additionally, the method counts every connection from good solutions, even if they do not help make decisions, which can confuse the process of finding the best solution. The biggest problem is that it does not handle loops well. When connections in a loop are used too much (the probability distribution is too high), they seem more important than they should be, which can make the method settle on a local optimum too quickly.

To address the above issues and maintain chromosome diversity, papers [19] [20] used an estimation of distribution algorithm, as in the previous study, to create the next set of chromosomes. They also used reinforcement learning to build a model based on the better solutions found in the group. A detailed analysis of these studies is available in this paper [21]. The paper presents an extended probabilistic model building a genetic network programming (PMBGNP) algorithm to speed up the GNP evolution. The key idea of PMBGNP is to pick the sub-branches of the underperforming (bad) individuals for building a probabilistic model. This model can improve GNP efficiency by using Sarsa learning to calculate Q values to identify the good structure. When using the probabilistic model with RL, if a task is completed successfully, only the final step that achieves the goal gets a reward. This reward is distributed slowly over the system, which can slow down how quickly it learns and evolves. Moreover, these algorithms

become less effective when the problems they are trying to solve get more complicated.

2.4.2.2 GNP with Reinforcement Learning

Integrating GNP with Reinforcement Learning (RL) represents an important advancement in the development of GNP. A significant enhancement was the integration of GNP with Q-learning, marking the first instance of GNP being combined with RL, as introduced by [22]. This was used to refine the transfer rules between states within GNP, and this approach was specifically applied to the prisoner's dilemma problem. Further studies [23] combined GNP with Q-learning to better suit the environment. In other papers [24] [25], GNP was integrated with the SARSA algorithm to enhance the GNP evolution efficiency, particularly in controlling the Khepera robots process.

To make GNP more efficient during evolution, Genetic Network Programming with Reinforcement Learning (GNP-RL) [26] was introduced. GNP-RL combines GNP and RL, where RL can help GNP generate chromosomes with relatively high fitness scores. In the original GNP algorithm, the structure of the graph remains constant during the evolution process and becomes fixed once an optimal individual is chosen during the testing phase. Conversely, integrating GNP with reinforcement learning allows for the alteration of node connections during the testing phase, enhancing the system's performance in changing environments. As an extension of GNP-RL, this paper [27] attempted to use multiple start nodes to derive multiple programs from a GNP structure in parallel.

Chromosome Structure The GNP-RL chromosome structure (Fig. 2.10) contains nodes and sub-nodes. Different from the GNP chromosome structure, each sub-node has a Q-value, which corresponds to each state-action pair. The key idea is to link a specific state to a corresponding action. In this scenario, the 'state' refers to the node ID, while the 'action' pertains to one of its sub-nodes. A node will execute only one of its sub-nodes at a given time, and the choice of which sub-node to execute is determined by the reinforcement learning algorithm. In a certain situation, an agent's possible actions are represented as distinct types of processing nodes within the chromosome. In contrast, the GNP-RL algorithm treats these actions as sub-nodes under a single processing node. The reinforcement learning algorithm will pick one of them to execute.

GNP-RL Transaction The figure (Fig. 2.11) illustrates the GNP-RL transaction, which differs from the GNP life cycle in the following way: Once the initial population

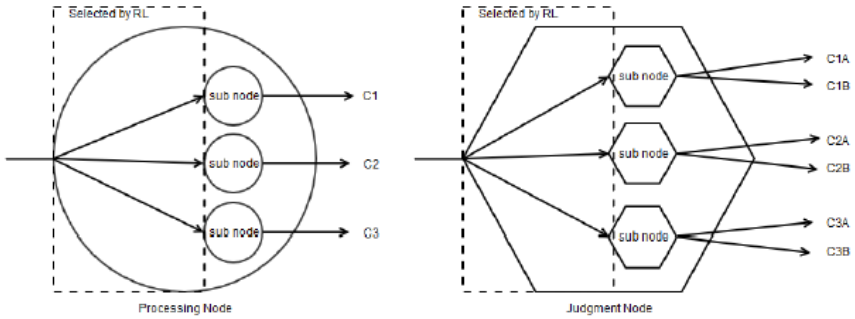


Figure 2.10: GNP-RL chromosome structure

is created, each chromosome is evaluated using a fitness function, and Reinforcement Learning trains the agents to determine the most effective sub-node. In the subsequent generation, the new population will be trained and evaluated by the fitness function again before being subjected to elite selection and evolutionary operations.

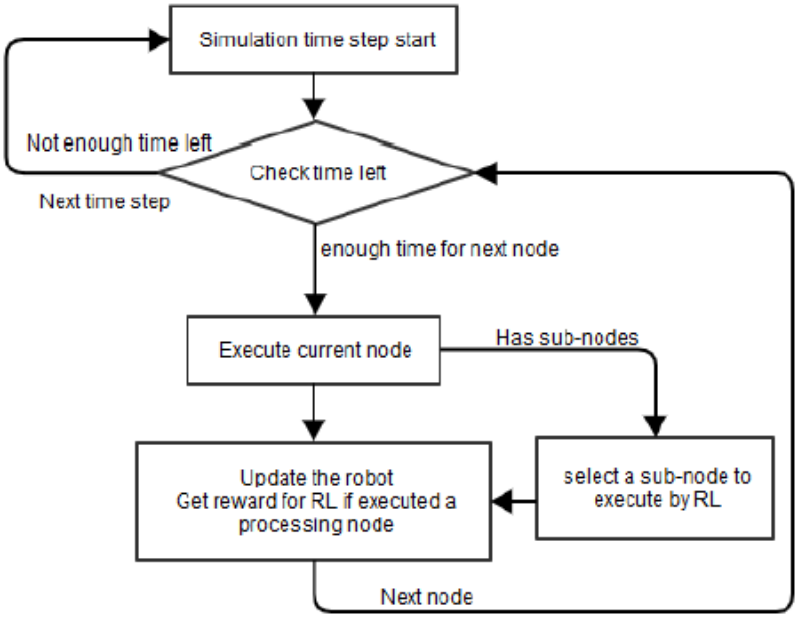


Figure 2.11: GNP-RL transection

2.4.2.3 GNP with Ant Colony Optimization

Integration of GNP with swarm intelligence algorithms is another crucial advancement in the development of GNP. Ant Colony Optimization (ACO) was combined with GNP in [28] [29] for the Evaluator group control system. The unique evolution operation and

pheromone intensity calculation methodology can improve GNP's exploitation ability. In GNP with ACO, the integrated approach dictates that for every ten GNP iterations, one iteration is specifically allocated to execute ACO.

2.4.2.4 GNP with ACO Life Cycle

ACO mimics how ants find food. Ants use a scent called pheromone to mark paths from the colony to food; stronger pheromone trails can attract more ants. As pheromone evaporates over time, less popular routes disappear, helping ants choose the best paths. As the table shows (Fig. 2.12), the similarity between GNP and ACO lies in some common logic. GNP with ACO is proposed [28] to enhance GNP's searchability.

GNP	Ant Colony Optimization
Individuals	Ants
Branch between nodes	Edge
Transition	Tour
Fitness	Total distance

Figure 2.12: GNP vs ACO

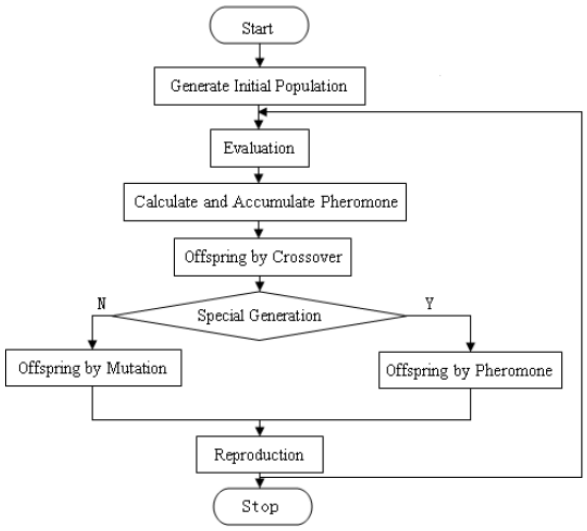


Figure 2.13: GNP-ACO life-cycle

The life cycle (Fig. 2.13) starts with population initialization, where the quantities of different nodes and connections within the chromosome are set randomly. During the pheromone calculation phase, the fitness function is used to evaluate the performance of the chromosomes, and based on this evaluation, along with the transition frequencies, the pheromone levels are calculated with the following equation:

$$h_m^n(i,k,a) = \frac{F^n - f_m^n}{F^n} \cdot \gamma_m^n(i,k,a), \tag{2.1}$$

After updating the pheromone, the life cycle proceeds to the genetic operation phase. In this phase, the generation is separated into general generation and special generation. Crossover occurs conventionally in each generation, with pheromone treated as a branch attribute of GNP. Mutations are typical in general generations. However, in special generations, pheromone information is used to create new individuals instead of mutation. Branches with more pheromone have a higher chance of being passed on to the new GNP individual.

Furthermore, the Artificial Bee Colony (ABC) algorithm was proposed in work [30]. The GNP chromosomes were evolved by an ABC-based evolutionary approach. The ABC algorithm consists of three sequential phases: the Employed bee phase, the Onlooker bee phase, and the Scout bee phase. During the Employed and Onlooker bee phases, individuals within the population attempt to disseminate their knowledge and spawn new individuals. A new individual will only replace its predecessor if it proves to be superior. In the Scout bee phase, any individual who fails to demonstrate improvement over a set number of iterations is replaced by a newly generated individual.

In other research [28] [29], it has been found that algorithms can struggle with more complex problems. To help Genetic Network Programming (GNP) work better, ACO was used to boost its ability to find good solutions. Yet, they did not eliminate crossover and mutations, which sometimes meant losing parts that worked well. Inspired by GNP with ACO, the ACNP algorithm was proposed in [31] to solve these issues. During evolution, the ACNP creates an experience table according to the structure of the chromosome, and a new population is generated according to the current experience table. After parent selection and evolution operations, a new experience table is created. The graph structure of GNP is optimized in this way.

2.4.3 Multi-goal Path-finding problem with evolution algorithm

Path-finding(path planning) and multi-goal problems are critical areas of AI study [32]. Unlike the A* algorithm for solving traditional path-finding issues (path-finding issues with a constrained environment, the evolutionary algorithm has better performance on path-finding for dynamic environments and real-time systems.

2.4.3.1 Travelling Salesman Problem

The Travelling Salesman Problem (TSP) [33] is a classic issue in path-finding. Many researchers use the TSP as a case study for exploring evolutionary algorithms, aiming to find efficient solutions for TSP. [34] uses GA to solve the Unmanned Aerial Vehicle (UAV) problem (finding an optimal solution to a single UAV path is the same as solving a TSP problem). However, the chromosome of GA is a vector of bit strings, and a computer program with an if-else logic type chromosome is more suitable for solving this kind of problem.

To address this issue, [35] uses GP to solve the UAV problem. Overall, path-finding is a multi-objective optimization problem and can be defined by the formula below:

$$\min f(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), f_3(\mathbf{x}))^T \tag{2.2}$$

$$\begin{aligned} f_1(\mathbf{x}) &= \sum_{i=1}^{n} l_i^2, \\ f_2(\mathbf{x}) &= \sum_{i=1}^{n} h_i^2, \\ f_3(\mathbf{x}) &= \sum_{i=1}^{n} f_{T_i}. \end{aligned} \tag{2.3}$$

Where $f_1(\mathbf{x})$ defines the cost function of path distance, $f_2(\mathbf{x})$ defines the cost function of flight altitude, and $f_3(\mathbf{x})$ defines the evaluation value for all threats.

Chromosome Representation The solution is described by a set of feasible functions. The chromosome structure is a syntax tree, where each non-terminal node represents a feasible function. There are a total of N_f non-terminal nodes, i.e., $F = \{f_1, f_2, \ldots, f_{N_f}\}$, and N_{term} terminal nodes, i.e., $T = \{a_1, a_2, \ldots, a_{N_{\text{term}}}\}$.

The purpose of using an evolutionary algorithm to solve the path-finding problem is to find the computer program that can represent the problem's nature, which means the computer program with the most optimal fitness score. Both the input and output of GP are computer programs. Thus, compared to the chromosome of GA, GP is more suitable for solving this problem.

The Generalized Traveling Salesman Problem with Neighborhoods (GTSPN) is an extended problem for TSP. In GTSPN, instead of having discrete cities as in TSP, each cluster consists of a "neighborhood," which is a set of regions or points. This paper

[36] solved this multi-goal path planning problem with a Hybrid Random-Key Genetic Algorithm (HRKGA).

- **Input:** a set $V = \{1, \ldots, n\}$ of the indices of the neighborhood sets S_i, the neighborhood sets S_i, n sets $W_i = \{1, \ldots, m_i\}$ of the indices of the neighborhoods $Q_{i,k}$, and the neighborhoods $Q_{i,k}$.
- **Output:** a minimal-cost cyclic tour $T = \{q_{\tau(1)}, \ldots, q_{\tau(n)}\}$ with objective value L.

GA needs to discover the optimal neighborhood within the neighborhood set. In the chromosome encoding, each gene contains an index k, which is an index for the neighborhood within the neighborhood set, and a vector part q_i, which is the node configuration within each neighborhood. Both k and q_i are randomly generated. Each gene is assigned a random number uniformly and encoded into binary format $[0, 1]$. In general, each chromosome is a linear equation. For instance, in the case of $V = \{1, \ldots, 4\}$ and $W_i = \{1, \ldots, 5\}$, the following chromosome:

$$1q_1.22 \quad 4q_2.72 \quad 2q_3.45 \quad 3q_4.02$$

is decoded into the following route:

$$q_4 \in Q_{4,3} \rightarrow q_1 \in Q_{1,1} \rightarrow q_3 \in Q_{3,2} \rightarrow q_2 \in Q_{2,4}$$

GA and GP are widely used with different equations to solve TSP and related problems. As can be observed, for path-finding issues or multi-goal path-finding issues, the optimal solution is an equation or other type of computer program. GP has better performance compared to GA, as a computer program input is more suitable for this. Therefore, the computer program type chromosome should be adapted. Thus, GP and GNP should be used for the path-finding problem and multi-goal path planning problem.

2.4.3.2 Robotic path planing

Robotic path planning is the problem where the aim is to find the most efficient path for a robot to travel without any collisions. This involves connecting several target

points within the operational area of the robot. This particular challenge is known as a multi-goal path planning problem (MTP), and it is a well-researched area within the evolution algorithm field.

GA This paper [37] implements path planning using GA on mobile robots. The goal is to let GA help the robot find the shortest collision-free path from the current position of the agent to the destination. In GA, each chromosome represents a possible path, which is stored in waypoints. The path is decoded by measuring equal intervals along a straight trajectory from the starting point to the endpoint. As shown in the figure (Fig. 2.14) below, waypoints are stored as pairs of x and y coordinates, which are integers. However, this design is for a constrained environment, not for a dynamic environment.

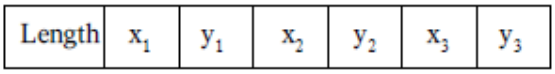


Figure 2.14: Chromosome structure

RTP-GA The paper [38] introduced real-time path-finding with Genetic Algorithm (RTP-GA) to solve the path-finding issue in a changing environment. RTP-GA is GA-based and combines with A*, a heuristic search algorithm, to increase the optimization capabilities and adaptability of GA.

RTP-GA Life Cycle The RTP-GA life cycle is shown below (Fig. 2.15). RTP-GA is integrated with the environment. Unlike GA, the fitness evaluation and parent selection occur after the evolution operation. After parent selection, the agent will take a series of actions and move through the environment. During the movement through the A* phase, Greedy A* is employed for navigating the environment until the target location is farther away than the Manhattan distance of the original location. Thus, it is necessary for the navigating entity to regularly update and track its location on the map and denote the areas it has already explored.

GP As mentioned in the previous section, for path-finding issues, GP has better performance than GA due to the difference in chromosome structure. This paper (ref) implements GP for multi-robot planning. Each chromosome is a computer program, similar to the chromosome in GA. After running the chromosome, it will return a

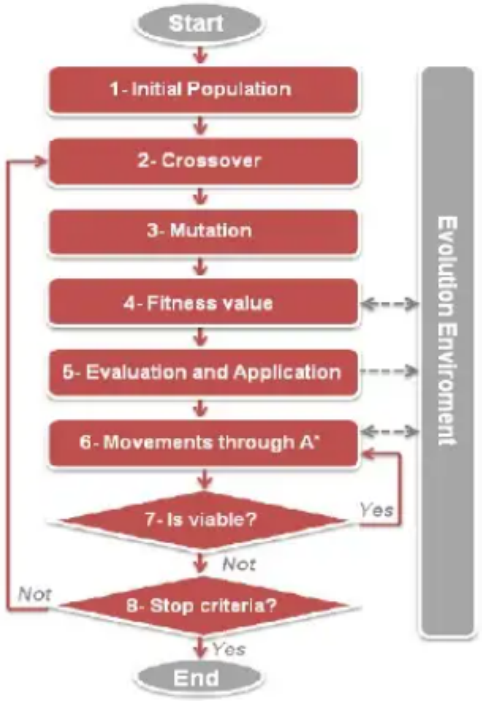


Figure 2.15: RTP-GA life cycle

path. The figure shows the design of the chromosome, where the non-terminal nodes represent the judgment for different situations, and the terminal nodes represent the movement of the agent. To implement an evolutionary algorithm for solving path-finding or multi-goal planning problems, the chromosome represents a solution for a given problem domain; it should be a computer program, where the running result is a path or a mathematical equation.

2.5 Conclusion

In this section, we have reviewed the state-of-the-art GNP variants. There are several directions to improve GNP, such as improving the chromosome structure and modifying the evolution operators. The efficiency of GNP can also be enhanced by integrating with other algorithms, such as GNP with EDAs, RL, and ACO. These algorithms usually help GNP pick the right nodes or actions for the evolution.

To improve the efficiency of our proposed GNP variant, we can modify the chromosome structure, such as reducing the number of nodes, or we can use customized crossover and mutation. Furthermore, we can allow other algorithms to help GNP

navigate the evolution process.

Chapter 3

General Strategies

In this section, we delve into the unique methodology that sets our Genetic Network Programming (GNP) research apart from existing works in the field. Our approach is characterized by a distinctive "Top-Bottom" strategy, where the journey begins with the conception of a complex "black box" solution, which can subsequently guide the evolution towards the most effective GNP individual.

On the other hand, we aim to derive a GNP computational graph solution from the "black box." The objective of this section is to explain the "Top-Bottom" approach, how this research utilizes the "black box" for our novel GNP variant with the "Taxi problem" case study, and our unique version of the "Divide and Conquer" strategy, specially made for multi-goal path-finding challenges.

3.1 Related Work

The object of this research is to convert or interpret the "black box" system into a computational graph. The "black box" system refers to systems that hide their internal logic from the users [39]. The term "interpretation" refers to providing a clear explanation of concepts. In data mining and machine learning, interpretability is defined as the ability to explain or provide meaning in understandable terms to a human [40].

According to these papers [41] [42], an interpretable model has three characteristics: Interpretability, Accuracy, and Fidelity. Humans can easily understand and interpret

these models, which include decision trees, rules, and linear models. A decision tree can be linearized into decision rules using the if-then form [43]:

If $\text{condition}_1 \wedge \text{condition}_2 \wedge \text{condition}_3$, then outcome.

This paper [7] categorizes various approaches to explain the "black box." In general, the "black box" involves transforming the complex AI model into a block of "IF-THEN" logic, which is human-readable. The paper broadly categorizes the approaches into three main types:

- **Model-specific explanations**: This approach focuses on interpreting specific types of models, such as deep neural networks or decision trees.
- **Model-agnostic explanations**: This approach provides insights that can be applied to different types of "black box" models.
- **Techniques focusing on specific aspects of interpretability**: This approach helps us understand how the input affects the output of the model.

The work in [44] provides a comprehensive review of how GP can be leveraged to enhance the interpretability of machine learning models. The focus is on two main categories: The first category is intrinsic interpretability, which seeks to directly evolve more effective (and interpretable) models through GP. The second category is post-hoc interpretability, which uses GP to explain other black-box machine learning models or the models that have evolved from simpler models, like linear models.

The XGP review section discusses intrinsic interpretability XGP approaches that take into account model complexity (including size), number of features, meaningful combinations between features, and questionnaire-based interpretability measures.

XGP focuses on the following properties to improve interpretability:

Smaller GP Model Size: Smaller GP models have better simulatability. To achieve this, GP can be modified to reduce the size of the chromosome, penalize large models during fitness evaluation and parent selection, and utilize customized operators.

Lower (Nonsize) GP Model Complexity: This can be categorized into Structural Complexity Reduction and Functional Complexity Control. The structural complexity of models directly impacts their interpretability. Simplicity in model structure can improve understanding of feature importance, interactions, and prediction inference. The following methods are used: [45] [46] [47]. To reduce Functional Complexity, the following methods are used: [48] [49].

Fewer Distinct Features in GP Models: The GP model simplifies interpretation by using fewer distinct features and concepts. Feature selection and feature construction can be used to address this problem.

Interpretable GP (Sub-)Model Structures: Existing strategies fall into three categories: interpretable function nodes, interpretable combinations between nodes, and problem decomposition.

The post-hoc interpretability methods focus on explaining the "black-box" models. These methods are divided into two main categories: global and local interpretability. Global interpretability methods use GP to generate interpretable models that explain the overall behavior of the black-box model across all instances. Local interpretability, on the other hand, seeks to explain the behavior of the black-box model for specific instances, resulting in explanations that focus on the immediate context surrounding a given data point or decision.

In our research, the "black box" will be converted to a GNP computer graph rather than an "IF-THEN" logic system or using GP. The GNP graph is more elegant in terms of reusability, modularity, and scalability:

- The chromosome structure of the GNP is a network. The GNP does not experience "bloat" as the problem's complexity increases.
- The GNP chromosome has better interpretable function nodes and interpretable combinations between nodes. The GNP comes with a function library, which consists of judgment functions and processing functions, achieving better interpretability in this way.

To enhance the GNP interpretation further, this research will utilize the "Top-Bottom" approach with the GNP to generate the GNP computer graph, and we will use the "divide and conquer" approach to decompose the complex problem into multiple sub-problems.

3.2 Top-Bottom Approach

3.2.1 "Black Box" Conception

In this context, the complex solution (or target model) serves as a "black box"; therefore, we do not examine its computational components, which could include millions of

neurons and billions of connection weights. We can only access the inputs and outputs of the "black box" system, nothing more. The target model knows the correct action for any given scenario but does not provide any reasoning for its actions. This solution is often embodied by advanced algorithms like neural networks, known for their exceptional ability to handle complex and nonlinear problems. Although they are not completely transparent or easily interpretable, they were chosen for their ability to guide us to a highly reusable and interpretable computational graph.

3.2.2 Bridging "Black Box" and Computational Graphs

The "Top-Bottom" approach is the bridge in this research that connects the "black box" with computational graphs. The cornerstone of this methodology lies in transforming the conceptual "black box," often represented by complex algorithms, into a format that aligns with the reusable, interpretable nature of computational graphs. This transformation involves a series of steps:

- **Abstraction and Decomposition**

  Initially, the complex problem is abstracted and decomposed into fundamental components or sub-problems. This process involves identifying key operations, decision-making processes, and data flow within the solution. For instance, in the multi-goal path-finding problem, the decision-making for different cases and the related actions will be identified, such as when the agent should move and the transitions between them (what action the agent will perform when encountering certain scenarios). This involves careful design to maintain the integrity of the solution's logic and performance. The primary challenge is to ensure that the computational graph remains functionally equivalent to the original solution.

- **Mapping to GNP representation**

  The GNP solution will be designed based on the "black box" system. This critical step converts the complex solution into a visual and structured format. Once the action and judgment parts in the problem domain are defined, the judgment and processing nodes will be created, and the chromosome for GNP will be designed accordingly.

  During the parent selection process, the fitness function will utilize the black box system to help the GNP evolve. Additionally, the crossover and mutation will swap and alter specific genes to guide the evolution in the right direction.

- **Iterative Refinement** The computational graph is then iteratively refined to

ensure that it accurately represents the complex solution while also benefiting from GNP's evolutionary processes. The evolved solution may contain duplicate or unused branches. To create the perfect solution, our proposed method will be used to further refine the generated solution.

3.2.3 Benefits in GNP Research

The top-bottom approach in Genetic Network Programming (GNP) offers a novel perspective on solving complex problems. While it presents several advantages:

- **Guided Evolutionary Process** Starting with a conception of the ideal solution provides a clear target for the evolutionary process. It helps in directing the evolution of GNP individuals more purposefully towards this target, potentially reducing the time and resources spent on aimless exploration. This focused direction can lead to more efficient convergence towards effective solutions.
- **Benchmarking Excellence**

  Having a defined "black box" solution serves as a benchmark for evaluating the progress and performance of evolving solutions. It allows researchers to measure how close each generation of GNP individuals gets to this ideal solution. The GNP design will also benefit from this, such as customized chromosomes and evolution operators, which can help the evolution avoid randomly generated bad genes.
- **Adaptability to Complex Problems**

  Complex problems often require complex solutions. The black box model solution enhances the adaptability of GNP, allowing it to handle more challenging scenarios within dynamic environments. This adaptability is crucial in fields where problems are not just complex but also continuously changing.

3.3 Sources of Ground Truth for the Problem

Here are the two possible approaches for obtaining the optimal GNP solution, or the ground truth for the problem. The key idea is to use the ground truth to guide the algorithm for transforming a string of judgment/processing node executions into a GNP solution.

3.3.1 Approach 1: Handcrafted Solution Dataset

The first approach starts with a handcrafted solution. This handcrafted solution receives the current state of the environment to determine the best series of actions. The steps are described as follows.

- **Input State Processing**: The state of the current environment is input into the handcrafted solution.
- **Node Execution**: Feed inputs to the handcrafted solution to get the best action. Based on this input, a sequence of nodes will be executed by the handcrafted solution. These executed nodes are the correct actions for the current states. After running the handcrafted solution, a list of executed nodes will be generated.
- **Generating a Dataset**: As the handcrafted solution runs against 400 different environmental scenarios in the training phase, it creates a 400-node execution list. These lists can comprise a rich dataset.
- **Pattern Recognition and Duplication Handling**: Within this dataset, it can be observed that some sequences of node executions are repeated with certain patterns. By identifying unique partial solutions and applying specific techniques, these patterns can be analyzed and distilled.
- **Deriving the Perfect GNP Solution**: By identifying patterns and unique sequences, we can refine the dataset using our proposed algorithm and formulate a 'perfect' GNP solution.

3.3.2 Approach 2: GNP Solution Dataset

- **Run our GNP Across Many Scenarios**: The GNP system is put to the test across 400 different cases. This is similar to the first approach where a handcrafted solution was tested.
- **Record Node Activation Sequences**: For each current state of the environment, the GNP executes a series of nodes. For example, given input A, it might run through nodes J1, J3, J4, and then P2. These sequences are the responses of the GNP solution, and they represent partial solutions to that particular environment.
- **Compile a Dataset**: The sequences of executed nodes in response to each environment are appended to the dataset. This dataset is a combination of the GNP's decisions and actions.

- **Analyze for Patterns**: Similar to the first approach, the pattern of executed nodes will be identified. We will use our proposed algorithm or approach to convert the string of judgment/processing node executions into an optimal GNP solution.

3.4 Divide and Conquer

3.4.1 Introduction to the Customized Strategy

The Divide and Conquer strategy, a timeless algorithmic approach, has been uniquely adapted in our research to address the complexities of multi-goal pathfinding problems within the GNP framework. This adaptation plays a crucial role in complementing the top-bottom approach, offering a structured method to break down complex, multi-goal problems into more manageable sub-problems.

3.4.2 Complementing the Top-Bottom Approach

Alignment with Ideal Solutions: The Divide and Conquer strategy aligns seamlessly with the top-bottom approach, where the 'perfect' solution is conceptualized first.

3.4.3 Implementation in GNP

Decomposition of Multi-Goal Challenges: In the context of GNP, the Divide and Conquer strategy breaks down the ideal solution into smaller, more understandable chunks, each representing a sub-goal in problem-solving. In the taxi problem, this strategy divides the ideal path into sections, each focusing on a specific sub-goal, such as reaching a passenger or getting to a specific drop-off location.

The Divide and Conquer strategy is essential for simplifying and managing complex tasks such as the taxi problem. Without the Divide and Conquer approach, every decision impacts the entire task, as each action must be evaluated not only on its immediate merits but also on how it affects the overall process. For example, in the taxi problem, the action of dropping off a passenger is linked to the preceding action of picking them up; if the taxi does not pick up the passenger first, it cannot drop them off.

The Divide and Conquer strategy, on the other hand, breaks down a large problem into smaller, separate parts, significantly reducing the problem's complexity. This separation enables simpler, more focused strategies for each component. Furthermore, it provides a clear framework for troubleshooting and refining strategies, as issues can be isolated and addressed within their specific sub-task. It also aids in quickly resolving problems in one part without disrupting the others.

Evolution of Specialized Sub-Solutions: For each sub-goal, specialized GNP individuals are evolved. These individuals are optimized to solve specific aspects of the path-finding problem, ensuring efficiency and effectiveness in each GNP evolution.

In the previous section, we discussed several methods for increasing the efficiency of GNP. The first method involves modifying the chromosome to reduce the number of nodes, thereby reducing computational complexity. The second method involves refining the evolution operators, which facilitates convergence. Integrating GNP with the Divide and Conquer strategy complements these gains. Using Divide and Conquer allows the chromosome to eliminate unnecessary nodes. This means that GNP evolution only involves essential nodes for each sub-goal, and customized evolution operators will only introduce the appropriate genes, making the evolution process more streamlined and effective for complex problems.

Integration of Sub-Solutions: Once optimal or near-optimal solutions are evolved for each sub-goal, the partial solutions will be combined to form a comprehensive solution for the entire multi-goal path-finding problem. However, this combined solution is not perfect and may include redundant parts. To address this problem, the generated solution will be refined based on the "black box" solution. By constantly refining against an ideal model, we improve the solution's applicability to complex path-finding challenges. This process ensures that the final solution is effective for each sub-goal and optimal overall.

3.5 Hand-crafting a GNP Solution

As a stand-in for the perfect solution, we have hand-crafted a solution in the form of a computational graph. This conveniently provides us with the ground truth for all possible scenarios and allows us to verify the conciseness of our derived graph solution. However, during the transformational process, the target model is only used as a fitness function to guide the creation of a computational graph.

3.5.1 Design Principle

The design is to let the agent rely on continuous feedback from the environment to inform its decisions and actions. The design reflects a cycle of perception and action, which is a fundamental principle in robotics and artificial intelligence. This model underscores the "black box" solution design for the intelligent taxi, where it constantly updates its state based on its past and current positions and uses a combination of sensory data and pre-computed knowledge (like heuristic distances) to navigate towards a goal.

3.5.2 Decision-Making Process Outline:

The design principle is shown in Figure 3.1: A looped arrow with the phrase "Sense, decide, act," represents the continuous cycle of the agent's process to make decisions. Below this, a three-step sequence is detailed:

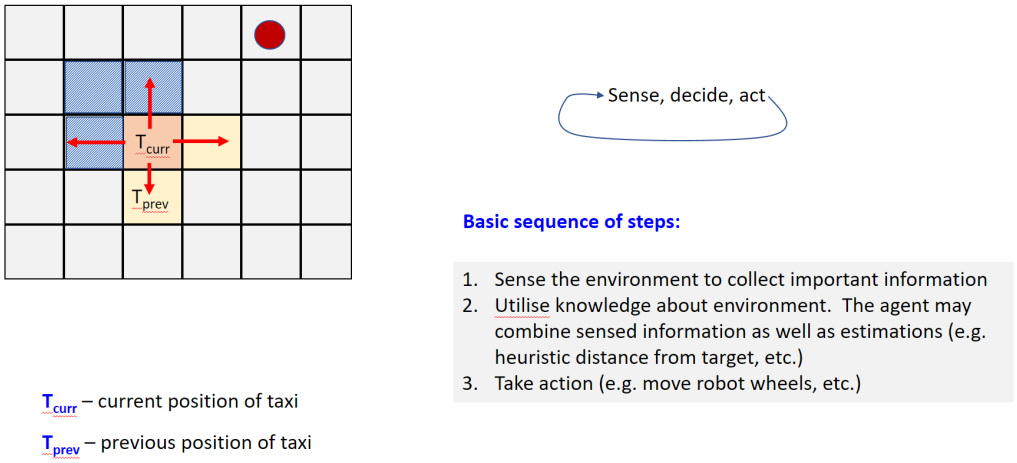


Figure 3.1: The continuous decision-making cycle of an agent in the Taxi Problem.

- **Sense**: The agent senses the environment to collect important information. This is likely the step where the agent acquires data from its surroundings, which could be the immediate grid squares it can move to.
- **Decide**: The agent utilizes knowledge about the environment. It may combine sensed information with estimations (like heuristic distance from the target) to make informed decisions. This implies a level of reasoning or computation, such as path planning or avoiding obstacles.

- **Act**: The agent takes action, such as moving robot wheels or, in this context, moving to another grid position.

In the Taxi problem, the objective is to have the taxi pick up a passenger and then drop them off at a specified destination. This multi-goal path-finding challenge involves treating the location of the passenger as the initial destination and the drop-off point as the final destination. To navigate the taxi toward each destination, the position of the taxi is compared with that of the current destination to determine what action to take.

In this context, a three-step sequence is detailed:

- **Sense**: The taxi will collect the current state of the environment, such as the position of the taxi and the goal.
- **Decide**: The taxi will utilize the knowledge of the environment, especially comparing the position of the taxi and the destination. For example, if the destination is to the southeast of the taxi, the taxi should move south and east. Additionally, the taxi should also judge whether it can move south or east. If there is a wall in front of the taxi to the south or east, the taxi should choose another direction to avoid the wall.
- **Act**: Based on the decision, the taxi will move in a certain direction, pick up, or drop off the passenger.

As observed, the key logic of the design is the IF-ELSE-THEN logic. We can use judgment functions to perform the IF-ELSE feature and processing nodes to perform THEN logic. For instance, if the passenger is north of the taxi, the judgment function will assess the positions of the taxi and passenger and make the decision, while the processing function will move the taxi. Based on this, the following judgment and processing functions are designed as follows:

- *J1* - Should the taxi pick up or drop off the passenger?
- *J2* (Drop off branch) - Compares the terminal/taxi location and determines the appropriate direction for the taxi or if it should drop off the passenger when they are at the same location.
- *J3* (Pick up branch) - Compares the passenger/taxi location and decides the appropriate direction for the taxi or if it should pick up the passenger when they are at the same location.

- *J4* (Should Move East Branch) - Decides whether the taxi should move east.
- *J5* (Avoid the obstacle to the East) - Decides whether the taxi should move up or down if there's a wall to the east of the taxi.
- *J6* (Should Move West Branch) - Decides whether the taxi should move west.
- *J7* (Avoid the obstacle to the West) - Decides whether the taxi should move up or down if there's a wall to the west of the taxi.
- *P1* - Directs the taxi to move north.
- *P2* - Directs the taxi to move south.
- *P3* - Directs the taxi to move east.
- *P4* - Directs the taxi to move west.
- *P5* - Instructs the taxi to pick up the passenger.
- *P6* - Instructs the taxi to drop off the passenger.

Each function is a void type function. When a certain function is called, that function will perform a judgment or processing feature and return a string, which is the next function to be called. For example, when function P1 is called, the taxi will move north, and P1 will return to J1. J1 will be called to judge whether the taxi should pick up or drop off the passenger. If the taxi still needs to pick up the passenger, J1 will return "J3," etc. The details of these functions will be explained in a later chapter.

3.5.3 Graph representation

We will get a computer program if we connect all the judgment and processing functions. To visualize this computer program, each node will represent a function, and the whole computer program will be a directed graph. As seen in the image below (see Figure 3.2), the judgment nodes represent nodes with judgment functions, the processing nodes represent nodes with processing functions, and the connections represent how these nodes relate to each other.

The graph begins at node J1, which judges whether the taxi should pick up or drop off the passenger. If the taxi should pick up the passenger, the graph goes to the *J3* branch; otherwise, it goes to the *J2* branch. After this, the program will compare the position of the taxi and the passenger. If they are at the same position, processing

node *P5* will be executed to pick up the passenger. If the passenger is east of the taxi, judgment nodes *J4* and *J5* will be executed to judge whether the taxi should move east and if there is a wall in front of the taxi. If the path is clear, processing node *P3* will be executed to move the taxi east. All the processing nodes point to *J1*, which is the start node of the graph.

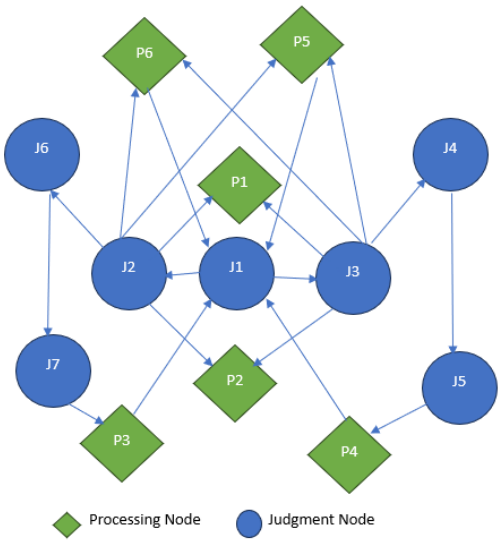


Figure 3.2: The visual representation of a computer program as a directed graph.

3.5.4 Implementation

Since we have all the judgment/processing functions and the complete graph, we need to implement the computer program to execute the complete graph to solve the taxi issue. To simulate the GNP graph transaction, the implementation is divided into two parts: the function library and the computer program.

Function Library

The function library is simply a library that contains all the judgment and processing functions. It functions more like a search engine: when the graph needs a certain function, the function library will find its directory according to the function name, and this found function will be called. The function library (see Figure 5.2) is a key-value pair data structure, using an integer number as the key and the function name as the value. When the key is found in the function library, the corresponding function will be called.

Computer program

```
functionsLibrary = {
    1: J1,
    2: J2,
    3: J3,
    4: J4,
    5: J5,
    6: J6,
    7: J7,
    8: P0,
    9: P1,
    10: P2,
    11: P3,
    12: P4,
    13: P5
}
```

Figure 3.3: Function library.

The computer program is just a loop which can execute the judgement and processing function iteratively. As we can see from the Pseudo code below (Algorithm 1), the implementation is a while loop. It will keep calling judgment and processing functions until the taxi has dropped the passenger successfully. The *current* is a pointer, which points to the current executed node. With the given total time frame, each processing and judgement function will take a certain time to execute, the loop will stop when it reaches the given time frame.

Algorithm 1 Hand-crafted Solution

```
1: current ← J1
2: while not reached the given time frame do
3:     if taxi dropped the passenger then
4:         break
5:     end if
6:     current_function ← functionsLibrary[current]
7:     returned_value ← current_function()
8:     time ← time + used_time
9:     current ← returned_value
10: end while
```

Algorithm walk through

Here is a snapshot (Fig. 3.4) of utilizing Algorithm 1 to pick up the passenger successfully. On the left-hand side of the snapshot is the taxi map; the yellow object represents the taxi, and the blue letter represents the pick-up location. On the right-hand side of the snapshot is the output of Algorithm 1. J1 (Should a taxi pick up or drop off the passenger) will be executed in the first step, and the related judgment function will be called (referring to step 6 in Algorithm 1). A value is returned, which is the next node to be executed. In this case, J3 (Pick up branch) is returned, which

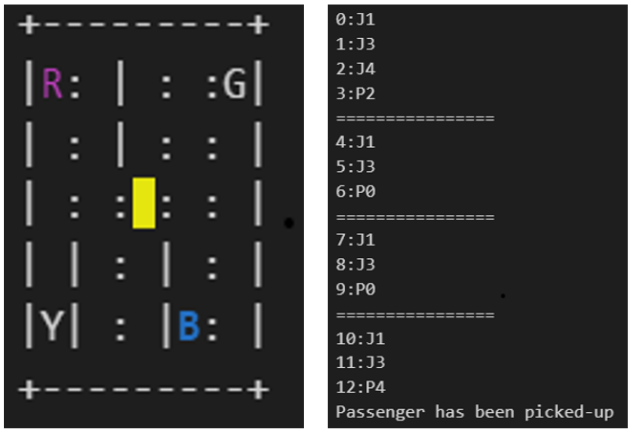


Figure 3.4: Function library.

means the taxi should pick up the passenger. The current node is set to J3, and the judgment function J3 will be called. The function will take the current state of the map as the input and return the next node to be executed. Therefore, J4 (Should Move East Branch) will be executed. As we can see from the map, the pick-up location is east of the taxi, and there is no obstacle. P2 (move taxi to the East) will be executed. After certain steps, the taxi will pick up and drop off the passenger. (The details of the judgment/processing functions will be explained in the proposed GNP Chapter.)

In this chapter, we established a handcrafted solution as the 'black box' solution for our 'Top-Bottom' approach. As the 'Top' part, it adapts the fundamental principles in robotics and artificial intelligence, iteratively receiving information from the environment and performing a series of actions. The handcrafted solution is structured around different judgment and processing functions, each tailored to specific scenarios within the Taxi Problem. As the ground truth of the 'Top-Bottom' approach, this research utilizes this handcrafted solution as the 'black box' solution to develop our GNP variant and the proposed algorithm in later sections.

Chapter 4

Using a Black Box as a Fitness Function in a GNP

4.1 Introduction

This section explores our proposed new variant of GNP in detail. It utilizes the 'Divide and Conquer' strategy to solve the Taxi problem, as well as a black box system to guide the GNP. This GNP is built based on the design of our hand-crafted solution. The focus is on several key aspects:

- **GNP Life-Cycle**: We will explain the details of the evolutionary cycle and how the 'black box' solution interacts with our GNP variant.
- **Chromosome Structure**: Based on the 'black box' solution, the problem is separated into "pick-up" and "drop-off" sub-problems. These two sub-problems will be explained in both genotype and phenotype.
- **Customized Evolution Operators**: These are specially crafted based on the 'black box' solution to guide the evolutionary process effectively toward the ideal solution, ensuring meaningful progress in each evolution step.
- **Innovative Fitness Function**: In this fitness function, we combine the fitness function with other algorithms and utilize the 'black box' solution to guide and evaluate each individual.

We will delve into each of these parts and show how they work together to make our new GNP variant solve the multi-goal path-finding issues efficiently.

4.2 GNP Life-Cycle

Different from other GNP variants, our GNP utilizes the "black box" solution to guide the evolution. With the "Divide and Conquer" approach, the multi-goal path-finding problem is separated into smaller sub-goals, such as the Taxi Problem being separated into the pick-up passenger part and the drop-off passenger part. Our GNP will solve these two small problems individually. This flow chart (Fig. 4.1) shows the life cycle of our GNP variant.

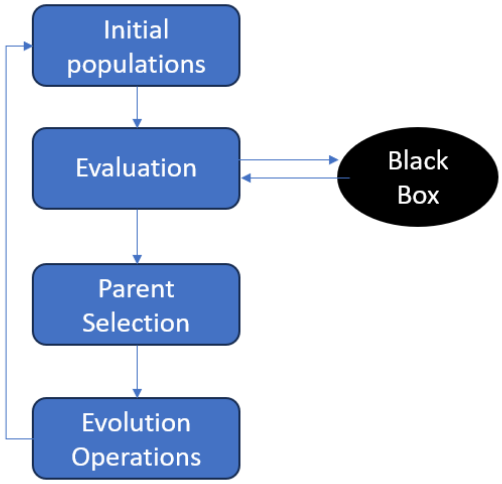


Figure 4.1: Proposed GNP variant life cycle.

- **Initialization:** Similar to other GNP variants, we initialize a set of individuals at the beginning of evolution for both pick-up and drop-off sub-problems.
- **Evaluation:** All the initial chromosomes will be sent for evaluation. During the evaluation phase, the fitness function will utilize the 'black box' solution. In the original GNP, for each chromosome, the fitness function will execute a node and update the environment. Once this node is executed, it will find the executable node in its connections. If the fitness function can find the executable node in its connections, that executable node will be executed. If the fitness function cannot find the executable node in its connections, the evaluation process for this

individual will stop, a fitness score will be assigned to this individual, and the fitness function will evaluate the next individual.

During the evaluation phase, the fitness function will also find the executable node in the connections of the executed node. If it finds the executable node, that node will be executed. If not, the fitness function will extract the information from the 'black box' solution to find the next possible actions. If this possible action is in the connections of the executed node, that possible action will be executed. Otherwise, the fitness function will evaluate the next individual.

- **Selection:** The individuals with the highest fitness scores are selected to be parents for the next generation.
- **Genetic Operations:** The selected individuals will go through the crossover and mutation process. With the knowledge of the 'perfect' solution, the customized genetic operators will be applied to enhance the efficiency of the evolution.
- **Next Generation:** The offspring from the crossover and mutation processes form the new generation. This new generation inherits characteristics from the previous generation.
- **Termination:** This cycle repeats for several generations. The termination criterion might be set to 50 generations or until a path meets a predefined fitness level.

4.3 Individual Structure

In the previous section, the handcrafted solution could solve the Taxi problem efficiently with 13 nodes, with node 1 always serving as the start node. Nodes 1 to 5 (J1 to J7) correspond to judgment nodes, and nodes 8 to 13 (P8 to P13) represent processing nodes. However, we want to speed up the evolution process; therefore, we need to reduce the number of nodes within each chromosome.

The original handcrafted solution has 13 nodes, and the newly refined graph has 11 nodes. The new graph is shown below (Fig. 4.2): Node J4 (Is there any obstacle to the West of the taxi) and Node J5 (How to avoid the obstacle to the West) can be combined to generate the new node J4 (Move west branch). The combined node has the following feature: it will judge if there is an obstacle to the west of the taxi. If there is no obstacle, it tells the taxi to move west. If there is an obstacle, it tells the taxi where to move to avoid the obstacle. Likewise, Node J6 (Is there any obstacle

to the East of the taxi) and Node J7 (How to avoid the obstacle to the East) can be combined to generate the new node J5 (Move east branch).

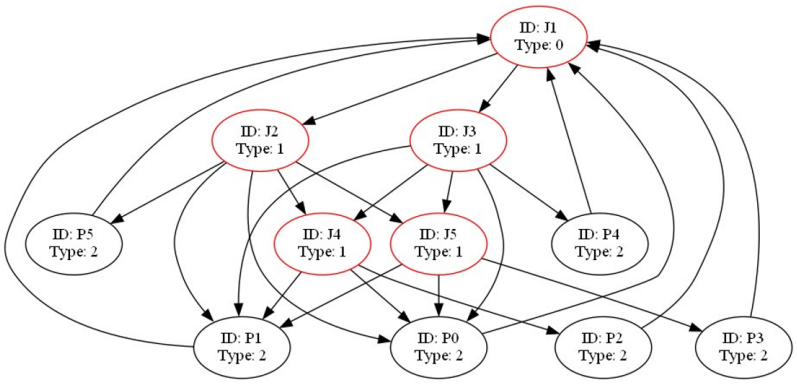


Figure 4.2: GNP graph

Since we use the 'Divide and Conquer' approach to separate the Taxi problem into the "pick up" problem and "drop off" problem, the newly refined graph will also be split and refined for each sub-problem, and some of the nodes can be deleted.

Node J1 (Should pick up or drop off the passenger) can be deleted for both sub-problems. The algorithm does not need to judge whether the taxi should pick up the passenger or drop off the passenger, since the first task is just to pick up the passenger and the second task is just to drop off the passenger.

For the "pick up" sub-problem, judgment node J3 (Should drop off the passenger) will be deleted since we do not need to judge the drop-off situation in the "pick-up" problem. Processing node P11 (Drop off the passenger) will also be deleted.

Likewise, for the "drop off" sub-problem, judgment node J2 (Should pick up the passenger) will be deleted. Processing node P10 (Pick up the passenger) will also be deleted.

In this section, we delve into the detailed structure of GNP chromosomes, focusing on both genotype and phenotype representations, as applied to the taxi problem involving specific tasks like 'pick up' and 'drop off'. The initial population is constructed based on our handcrafted solution, and the 'Divide and Conquer' approach will be applied to solve the taxi problem.

4.3.1 Phenotype Structure

The phenotype in a GNP context refers to the physical or observable characteristics of the network, represented here as a directed graph. The structure of phenotype and functionality is crucial for understanding how the GNP executes specific tasks.

Illustration for "Pick Up" Sub-Problem:

Directed Graph Structure: The phenotype is depicted as a directed graph (Fig. 4.3). This graph showcases the flow of decision-making in the task.

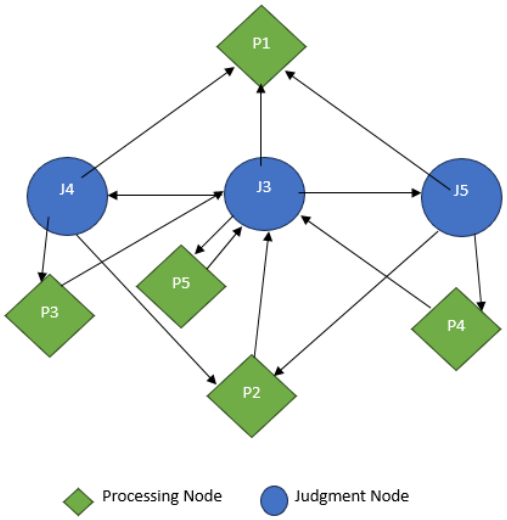


Figure 4.3: Graph structure for "pick-up" sub-problem.

- **Start Nodes:** These nodes initiate the decision process. In our model, a start node always points to a judgment node (e.g., J3) to evaluate the initial condition: 'Should the taxi pick up the passenger?'
- **Judgment Nodes:** These nodes are responsible for assessing various conditions. J4 and J5 judge the situation on the west/east of the taxi respectively. For example, J4 judges if there is no obstacle to the east of the taxi, the taxi moves to the east (processing node P3 is executed).
- **Processing Nodes:** These nodes dictate the taxi's actions based on the judgment nodes' decisions. They loop back to judgment node 3, creating a dynamic decision cycle without terminal nodes.

"Drop Off" Chromosome Structure

While similar to the 'Pick Up' chromosome, the 'Drop Off' chromosome (Fig. 4.4) features 8 nodes, including 5 processing nodes for directional movement and 3 judgment nodes. The only difference is that instead of using judgment node 3 (J3), it uses judgment node 2 (J2) to judge "Should drop off the passenger?"

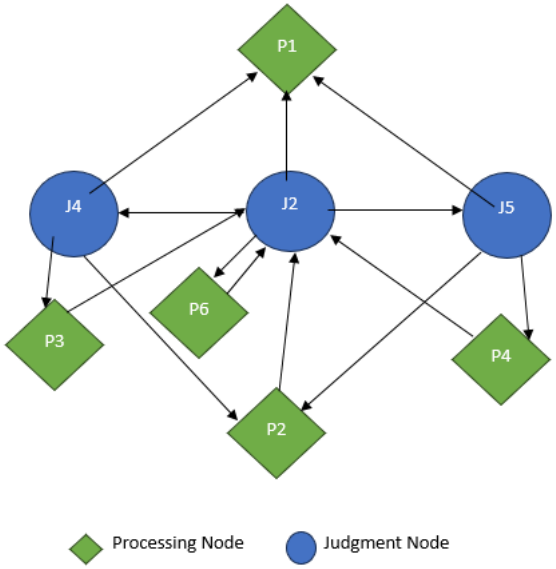


Figure 4.4: Graph structure for "drop-off" sub-problem.

4.3.2 Genotype Structure

The genotype represents the genetic makeup of the GNP, encoded as a solution to the problem.

Representation:

Encoded Solution: The genotype is an array of 35 genes, each encoding specific information (Fig. 4.5). This array is divided into 8 segments, correlating with the 8 nodes in the phenotype.

1	2	2	3	2	3	4	5	11	3	3	4	5	10
ID	Number Of Connections	con	con	ID	Number Of Connections	con	con	con	ID	Number Of Connections	con	con	con

Figure 4.5: Genotype Structure

Gene Properties:

- **Node ID**: A unique integer representing the identification of each node.

- **Number of Connections**: Defines how many connections the node has.
- **Connection**: Represents the node to which this node is connected.

For example, "1" is J1, "2" is J2... "6" is P1..., Judgment node 1 (J1) is connected to judgment node 2 (J2) and judgment node 3 (J3). Therefore, the number of connections for J1 is 2, and it is connected to node 2 and node 3.

Implementation

For the implementation, the GNP uses the genotype as the initial chromosome; everything is encoded in a one-dimensional array, and each gene is an integer. In this structure, both the Node ID and the number of connections are fixed; however, their connections are formed randomly. Each connection corresponds to a Node ID, represented by a randomly generated number within the range of the total number of nodes.

For the pick-up task, the graph comprises 8 nodes. Each connection in this graph is represented by a randomly generated integer ranging from 1 to 8. The overall population size for a sub-problem is set at 50.

4.3.3 Function Library:

It is a key-value pair data structure that stores judgment/processing functions. The key stores the Node ID, while the value stores the name of the judgment/processing functions. When the program reaches a specific node, the corresponding function is called based on the function library. Each function within this library carries distinct responsibilities:

Processing nodes and functions:

Processing node 1
input:current environment state
Description: move the taxi to the South
action: taxi move to the South
Execution time:1

Processing node 2
input:current environment state
Description: move the taxi to the North

action: taxi move to the North
Execution time:1

Processing node3
input:current environment state
Description: move the taxi to the East
action: taxi move to the East
Execution time:1

Processing node 4
input: current environment state
Description: move the taxi to the west
action: taxi move to the West
Execution time:1

Processing node 5
input: current environment state
Description: pick-up the passenger
action: taxi pick-up the passenger
Execution time:1

Processing node 6
input: current environment state
Description: drop-off the passenger
action: taxi drop-off the passenger
Execution time:1

Judgement nodes and functions :

Judgement node 2
input: current environment state
Description: drop-off branch
Decision:

```
IF passenger is at the same position as taxi
    THEN perform Go P6 (Drop off the passenger)
ELSE IF passenger is to the East of the taxi
    THEN perform Go J5 (move East branch)
ELSE IF passenger is to the West of the taxi
    THEN perform Go J4 (move West branch)
ELSE IF passenger is to the South of the taxi
    THEN perform Go P1 (move the taxi to the South)
ELSE IF passenger is to the North of the taxi
    THEN perform Go P2 (move the taxi to the North)
```

Execution time: 5

Execution time: 5

Judgement node 3
input:current environment state
Description: Pick up branch
Decision:

```
IF passenger is at the same position as taxi
    THEN perform Go P5 (Pick up the passenger)
ELSE IF passenger is to the East of the taxi
    THEN perform Go J5 (move East branch)
ELSE IF passenger is to the West of the taxi
    THEN perform Go J4 (move West branch)
ELSE IF passenger is to the South of the taxi
    THEN perform Go P1 (move the taxi to the South)
```

```
ELSE IF passenger is to the North of the taxi
    THEN perform Go P2 (move the taxi to the North)
```

Execution time:5

Judgement node 4
input:current environment state
Description:move West branch
Decision:

```
IF there is a wall on the West of the taxi
    THEN
        IF the row of the taxi's position is smaller than 2
            THEN perform Go P1 (move the taxi to the South)
        ELSE IF the row of the taxi's position is greater than 2
            THEN perform Go P2 (move the taxi to the North)
ELSE
    perform Go P4 (move the taxi to the West)
```

Execution time:5

Judgement node 5
input:current environment state
Description:move East branch
Decision:

```
IF there is a wall on the East of the taxi
    THEN
        IF the row of the taxi's position is smaller than 2
            THEN perform Go P1 (move the taxi to the South)
        ELSE IF the row of the taxi's position is greater than 2
            THEN perform Go P2 (move the taxi to the North)
ELSE
    perform Go P3 (move the taxi to the East)
```

Execution time:5

4.3.4 Summary

This section provides a comprehensive overview of the genotype and phenotype structures for both sub-problems. It is crucial for understanding the decision-making process in the GNP system. The intricate relationship between genotype and phenotype underpins the adaptability and efficiency of the network in navigating complex tasks.

4.4 Fitness function

4.4.1 Introduction

In this section, we will provide a detailed explanation of our innovative fitness algorithm. We propose a novel approach that allows the 'black box' solution to evolve during the execution of individual structures. This differentiates our method from traditional fitness functions, as the 'black box' solution guides the evolution in the correct direction. Furthermore, we have integrated the fitness equation with the A* algorithm to address the limitations of the traditional fitness equations. The section is organized into two main parts: the first part presents a literature review of fitness functions, while the second part delves into the specifics of our fitness function. This includes the execution of the individual and the fitness equation.

4.4.2 Related Work

A fitness function is an objective function used to summarize, in a single figure of merit, how close a given design solution is to meeting the set objectives. Fitness functions are used in evolutionary algorithms (EA), such as GA, GP, and GNP, to guide simulations to optimal design solutions [50]. In these EA algorithms, the core process involves testing these solutions, eliminating the least effective ones, and creating new ones from the best. The effectiveness of each solution is determined by a fitness function, which assesses how well a solution meets the specified criteria.

The fitness function is all about multi-objective optimization in GNP. Two primary approaches dominate this field: the weighted sum approach and Pareto optimization. Each method presents unique strategies and implications, particularly relevant in the context of GNP fitness functions [51].

Weighted Sum and Penalty Functions

Weighted Sum: The weighted sum method integrates multiple objectives into a single objective function by assigning a specific weight to each objective. These weights reflect the relative importance of each objective and are used in a process of normalization [52]. These weights are used to calculate a single fitness value for each solution. Here is how it works:

- **Normalization:** The value of each objective is normalized, meaning it is brought to a common scale (e.g., by using costs or degrees of fulfillment).
- **Fitness Calculation:** You calculate a 'raw fitness' score for each solution by summing the weighted values of all objectives.
- **Formula:** If you have objectives $o_1, o_2, \ldots, o_O$ and weights $w_1, w_2, \ldots, w_O$, the raw fitness is $f_{\text{raw}} = \sum_{i=1}^{O} o_i \cdot w_i$, ensuring that the sum of weights equals 1.

Penalty Functions: If there are constraints that solutions must respect, penalty functions can be used. These functions penalize the fitness of solutions that violate constraints.

- **Penalty Application:** The raw fitness score is multiplied by the penalty function(s). If there are no violations, the penalty is 1 (no effect); if there are violations, it reduces the fitness score.

Pareto Optimization

Pareto-Optimal Solutions: Within GNP, a solution is Pareto-optimal if no other solution can improve one objective without worsening another [52]. The collection of all such solutions is the Pareto set.

- **Pareto Front:** This is a graphical representation of the Pareto set. In a two-objective problem, this would be a curve or line on a graph where each point is a Pareto-optimal solution.

Constraints: Just like in the weighted sum approach, solutions that meet all constraints are considered better.

Usage in Evolutionary Algorithms (EAs): Evolutionary Algorithms (EAs), such as SPEA2 and NSGA-II, are highly effective for approximating the Pareto front in multi-objective optimization within GNP [52]. These algorithms excel in creating a

varied set of solutions. This variety offers a wide perspective, which is beneficial for making optimal decisions.

Advantages and Disadvantages

Weighted Sum with Penalty: It is simple and combines any number of objectives and restrictions. This method is particularly useful in scenarios where the objectives are well-understood and can be quantifiably balanced against each other. In the context of GNP, this approach aids in simplifying the optimization process by providing a clear directive towards a single, aggregated objective.

However, it requires predefined weights and might miss certain solutions (especially in non-convex problems). Improvements in one objective might offset shortcomings in another, which can be problematic in complex network scenarios where such trade-offs are not desirable.

Pareto Optimization: This offers a comprehensive set of equivalent alternatives without needing predefined weights. However, visualizing and handling the solutions become difficult with more than three or four objectives.

Comparison

Overall, the weighted sum method is good for convex problems, and Pareto optimization is good for non-convex problems. Pareto optimization is preferred when little is known about the potential solutions or when the number of objectives is limited (up to three or four). The weighted sum is better for repeated optimizations of similar tasks or when no post-optimization decision-making is needed.

In summary, the choice between weighted sum and Pareto optimization depends on the specifics of the problem, such as the number of objectives, the nature of the objectives (convex or non-convex), and whether a comprehensive set of solutions is needed or specific trade-offs can be predefined [51].

For the taxi problem, the weighted sum method with a penalty function is an appropriate method for creating a fitness function. To find the optimal solution, the following objectives need to be considered: pick up, drop off, distance, and penalty. It is clear that there are four objectives involved. This is a case where the objectives are well-understood and can be quantifiably balanced against each other. It reduces the complexity of the problem by combining all objectives into a single fitness value. This makes the fitness function easier to compute and interpret. However, the main challenge is setting the weights correctly, as they can significantly affect the optimization process and the quality of the solution.

Fitness functions for different problem

This subsection reviews the diverse landscape of fitness functions, highlighting how they are made to meet the specific demands of different problems.

TileWorld

The TileWorld problem is a popular research problem in GNP study. It requires the agent to find the tile and push the tile into the hole. This is a classic multi-goal path planning problem. The following papers [53][14] explain the details of the fitness function:

$$f = 100 \times DT + 3 \times (ST^{\text{pre}} - ST^{\text{used}}) + 20 \times \sum_{t \in \text{Tile}} (D_t - d_t). \tag{4.1}$$

Overall, this fitness function uses the weighted sum method to build the fitness function. Here, DT measures the count of tiles correctly placed into holes. As the DT is an important objective in the problem domain, the weight for it is 100. The symbol ST^{pre} indicates the total allocated steps, while ST^{used} reflects the steps actually taken. The term Tile denotes the collection of tiles, with D_t and d_t representing the initial and final distances from tile t to its closest hole. The distance is calculated with the Manhattan distance.

This function encapsulates three objectives within the TileWorld framework: (1) maximizing the number of tiles deposited, (2) minimizing the steps employed, and (3) reducing the distance to holes for any remaining tiles when not all can be deposited within the step limit.

For a holistic evaluation across multiple scenarios, the following cumulative fitness metric is employed:

$$F = \sum_{\text{world}=1}^{10} f(\text{world}). \tag{4.2}$$

This aggregate fitness score, F, yields an assessment of an individual's overall proficiency across various test maps, providing a comprehensive evaluation of its performance.

Unmanned Aerial Vehicles

Similar to TileWorld, this paper [34] also uses the weighted sum method to build

the fitness function for the UAV problem.

$$C = \sum_{i=1}^{n} (w_1 l_i^2 + w_2 h_i^2 + w_3 f_{t_i}) \tag{4.3}$$

As the equation above shows, the symbol l refers to path segment lengths, h to UAV altitudes, and f_{t_i} to threat evaluations on the UAV's path. Each of these factors contributes to the multi-objective function that the path planning algorithm aims to optimize, balancing between the shortest and safest route at an optimal altitude.

Robot Path Planning

In robot path planning, particularly in an environment where the robot must avoid circular obstacles, the fitness function is designed to evaluate the quality of a given path. The fitness function is shown below in this paper [37]:

$$f = \sum_{i=1}^{4} \text{path}_i + \sum_{i=0}^{n_{\text{collision}}} C_{\text{penalty}} \cdot \max(0, r_o - d_i) \tag{4.4}$$

Path Length Term: This part of the fitness function sums up the lengths of the sub-paths. The variable path_i represents the length of the ith sub-path in the route. The total path length is important because, generally, shorter paths are preferred in path planning for efficiency.

Collision Penalty Term: This part of the fitness function adds penalties for collisions with obstacles. For each obstacle, if the path comes within the radius r_o of the obstacle, a penalty is applied. The penalty is calculated as $C_{\text{penalty}} \cdot \max(0, r_o - d_i)$, where:

- C_{penalty} is a constant that determines the severity of the penalty for hitting an obstacle.
- r_o is the radius of the obstacle.
- d_i is the distance from the path to the center of the ith obstacle.

If the path does not intersect with an obstacle (i.e., d_i is greater than r_o), the penalty for that obstacle is zero. The *max* function ensures that the penalty is non-negative.

The collision penalty is a critical aspect of the fitness function, as it strongly discourages routes that come too close to obstacles, with a higher penalty applied for

paths that penetrate deeper into the obstacle's area. In a multi-goal path planning problem, it is necessary to use penalty terms to find a collision-free path; in the taxi problem, when the taxi hits an obstacle or makes a wrong pick-up/drop-off, a penalty can be applied.

In conclusion, for a multi-goal path planning problem, there are normally more than 3 objectives involved; therefore, the weighted sum method is more suitable for building the fitness function. The difference between the final distance and the initial distance is an important term, which can be calculated through the Manhattan distance or Euclidean distance. Moreover, a penalty term can be added to avoid obstacles or incorrect actions, such as wrong pick-up/drop-off. Most importantly, different weights need to be assigned for different objectives to balance them against each other. In the taxi problem, the correct pick-up/drop-off is important, thus we need to assign higher weights for these objectives.

4.4.3 Structure of the fitness function

The "pick-up" and "drop-off" sub-problems share the same fitness function and logic. The structure of the fitness is shown in the pseudo-code below (Algorithm 2):

Algorithm 2 Fitness evaluation

1: $current\_environment \leftarrow$ get the initial state of the environment
2: $initial\_distance \leftarrow$ initial distance between taxi and the destination/passenger
3: Execute the current chromosome, letting the agent run on the map
4: $final\_environment \leftarrow$ get the final state of the environment
5: $final\_distance \leftarrow$ final distance between taxi and the destination/passenger
6: $fitnessValue \leftarrow$ *fitness equation*$(initialDistance, finalDistance, otherObjects)$
7: **return** $fitness\_value$

At the beginning of the chromosome evaluation, the fitness function will read the initial state of the environment. This includes the initial position of the taxi and the locations of the passenger or the drop-off point. At this stage, the initial distance between the passenger and the destination (drop-off point) is calculated. This serves as a baseline to measure improvement.

Each chromosome (a set of instructions or path) is then executed, and the taxi will be moved based on the executed processing node. After executing the chromosome, the final state of the environment is recorded. This includes the new position of the taxi.

Once the current chromosome execution is complete, it collects the final state of

the environment and calculates the final distance between the passenger and the destination. The difference between the initial distance and the final distance is one of the objectives in the fitness equation. All of this data will be entered into the fitness equation, and the final fitness value will be returned as the fitness value for the current chromosome.

Algorithm walk through

- Step 1: Suppose the initial taxi position is at the position (2, 2).
- Step 2: After executing the chromosome, the state of the taxi map will be updated.
- Step 3: The final taxi position is at the position (0, 0)
- Step 4: The distance between the initial and final taxi positions will be calculated.
- Step 5: The distance and other objects will be fed to the fitness equation to calculate the fitness value for the current chromosome.

Distance calculation

In the previous section, we learned that the initial distance is normally calculated with the Manhattan distance [54] and Euclidean distance [54]

The Manhattan distance between two points is calculated using the following equation:

$$\text{Manhattan Distance} = |x_2 - x_1| + |y_2 - y_1| \tag{4.5}$$

where (x_1, y_1) and (x_2, y_2) are the Cartesian coordinates of the two points. The symbol $|\cdot|$ denotes the absolute value, ensuring the distance is always a non-negative number.

Steps to Calculate Manhattan Distance:

1. **Identify Coordinates:** Determine the coordinates of the two points. Assume Point 1 has coordinates (x_1, y_1) and Point 2 has coordinates (x_2, y_2).
2. **Calculate Differences in Each Axis:**
   - Calculate the difference in the x-axis: $x_2 - x_1$.
   - Calculate the difference in the y-axis: $y_2 - y_1$.

3. **Take Absolute Values:** Since distance cannot be negative, take the absolute value of each difference.
4. **Sum the Absolute Differences:** Add the absolute values of the differences together.

However, there are some limitations for using Manhattan distance and Euclidean distance, for example:

Consider Point A at coordinates $(1, 1)$ and Point B at coordinates $(2, 4)$.

- Difference in x-axis: $2 - 1 = 1$
- Difference in y-axis: $4 - 1 = 3$

Therefore, the Manhattan distance is:

$$|1| + |3| = 1 + 3 = 4 \tag{4.6}$$

Here is another example, the Manhattan distance between Point A and Point B is 4. If we have another point X at coordinates $(1, 5)$. The Manhattan distance from Point A to Point B is:

- Difference in x-axis: $1 - 1 = 0$
- Difference in y-axis: $5 - 1 = 4$

Therefore, the Manhattan distance is:

$$|0| + |4| = 0 + 4 = 4 \tag{4.7}$$

Consider the following scenario, where Point A is the destination: This is where the taxi needs to go. Point B (Initial State of the Taxi): This is where the taxi starts. Point X (Final State of the Taxi after executing the chromosome): This is where the taxi ends up after following a set of instructions. The goal for the taxi is to move closer to Point A, the destination.

In GNP, a fitness function is used to evaluate how 'good' a solution is (in this case, the route taken by the taxi). The fitness function in this scenario considers the

difference in Manhattan distance from the taxi's initial state (Point B) to its final state (Point X) relative to the destination (Point A). A greater difference in this distance implies a better fitness value. Essentially, if the taxi gets closer to Point A, the fitness value increases.

In this case, the taxi makes some movements (from Point B to Point X), but the Manhattan distance from the taxi's location to the destination (Point A) does not change. This means that, despite moving, the taxi is not getting any closer to the destination. As a result, the fitness value does not improve, which is problematic for the evaluation.

In GNP, evolution happens when individuals with better fitness values are selected for creating the next generation. If the fitness value does not increase despite the taxi's movement, it indicates that the algorithm is not effectively guiding the taxi closer to the destination. This lack of improvement in fitness values can lead to a situation where the population (all the potential routes the taxi could take) does not evolve. In other words, the algorithm might keep suggesting routes that don't bring the taxi any closer to Point A.

To address the above problem, our fitness function will calculate the actual distance rather than the Manhattan Distance or Euclidean distance. To do this, we use the following function (Algorithm 3) with the A* algorithm to overcome this shortcoming.

This function calculates the actual path distance from the taxi's current position `taxi_row, taxi_col` to a target location. At the beginning, a graph is created to represent the possible paths in a 5x5 grid, which simulates the taxi environment. Nodes are added to the graph, each representing a point in the grid. In the taxi environment, the obstacle is set between two grids vertically, therefore it needs the `cannot_move_right` and `cannot_move_left` lists to define points in the grid where right or left movements are not possible, simulating obstacles in movement. Edges are added between nodes to represent possible movements. The conditions check the `cannot_move_right` and `cannot_move_left` lists to avoid adding edges where movement is restricted. The function then uses the A* algorithm to find the shortest path from the current location (start) of the taxi to the destination. The distance is calculated as the number of steps in the path minus one (because the path includes both the beginning and end points).

Algorithm walk through

Let us use an example to illustrate how this function works. Suppose we have a taxi at position (2, 2) and a passenger at position (0, 0).

- **Graph Construction:** A 5x5 grid graph is constructed with nodes for each grid cell and edges representing valid movements.
- **Adding Movement Restrictions:** Edges are added or not added based on predefined movement restrictions. If there is an obstacle between two grids, the taxi cannot cross it.
- **Pathfinding with A*:** Suppose the shortest path found by the A* algorithm from (2, 2) to (0, 0) is $[(2,2),(1,2),(0,2),(0,1),(0,0)]$.
- **Calculate Distance:** The distance is 4 since the path includes 5 nodes, and the number of moves is one less than the number of nodes in the path.

Algorithm 3 A* Pathfinding for distance calculation

```
1: function ASTAR(taxi_row, taxi_col, location)
2:     rows ← 5
3:     cols ← 5
4:     Initialize graph G
5:     for y ∈ {0, ..., rows − 1} do
6:         for x ∈ {0, ..., cols − 1} do
7:             Add node (y, x) to G
8:         end for
9:     end for
10:    cannot_move_right ← {obstacle, location ...}
11:    cannot_move_left ← {obstacle, location ...}
12:    for y ∈ {0, ..., rows − 1} do
13:        for x ∈ {0, ..., cols − 1} do
14:            if (y, x) not in cannot_move_right and x + 1 < cols then
15:                Add edge (y, x) to (y, x + 1) in G
16:            end if
17:            if (y, x) not in cannot_move_left and x − 1 ≥ 0 then
18:                Add edge (y, x) to (y, x − 1) in G
19:            end if
20:            if y + 1 < rows then
21:                Add edge (y, x) to (y + 1, x) in G
22:            end if
23:            if y − 1 ≥ 0 then
24:                Add edge (y, x) to (y − 1, x) in G
25:            end if
26:        end for
27:    end for
28:    start ← (taxi_row, taxi_col)
29:    goal ← location
30:    path ← use A* to find path from start to goal in G
31:    distance ← length of path − 1
32:    return distance
33: end function
```

4.4.4 Running Individual Structure

In the fitness evaluation phase, the individual needs to be executed for fitness evaluation. Different from other methods, we propose a new algorithm that enhances the fitness evaluation phase by using a 'black box' solution to guide the evaluation process.

To better understand the problem, we need to recap the structure of the chromosome. The chromosome consists of a graph and a function library:

Chromosome Representation

A chromosome is represented as a one-dimensional array. Each element in the array contains:

- A node ID.
- The number of connections to that node.
- The IDs of connected nodes.

Function Library

Each node ID corresponds to a function in a predefined function library. When a node ID is encountered during execution, the corresponding function is called.

'Black box' solution representation

In the previous section, we created a handcrafted solution as the 'black box' solution. The 'black box' solution only receives an input and returns an output. To guide the evolution, we convert the handcrafted solution into a function. This function plays a critical role:

- **Input**: It accepts the current state of the environment.
- **Output**: It returns a list of possible executed nodes (Node ID) for the agent.

The function is named `Get_Infor_From_Solution`, which is a combination of the judgment functions and processing functions. It operates primarily through a cascade of "if-else" statements, each corresponding to a specific condition of the agent's state.

Decision-Making Process

- **Condition Assessment**: It checks the state against a set of predefined conditions.
- **Action Determination**: If a condition is satisfied, the function identifies the appropriate actions that the agent can take.
- **List Compilation**: The identified actions are then appended into a list. This list represents the optimal actions that the agent can perform from its current state.

Example Scenario

As we can see from the image below(Fig.4.6), consider a simulation with the following elements:

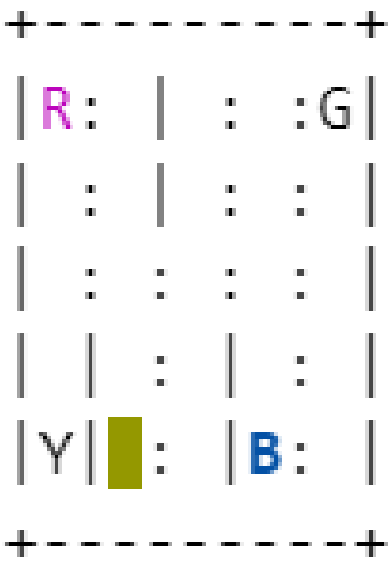


Figure 4.6: Taxi Map

- **Taxi (Agent)**: Represented by a yellow rectangle.
- **Drop-off Destination**: Indicated by a red "R" located at the grid's top-left corner.
- **Relative Positioning**: The destination is to the northwest of the taxi's current position.

Given this scenario, the "black box" suggests possible actions based on the taxi's relative position to the drop-off destination:

- **North Movement**: The taxi can move North to approach the destination vertically(processing node P1, node ID: 6).
- **West Movement**: The taxi can move West to close the distance horizontally(processing node P3, node ID: 9).
- **Drop-off Action**: If the taxi is at the destination, it can perform the drop-off. (processing node P6, node ID: 11)

For each possible action, the corresponding node ID from the simulation's logic is added to the list of optimal actions. Therefore, the output of the "black box" is [6,9,11]. Assuming that we have a black box system (deep neural network) that returns the probability of distribution for all possible actions that can be taken, we can simply pick the top 3 actions based on the ranking of probabilities. (e.g. Move North [0.8], Move West [0.6], Drop off [0.5], Move East [0.2], Move South [0.24], Pick-up [0.1].

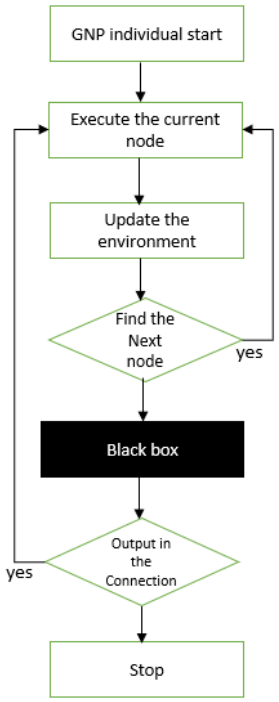


Figure 4.7: Runing the individual

Execution Flow

The above flow chart (Fig.4.7) illustrates the procedure of running an individual in our context. Within the given individual, the first node will be executed, and the corresponding function will be called. The function will return a node ID, which is the next node to be executed. If the returned node ID is in the current node's connections, that node will be executed next. Otherwise, the program will go through the outputs from the "black box". If the program finds an output (node ID) that can also be found in the connections, that node will be executed next. Otherwise, the program is stopped.

The pseudo-code (Algorithm 4) shows the details of the execution flow.

- **Start Execution**: The algorithm starts at the first index of the chromosome array.
- **Function Call**: It calls the function associated with the node ID at this index.
- **Next Node ID**: The function returns a next node ID, indicating the subsequent action or node to execute. This is shown on line 9 of the pseudo-code.

Finding Connections

Current Node Connections: The algorithm determines the connections for the current node based on the chromosome structure:

- The Node ID is at the position of "index" in the chromosome.
- The number of connections is at index + 1.
- The actual connections start at index + 2 and extend for the number of connections specified.

Next Node Check: It checks if the returned next node ID is among the current node's connections.

Connection Found: On lines 11 to 12, the algorithm will check if the next node ID to be executed is in the current node's connections. If it finds the node ID in the connection list, it will call the `FIND_NODE_INDEX` function to find the index of this node ID in the chromosome.

Utilizing the 'black box' Solution

Connection Not Found: If it cannot find the next node ID to be executed in the connection list, we will use the 'black box' solution. The function `Get_Infor_From_Solution`

will be called, and the 'black box' solution will provide a list of possible solutions for the current state of the environment.

- It then checks if the current node's connections contain any node IDs that are in the list of possible solutions.
- If a match is found, `FIND_NODE_INDEX` is used to find the corresponding index.

Breaking the Loop

If the algorithm cannot find a connection that aligns with the 'black box' solution, it breaks the loop. The evaluation for this chromosome is stopped. Otherwise, it will increment the time and keep evaluating this chromosome. This prevents the algorithm from pursuing paths that do not contribute to achieving the desired outcome.

The 'black box' solution serves as a guide, directing the evolutionary process more efficiently. In the traditional method, the evaluation of the current chromosome halts when it fails to find the next node to execute from the connections. However, our method allows the evaluation process to continue. The 'black box' solution offers more accurate information to enhance exploration. By comparing the node IDs in the current connections with those in the potential solution list, the algorithm ensures that evolution is moving towards the optimal solution.

Algorithm 4 Graph Traversal Function

```
1: function GO_THRU(graph_array)
2:     path ← empty list
3:     env ← INITIAL_ENVIRONMENT
4:     index ← 0
5:     time ← 1
6:     while time ≤ timeFrame do
7:         node_id ← graph_array[index]
8:         Append node_id to path
9:         nextNode ← FUNCTIONS_LIBRARY[NODE_ID](env)
10:        current_node_con ← graph_array from (index + 2) to (index + 2 +
   graph_array[index + 1])
11:        if nextNode ∈ current_node_con then
12:            index ← FIND_NODE_INDEX(nextNode, graph_array)
13:        else
14:            solution_connections ← GET_INFOR_FROM_SOLUTION(env)
15:            found_connection ← None
16:            for each connection in con do
17:                if connection ∈ solution_connections then
18:                    found_connection ← connection
19:                    index ← FIND_NODE_INDEX(found_connection, graph_array)
20:                    break
21:                end if
22:            end for
23:            if found_connection is None then
24:                break
25:            end if
26:        end if
27:        time ← time + 1
28:    end while
29:    return path, env
30: end function
```

Algorithm walk through

Given the setup with chromosome and the `functions_library`, the `go_thru` function navigates through a graph based on the output of these functions and environmental information. Here is a walkthrough:

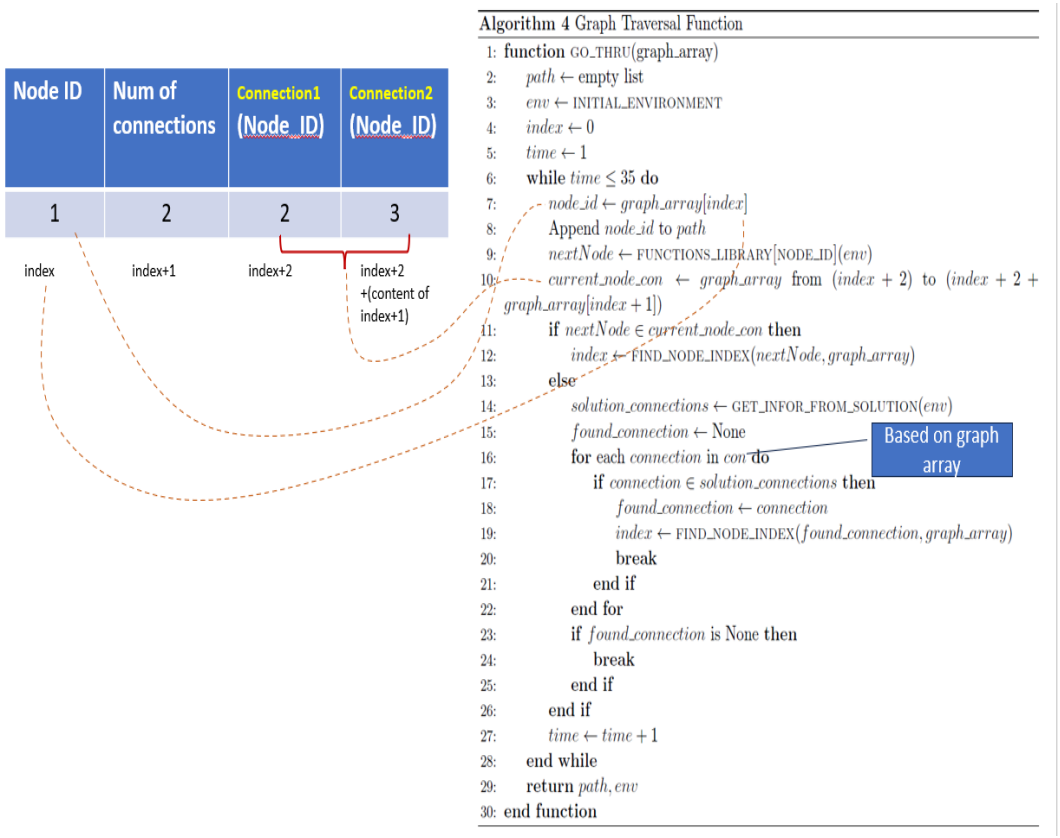
1. Start at **node 1**, execute `J1()`, which returns **2**. Since **2** is a direct connection of node 1, we move to **node 2**.
2. At **node 2**, execute `J2()`, which returns **4**. We move to **node 4** as it is a direct connection.
3. At **node 4**, execute `J4()`, which returns **6**. We move to **node 6** as it is a direct connection.
4. Suppose at some point, a node's function returns a value not directly connected. The function then checks `get_infor_from_Solution()` for any match. If `get_infor_from_Solution()` returns $[2, 3, 5]$, and one of the connections is within this list, it moves to that node. Otherwise, the traversal stops.
5. Eventually, this function will return an updated environment and a path, which is the result of traversing the chromosome, [1,2,4,6,1] for this case, since node 6 p0 is a processing node, the taxi will be moved, the environment will be updated.

```
def J1(): return 2
def J2(): return 4
def J3(): return 5
def J4(): return 6
def J5(): return 1
def P0(): return 1

functions_library = {
    1: J1, 2: J2, 3: J3, 4: J4, 5: J5, 6: P0,
}

graph_array = [1, 2, 2, 3, 2, 3, 6,
1, 4, 3, 4, 2, 6, 4, 5, 4, 2, 5, 6, 5, 1, 2, 6, 1, 1]

# Assuming get_infor_from_env() returns [2, 3, 5]
```



Algorithm 4 Graph Traversal Function

```
function GO_THRU(graph_array)
    path ← empty list
    env ← INITIAL_ENVIRONMENT
    index ← 0
    time ← 1
    while time ≤ 35 do
        node_id ← graph_array[index]
        Append node_id to path
        nextNode ← FUNCTIONS_LIBRARY[NODE_ID](env)
        current_node_con ← graph_array from (index + 2) to (index + 2 + graph_array[index + 1])
        if nextNode ∈ current_node_con then
            index ← FIND_NODE_INDEX(nextNode, graph_array)
        else
            solution_connections ← GET_INFOR_FROM_SOLUTION(env)
            found_connection ← None
            for each connection in con do
                if connection ∈ solution_connections then
                    found_connection ← connection
                    index ← FIND_NODE_INDEX(found_connection, graph_array)
                    break
                end if
            end for
            if found_connection is None then
                break
            end if
        end if
        time ← time + 1
    end while
    return path, env
end function
```

Figure 4.8: Runing the individual with black box

4.4.5 Fitness equation

The fitness equation is designed to evaluate the performance of tasks such as taxi pick-up and drop-off scenarios. It uses the Weighted Sum Method with additional penalty terms to calculate a fitness score, $f(i)$ for each solution i.

$$f(i) = W_1 \cdot P + W_2 \cdot (ST - SU) + W_3 \cdot \max(D_{ti} - D_{tf}, 0)$$

$$f(ii) = W_1 \cdot D + W_2 \cdot (ST - SU) + W_3 \cdot \max(D_{pi} - D_{pf}, 0)$$

where:

- W_1, W_2, W_3, W_4, are the weights associated with each term of the fitness function,
- P is 1 if the passenger was picked, 0 otherwise,

- ST is the given time frame for a task,
- SU is the actual time used,
- D_{ti} is the initial distance from taxi to passenger,
- D_{tf} is the final distance from taxi to passenger,
- D_{pi} is the initial distance from passenger to dropoff,
- D_{pf} is the final distance from passenger to dropoff,
- D is 1 if the passenger was dropped off successfully, 0 otherwise.

Overall the fitness equation is made up of 4 objects, each with its purpose and associated weight:

- **Successful Task Completion:** This is represented by binary indicators—P for pick-up and D for drop-off—where "1" indicates success and "0" indicates failure.
- **Time Efficiency:** The term $(ST - SU)$ assesses time efficiency, with ST as the allocated time frame and SU as the actual time used. Solutions that complete the task faster than allotted are preferred.
- **Distance Reduction:** For pick-up, $\max(D_{ti} - D_{tf}, 0)$ measures the reduction in distance from the taxi to the passenger. For drop-off, $\max(D_{pi} - D_{pf}, 0)$ applies similarly to the passenger-drop-off distance, The max function ensures that the fitness score is not negative.
- **Penalty for Inefficiency:** This object records the penalties for wrong Pick up/Drop off, this ensures the solution is efficient.

Weights assignment

The terms of the fitness equation are each multiplied by a weight that signifies the relative importance of each objective. In the equation, the successful pick-up and drop-off are the most important factors, and they are set to a value that significantly influences the fitness score: W_1 is 100.

While the primary goal is to ensure passengers are picked up and dropped off successfully, other factors like efficiency in terms of distance and time are also important but to a lesser degree. The ability to minimize the distance traveled is also very

important. The weight W_3 reflects this and is assigned the second-highest value of 50. Similarly, W_2 is 30.

To discourage solutions that are redundant and do not reduce distances, a penalty is introduced. This is the least significant objective but still influences efficiency. This weight W_4 is set to 20.

4.5 Parent Selection

In our GNP, we use tournament selection as the parent selection method. The basic idea is to run a "tournament" among a few randomly selected individuals from the population and then select the best out of these to be a parent. The individuals with relatively high fitness scores will be selected.

This method is selected for the following reasons: First, it is computationally efficient, especially for large populations, because it only evaluates a small subset of the population at each step. Second, it allows for the control of selection pressure. A larger tournament size increases the selection pressure because there is a higher chance that the best individuals will be selected. Conversely, a smaller tournament size gives weaker individuals a higher chance to be selected, which maintains diversity in the population. Third, tournament selection is straightforward to implement and does not require global information about the population, like fitness ranking or scaling.

As shown from the pseudo-code (Algorithm 5) below, the process is as follows:

- **Tournament Size**: Decide on a tournament size k. This is the number of individuals that will be randomly selected to participate in each tournament. Common sizes are 2 or 3, but they can be larger.
- **Random Selection**: Randomly select k individuals from the population without replacement.
- **Fitness Evaluation**: Evaluate the fitness of each of the k individuals.
- **Selection**: The individual with the highest fitness is declared the winner of the tournament and is selected to be a parent in the new generation.
- **Repetition**: Repeat this process as many times as needed to create a pool of parents for breeding.

Algorithm 5 Tournament Selection

```
1: procedure TournamentSelection(population, tournamentSize, numberOfSelections)
2:     selections ← new list()
3:     for i ← 1, numberOfSelections do
4:         tournament ← new list()
5:         for j ← 1, tournamentSize do
6:             individual ← RandomlySelectFrom(population)
7:             AddToList(tournament, individual)
8:         end for
9:         winner ← FindHighestFitness(tournament)
10:        AddToList(selections, winner)
11:    end for
12:    return selections
13: end procedure
```

4.6 Evolution Operators

4.6.1 Crossover

Once all the elites are selected from the parent selection phase, they will be sent to the crossover phase. We use uniform crossover. Here is a simplified explanation of the crossover process in our GNP:

- **Selection of Parents**: Two parent networks are randomly selected from the elites.
- **Crossover Point Selection**: For each gene position in the offspring, the algorithm decides randomly (with equal probability) whether to inherit the gene from the first parent or the second parent. In our case, the crossover swaps the connections (for the given index of chromosome i, the items from $i + 2$ to $i + 2 +$ number of connections). The possibility of swapping this connection is 0.6, which is the crossover rate.
- **Recombination**: The selected connections from the parents are exchanged or recombined to create new individuals. As shown in the picture, two chromosomes A and B are selected, and they swap their connections to generate two new chromosomes.

4.6.2 Mutation

To maintain the diversity of the individuals among the population, the chromosomes should be mutated after the crossover. Here, we use our customized mutation method for both sub-problems.

Evolution is limited to node connections, allowing compact programs to be generated and evolved efficiently using a directed graph with a relatively small size [14]. In our mutation, the node mutation only happens in the connections. The connection is swapped within the range of node IDs with a 0.1% mutation rate. The pseudo-code (Algorithm 6) below shows the details of the mutation:

1. **Initialization:** The function initiates by creating an empty list `new_chromosome` to store the mutated graph structure. The properties of the chromosome are found. For each branch, the function reads its `nodeID`, the number of connections (`num_connections`), and the actual connections (`a_connections`).
2. **Mutation Process:** Each connection in `a_connections` is examined to decide whether it should be mutated, based on a mutation probability of 0.1%. If a mutation is to occur, the specific connection is randomly altered to a new connection ID within a predefined range of [1, 8]. Since the range of Node IDs is 1 to 8, this ensures the mutation will not introduce other nodes that could affect the evolution in the wrong direction.
3. **Building the Mutated List:** The node's ID, its number of connections, and the potentially mutated connections are appended to the list `new_chromosome`.
4. **Return:** The function returns `new_chromosome` when all these steps are finished.

Algorithm 6 Mutate Node Connections

```
1: procedure MutateConnections(a)
2:     i ← 0
3:     new_a ← empty list
4:     while i < length of a do
5:         nodeID ← a[i]
6:         num_connections ← a[i + 1]
7:         a_connections ← slice of a from i + 2 to i + 2 + num_connections
8:         modified_connections ← empty list
9:         for current_connection in a_connections do
10:            if random probability ≤ 0.1 then          ▷ Mutation rate of 0.1
11:                mutated_connection ← random integer from 1 to 8
12:            else
13:                mutated_connection ← current_connection
14:            end if
15:            Append mutated_connection to modified_connections
16:        end for
17:        Append nodeID to new_chromosome
18:        Append num_connections to new_chromosome
19:        Extend new_chromosome with modified_connections
20:        i ← i + 2 + num_connections
21:    end while
22:    return new_chromosome
23: end procedure
```

4.6.3 Conclusion

In this section, we explained the implementation of the proposed GNP variant. To solve the taxi problem efficiently, we use a "divide-and-conquer" approach to divide it into 2 sub-problems. Based on the design of the hand-crafted solution, the total number of nodes is reduced from 13 to 8 nodes for both sub-problems. In the evaluation phase, the fitness function uses A* to calculate the actual distance rather than the Manhattan distance or Euclidean distance. Additionally, the fitness function uses the input and output of a "black box" system to guide the evolution. Moreover, this GNP uses customized evolution operators to further accelerate the evolution process.

Chapter 5

Black Box System to GNP Conversion

5.1 Introduction

In this section, we are going to explain our proposed methodology in detail. Our methodology aims to refine the GNP solution to achieve a perfect solution. It is structured into three key steps:

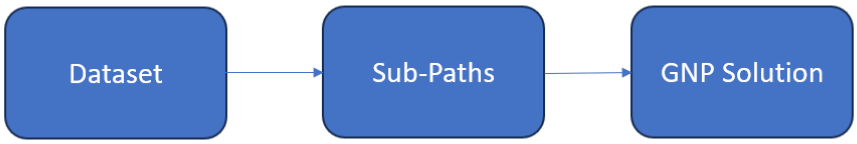


Figure 5.1: GNP solution Refinement process.

- **Generate the Dataset**: The dataset will be created based on the handcrafted solution.
- **Derive Unique Sub-Paths**: From the dataset, we will extract unique sub-paths that emerge across different solutions.
- **Construct the Perfect GNP Solution**: Utilizing these unique sub-paths, we

will build an enhanced GNP solution.

5.2 Dataset Generation

As mentioned in the previous sections, the dataset will be created using the 'black box' system, which, in our case, is the handcrafted solution. The handcrafted solution will serve as a benchmark for the GNP solution dataset, allowing us to discover the regular patterns within the GNP solution dataset and identify unique sub-paths.

The dataset is created by executing the computer graph in numerous environments, capturing the actual paths taken. The current state within the environment will be used as the input for the handcrafted solution. The judgment nodes within the handcrafted solution will help invoke the best action via a series of judgment and processing node calls (we call them 'paths'; the small chunks within the 'paths' are 'sub-paths'). As shown in the example image, nodes J1, J3, J4, P2, etc., will be called, and the corresponding functions within the function library will be executed. Their node IDs (1, 3, 4, 2, etc.) will be added to the dataset. Thus, the dataset is composed of executed node IDs.

By combining these paths from 400 distinct environments, we compile comprehensive datasets for analysis. The handcrafted solution dataset will serve as a benchmark for the GNP solution dataset, allowing us to discover the regular patterns within the GNP solution dataset and identify unique sub-paths.

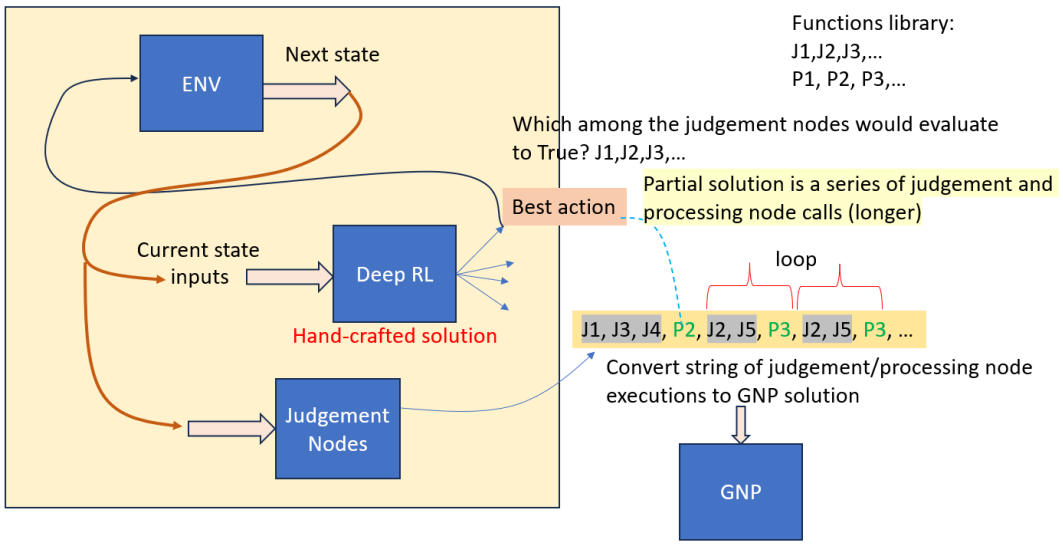


Figure 5.2: Dataset Generation process.

Presumably, there is no handcrafted solution, and we only have a "black box" which can only return the best action as output. How can we generate the dataset? To generate the dataset in a more general way, this approach can be utilized:

- **Input**: Sense the environment.
- **Run the "black box"**: Run the "black box" until the MAX STEPs are reached.
- **Get the response from the Judgment nodes**: Using our set of judgment nodes, for each of the judgment nodes:
  - Feed the input and check which judgment node returns true (e.g., J1, J3, J4).
  - Add to the "path".
- **Get the action taken by the black box**: E.g., P8 (move East).

With the given 'black box' system, which only returns the best action for different inputs, the input here is the current state of the environment. For the problem, some judgment nodes with judgment functions will be predefined. These judgment nodes will only return 'true' or 'false.' When a specific judgment node returns 'true,' the ID of this judgment node will be added to the dataset.

The image (Fig.5.3) below shows the partial snapshot of the generated dataset. Each line in the dataset is a complete path for an environment.

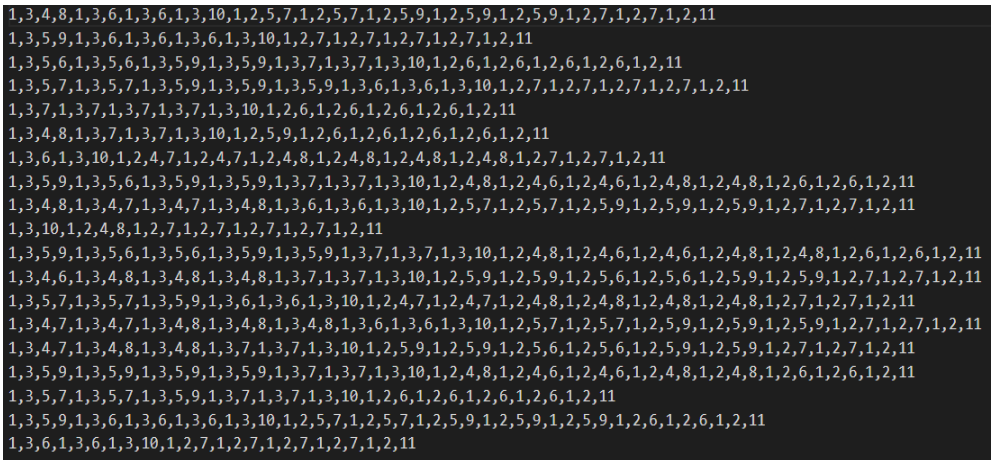


Figure 5.3: Dataset sample.

5.2.1 Hand-crafted solution dataset

A single handcrafted solution will be executed across 400 different Taxi environments. The pseudo-code (Algorithm 7) below illustrates the process of the handcrafted dataset generation.

- **Initialization**: The code starts with the initialization of the starting node and an empty dataset.
- **Loop Over Simulations**: The for loop iterates from 1 to 400, inclusive. Each iteration represents a separate simulation run with a unique seed, corresponding to the 400 unique Taxi environments.
- **Graph Execution**: A while loop continues as long as the total time frame is less than 66 (we will explain why the maximum steps are 66 in a later section). This condition ensures that the dataset generation process is performed within the given time frame (or actions) for each simulation, and each executed node is recorded and appended to the dataset.

Once the for loop completes, the algorithm will generate a dataset through multiple simulations of the Taxi-v3 environment, with each simulation contributing a fixed number of paths to the dataset.

Algorithm 7 Handcraft Dataset Generation

```
start ← "1"
dataset ← []
total ← 0
current ← "1"
for i ← 1 to 400 do
    env ← gym.make("Taxi − v3", render_mode =′ ansi′).env
    (state, info) ← env.reset(seed = i)
    while total < 66 do
        current_function ← functionsLibrary[current]
        returned_value ← current_function()
        total ← total + 1
        dataset.append(current)
        current ← returned_value
    end while
end for
```

5.2.2 Pattern Identification

With the dataset at our disposal, the next step involves analyzing the generated handcrafted solution dataset. We can identify the following patterns:

- **Path Composition**: The path is composed of multiple sub-paths, and each sub-path always starts from Judgment node 1 (the integer following the colon represents the Node ID) and ends with one of the processing nodes. This is because Judgment Node 1 is programmed to activate after the execution of a processing node.
- **Sub-path Length**: The lengths of the sub-paths are consistent, comprising either three or four nodes.
- **Reusability and Recombination**: These sub-paths appear to be both repeatable and reusable, suggesting that all solutions could potentially be assembled from various combinations of these sub-paths.

Here is an example of a single path in the handcrafted solution dataset. The path is composed of multiple sub-paths: [1, 3, 4, 8], [1, 3, 4, 7], [1, 3, 4, 8], [1, 3, 6]... Each sub-path always starts from Judgment node 1 and ends with a processing node. The length of the sub-path is either 3 or 4. The sub-path [1, 3, 4, 8] appears twice here, indicating it is reusable.

5.3 Sub-path Extraction

The pattern of the sub-path within the dataset has been identified in the last section. The path is composed of some unique sub-paths, and we need to extract all these sub-paths within the dataset. To meet the requirements of the pattern, these two functions work together to find the unique sub-sequences of specific lengths from the datasets.

5.3.1 Function `find_subsequences_of_length`

This function takes two parameters: `data`, a list of integers, and `length`, an integer specifying the length of sub-sequences to find. It extracts these sub-sequences from `data`, this ensures the length of the sub-path is 3 or 4 and it should always start from Judgement node 1, then it returns them as a list of tuples.

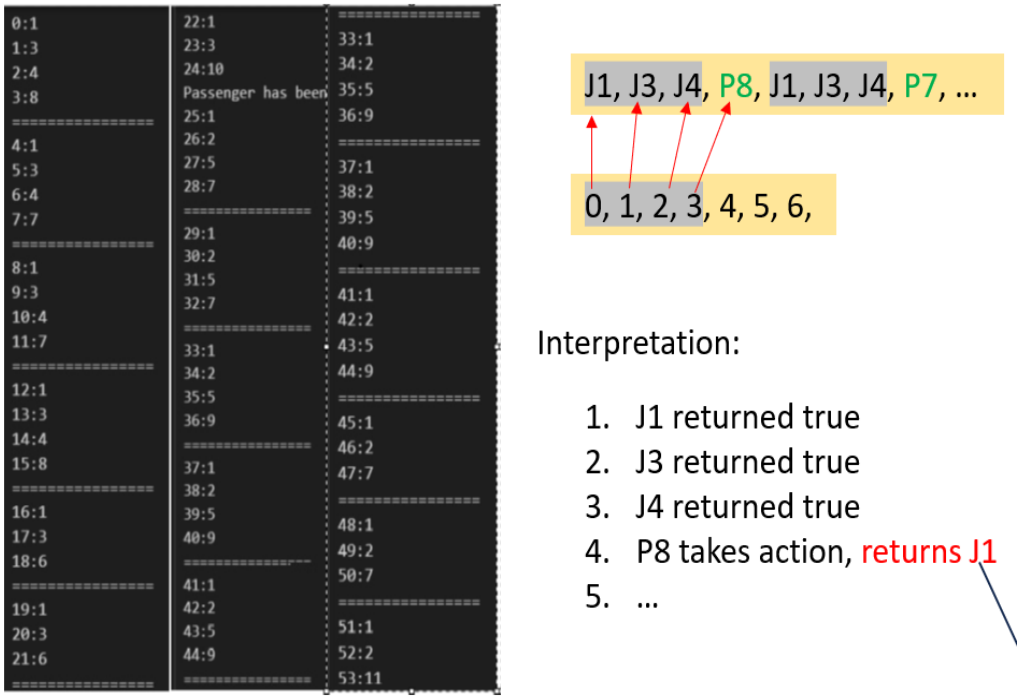


Figure 5.4: Example of a single path in the handcraft solution dataset

1. **Initialization of `subsequences` list:**

   ```
   subsequences = []
   ```

   This list stores the sub-sequences that match our desired conditions.

2. **Iterating over the `data` list:**

   ```
   for i in range(len(data)):
   ```

   The function iterates through each index `i` in the `data` list to examine potential starting points for sub-sequences.

3. **Checking for a specific start condition:**

   ```
   if data[i] == 1:
   ```

This condition requires that a sub-sequence must start with the integer `1`. If this condition is met, the function then extracts a sub-sequence of the specified length starting from this index.

4. **Extracting and validating sub-sequences:**

```
subsequence = data[i:i+length]
if len(subsequence) == length and subsequence[-1] != 1:
```

A sub-sequence is extracted starting from index `i` up to `i+length`. The function then checks two conditions:

- The extracted sub-sequence must be exactly `length` elements long.
- The last element of the sub-sequence must not be `1`.

5. **Storing valid sub-sequences:**

```
subsequences.append(tuple(subsequence))
```

If a sub-sequence meets the conditions, it is converted to a tuple for hashability, which helps identify unique sub-sequences later on. and it will be added to the `subsequences` list.

6. **Returning the list of sub-sequences:** The function returns the `subsequences` list, containing all sub-sequences that meet the requirement.

5.3.2 Function `findUniqueSubsequences`

This function is designed to find unique sub-sequences of specified lengths from the dataset.

1. **Defining desired sub-sequence lengths:**

```
lengths = [3, 4]
```

This specifies the lengths of sub-sequences, in this case, because the length of the sub-path is either 3 or 4.

2. **Initialization of `unique_subarrays` set:**

```
unique_subarrays = set()
```

A set is used to retain unique sub-sequences while eliminating duplicates.

3. **Iterating over the dataset:**

```
for data in big_dataset:
```

The function iterates over each path within the collection of `big_dataset`.

4. **Finding sub-sequences for each specified length:** Within the dataset iteration, another loop iterates over the desired lengths:

```
for length in lengths:
    subsequences = find_subsequences_of_length(data, length)
    unique_subarrays.update(subsequences)
```

For each `data` list and each specified `length`, the function `find_subsequences_of_length` is called to find all sub-sequences that meet the criteria.

5. **Returning the unique sub-sequences as lists:** Once the function has verified all of these conditions, the sub-sequences will be added to a set. After converting each unique sub-sequence to a list, the algorithm returns the result.

Up to here, we have a list of sub-arrays, however, there are still some duplicate sub-arrays in the list. For example, sub-path [1,2,7] and sub-path [1,2,7,8] are in the list, but sub-path [1,2,7] is a sub-sequence of [1,2,7,8], the sub-path [1,2,7] should not be in the list. Therefore, we need to filter the list further.

This function, `find_unique_subarrays`, is designed to filter out arrays from a list of arrays that are sub-sequences of longer arrays in that list. It ensures that unique arrays (in terms of not being a sub-sequence of any other array) are kept. Let's go through the code step by step:

1. **Sorting the arrays based on their lengths in descending order:**

```
arrays.sort(key=len, reverse=True)
```

The `sort()` function is used to sort the `arrays` list in place. The `key` parameter is set to the built-in `len` function, which sorts arrays based on length. The `reverse=True` argument indicates that the sort is done in descending order, so the longest arrays come first. This is a necessary step for efficiently checking sub-sequences, as longer arrays cannot be sub-sequences of shorter ones.

2. **Initialization of an empty list `unique_arrays`:**

```
unique_arrays = []
```

This list will store the arrays that are determined to be unique, i.e., not a subsequence of any longer array in the original list.

3. **Iterating over the sorted list of arrays:**

```
for i in range(len(arrays)):
```

This loop goes through each array in the sorted `arrays` list by index.

4. **Checking if an array is a subsequence of any longer array:**

Inside the first loop, a variable `is_subsequence` is initialized to `False` for each array. Then, another loop iterates over the arrays again:

```
for j in range(len(arrays)):
    if i != j and len(arrays[i]) < len(arrays[j]):
        if arrays[i] == arrays[j][:len(arrays[i])]:
            is_subsequence = True
            break
```

- The condition `if i != j` ensures that an array is not compared with itself.

- The condition `len(arrays[i]) < len(arrays[j])` ensures that only longer arrays are considered for comparison to avoid unnecessary checks and logically, a longer array cannot be a sub-sequence of a shorter or equal-length array.
- The innermost `if` statement checks if the current array (`arrays[i]`) is a sub-sequence of the longer array (`arrays[j]`). This is done by comparing the current array with the slice of the longer array from the beginning to the length of the current array. If they are equal, it means `arrays[i]` is a sub-sequence of `arrays[j]`, and `is_subsequence` is set to `True`.

5. **Adding unique arrays to the result list:**

   After checking against all other arrays, if `is_subsequence` remains `False`, it means the current array is not a sub-sequence of any longer array, and it is added to the `unique_arrays` list:

```
if not is_subsequence:
    unique_arrays.append(arrays[i])
```

6. **Returning the list of unique arrays:**

   Finally, the function returns the `unique_arrays` list, which contains all arrays that were not sub-sequences of any longer arrays in the original list.

The main goal of this function is to ensure that the returned list of arrays does not include any array that can be found as a starting segment (sub-sequence from the beginning) of any longer array in the original list. The code checks only if an array is identical to the start of a longer array, rather than checking if it is a sub-sequence that might occur anywhere within the longer array. This is because sub-path [1,2,7] is a sub-sequence of sub-path [1,2,4,7]; however, both of them are unique, so they should be retained.

After the sub-path extraction, we get a list of unique sub-paths. These sub-paths can be combined in any order to form a complete path for any Taxi environment. The image below shows the extracted sub-paths. There are a total of 18 sub-paths, and each sub-path matches the required pattern.

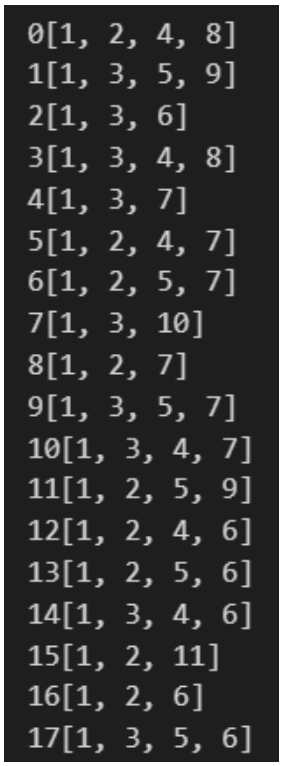


Figure 5.5: Unique sub-paths

5.4 GNP solution construction

After the sub-path extraction, we obtain a list of unique sub-paths. We need to construct the GNP solution using these sub-paths. The process involves three steps:

Initialization: A default dictionary of sets is used to initialize a graph representation. This approach avoids duplicate keys, allows for the automatic generation of keys, and ensures that each edge is only stored once.

Path Processing: Each path is processed sequentially. An edge from the first node to the second node in each sub-path is inferred and added to the graph for every pair of consecutive nodes. Based on traversal information, edges are added to the graph during this step.

Graph Finalization: Not every node may have been captured in the initial processing, especially nodes with no outgoing edges in the provided paths. Based on observations from the handcrafted solution, processing nodes always connect to judgment node 1. Therefore, some connections will be manually added to ensure completeness.

Algorithm 8 Reconstruct GNP solution from Paths

```
Graph ← empty defaultdict with set as default value
Paths ← list of paths
                                        ▷ Reconstruct the graph from paths
for each path in Paths do
    for i = 0 to length of path − 2 do
        Graph[path[i]].add(path[i + 1])
    end for
end for
                                        ▷ Convert sets to sorted lists for consistency
for each node in Graph do
    Graph[node] ← sorted list of Graph[node]
end for
                                        ▷ Manually add connections for Processing Node
for node =Processing node do
    Graph[node] ← [1]
end for
```

As illustrated by the pseudo-code(Algorithm 8) above, the following steps will explain how to construct GNP solutions:

1. Initializing the Graph

```
reconstructed_graph = defaultdict(set)
```

Here, the `reconstructed_graph` is initialized as a `defaultdict` where each value is set to be a `set`. This structure is ideal for graphs, in which each edge is unique. `defaultdict` is a subclass of the built-in `dict` class. If a key is not found in the dictionary, it returns a default value. This default value is specified when the `defaultdict` is instantiated with the set. This means if you try to access or modify a key that does not exist in the dictionary, `defaultdict` automatically creates the key with the value of an empty set, this will avoid a `KeyError`.

2. List of Paths

```
paths = [...]
```

This section provides a list of paths. Each path is a list of integers that represents a sequence of node IDs in the graph.

3. Reconstructing the Graph

To reconstruct the graph, the code iterates over each path to connect each node to its subsequent node.

```
for path in paths:
    for i in range(len(path) - 1):
        reconstructed_graph[path[i]].add(path[i + 1])
```

For every pair of nodes $(path[i], path[i+1])$, the code adds an edge from $path[i]$ to $path[i+1]$ by adding $path[i+1]$ to the set of edges for $path[i]$ in `reconstructed_graph`.

4. Sorting the Edges

```
reconstructed_graph = {k: sorted(v) for k, v in reconstructed_graph.items()}
```

Once all edges are added, this line iterates over the `reconstructed_graph` dictionary to convert each set of edges into a sorted list. This ensures that the output is deterministic (the order of elements in a set is not guaranteed, but a list is ordered).

Here is an example:

$$\text{paths} = [[1, 2, 4], [1, 3, 5]]$$

For each sub-path, they will be processed as follows:

1. For the first sub-path $[1, 2, 4]$:
   - Add an edge from 1 to 2. The graph now indicates an edge from node 1 to node 2.
   - Add an edge from 2 to 4. This adds an edge from node 2 to node 4.

   After processing this path, the graph has edges: $1 \to 2$ and $2 \to 4$.
2. For the second path $[1, 3, 5]$:
   - Add an edge from 1 to 3. Node 1 now has edges to both 2 and 3.

- Add an edge from 3 to 5. This indicates an edge from node 3 to node 5.

After processing this sub-path, the graph includes additional edges: $1 \rightarrow 3$ and $3 \rightarrow 5$.

This is the constructed graph:

$$
\begin{aligned}
&1 : \{2, 3\} \quad \text{// Node 1 connects to nodes 2 and 3} \\
&2 : \{4\} \quad \text{// Node 2 connects to node 4} \\
&3 : \{5\} \quad \text{// Node 3 connects to node 5}
\end{aligned}
$$

After converting sets to sorted lists, the final graph is:

$$
\begin{aligned}
&1 : [2, 3] \\
&2 : [4] \\
&3 : [5]
\end{aligned}
$$

5. Manually Adding Connections

```
for node in range(6, 12):
    reconstructed_graph[node] = [1]
```

This explicitly sets nodes 6 through 11 to have a single edge pointing back to node 1. Nodes 6 through 11 are processing nodes, and they are all connected to node 1, as stated in the earlier sections.

After graph construction, it generates the complete graph, which can be visualized in the figure (Fig.5.7). There are a total of 11 nodes categorized into three types: start nodes, judgment nodes, and processing nodes. Node 1 always serves as the start node. Nodes 1 to 5 (J1 to J5) correspond to judgment nodes, and nodes 6 to 11 (P6 to P11) represent processing nodes. Note: there is no terminal node in the individual.

```
1: [2, 3]
2: [4, 5, 6, 7, 11]
4: [6, 7, 8]
3: [4, 5, 6, 7, 10]
5: [6, 7, 9]
6: [1]
7: [1]
8: [1]
9: [1]
10: [1]
11: [1]
```

Figure 5.6: Final constructed graph.

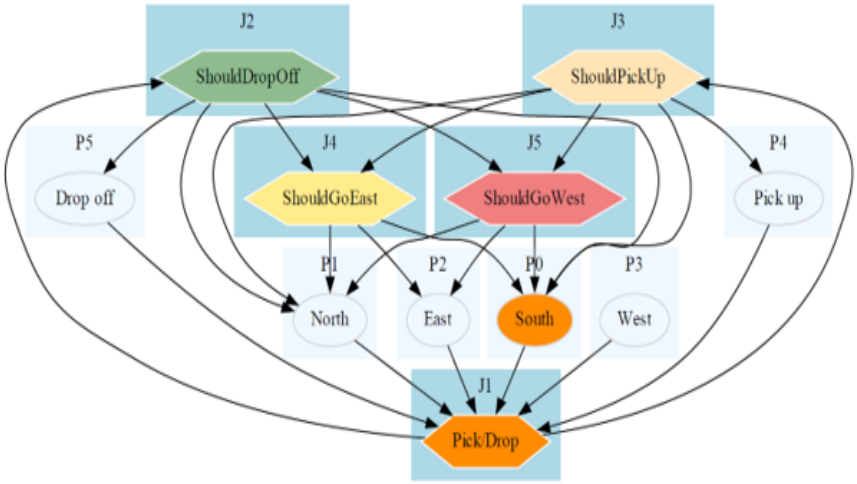


Figure 5.7: Graph visualization.

Chapter 6

Experiments and Analysis of Results

6.1 Hand-crafting a GNP Solution

6.1.1 Motivation

As the ground truth of the Top-Bottom approach, the handcrafted solution is very important. Validating the effectiveness of a handcrafted solution in research involves testing its generalizability across various scenarios. Our objective is to assess the solution's adaptability in addressing the Taxi problem under diverse conditions. We aim to determine whether the handcrafted approach can consistently solve the problem across all these varied cases. To achieve this, we plan to evaluate the solution in 500 different Taxi environments.

6.1.2 Experiment Design

- Initialize the Taxi-V3 environment in a for-loop. For each iteration i, it corresponds to the seed for one kind of environment.
- Implement the handcrafted solution using the function library within each iteration of the Taxi environment.
- Evaluate whether the handcrafted solution successfully picks up and drops off the

passenger in each environment.

Target: The taxi can pick up and drop off the passenger successfully in all 500 Taxi environments.

6.1.3 Result

The table (Fig. 6.1) above shows the test result for the handcrafted solution. The result indicates that 66 steps are sufficient for the solution to handle all 500 scenarios without fail. This is the reason we set the time frame to 66 in the previous section. This result demonstrates the generalizability of the handcrafted solution, which can solve the taxi problem for all 500 scenarios.

MAX_TIME (steps)	Score	
42	178/500	36%
50	328/500	66%
60	454/500	91%
65	490/500	98%
66	500/500	100%
70	500/500	100%
80	500/500	100%

Figure 6.1: Hand-crafted solution test result

6.2 Using a Black Box System as Fitness Function in a GNP

6.2.1 Learning Ability of the Proposed GNP variant

Motivation

Our proposed GNP variant is designed based on a handcrafted solution. Our goal is to enable our proposed GNP variant to effectively solve the multi-goal path-planning issue in dynamic environments. To test its performance, we aim to challenge it by exposing it to 500 distinct Taxi-V3 environments and assessing how well it can pick up and drop off passengers.

Experiment Design

For the GNP test, during the training phase, we will test it on 500 Taxi environments (seed = 0 to 499).

Running one GNP code on one environment to solve the pick-up problem takes around 9 seconds. Running GNP to solve both pick-up and drop-off problems in 500 different environments takes a considerable amount of time (around 9 * 500 * 2 = 9000 seconds). Furthermore, executing all test cases in a single loop is impractical. If any implementation fails in any case, debugging becomes challenging. Therefore, we need to devise a test plan to address these issues.

Parameter settings

The above table (Fig. 6.2) shows the parameter settings for our GNP variant:

Population Size	Generation Number	Parent Matching	Mutation Rate	Crossover Rate
100	2000	20	0.001	0.06

Figure 6.2: GNP parameter settings

Test plan

In the evaluation process, the GNP will solve one sub-problem for each of the 500 environments within separate files. We will record the fitness score for the optimal solution and whether the GNP successfully solved each sub-problem.

To efficiently manage this process, we employ a "Generator" and an "Executor."

The "Generator" is tasked with creating 500 different files. It reads a template code, modifies only the seed for each environment, and generates the 500 files accordingly.

The "Executor" handles the execution of these generated files. It processes 20 files simultaneously, executing them in parallel batches. The runtime for each batch of 20 files is approximately 30 seconds.

Target:

The problem-solving rate for the proposed GNP should be higher than 90 per cent during the testing phase.

Results

The table above (Fig.6.3) shows the test result of our proposed GNP variant. This

Training	Training phase
Pick-up	91%
Drop-off	89.40%

Figure 6.3: Proposed GNP variant test result

table shows the success rate for pick-up sub-problems, with a 91 percent success rate during the training phase. However, the success rates for drop-off sub-problems showed an 89.4 percent success rate in the training phase. In this experiment, all 500 cases were used up in the training phase, but they all succeeded.

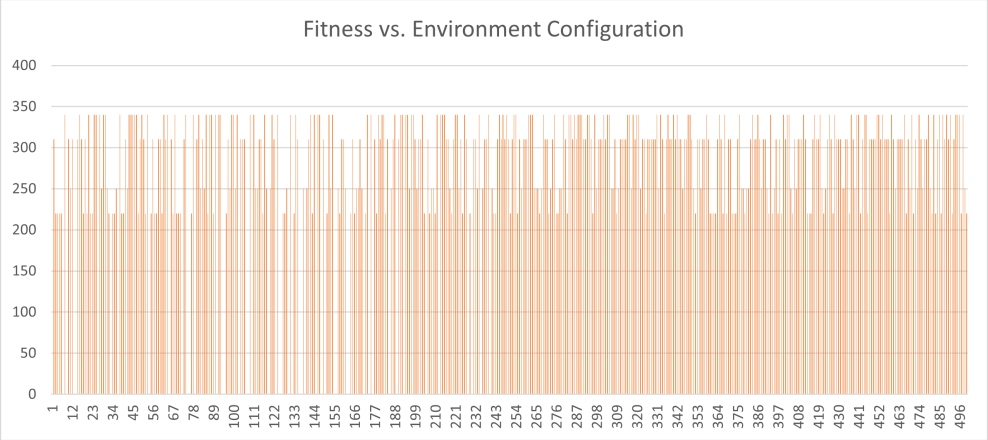


Figure 6.4: Average Population Fitness vs. Environment Configuration

The algorithm succeeded in picking up and then dropping off the passenger in 451 out of 500 environment configurations = 90.2%. Fig.6.4 shows the average fitness value of the population of GNP individuals in the different environment configurations.

Figures (6.5–6.16) show some graphs of how the fitness values of individuals improve over 2000 generations.

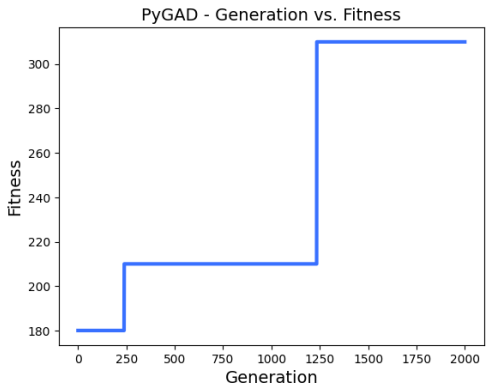


Figure 6.5: Environment 1: pick-up

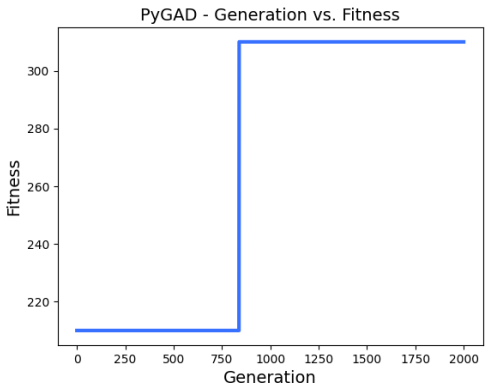


Figure 6.6: Environment 1: drop off

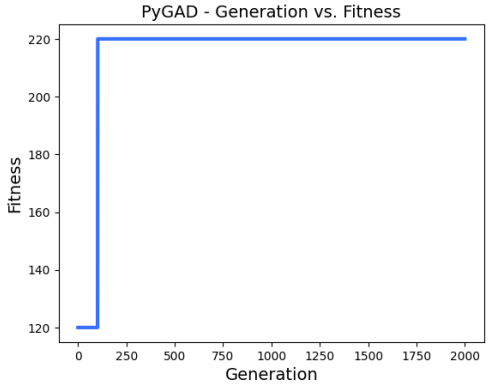


Figure 6.7: Environment 2: pick-up

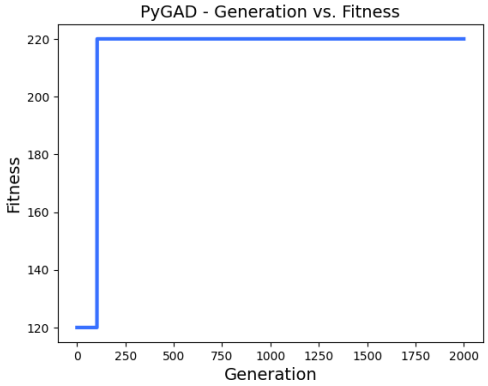


Figure 6.8: Environment 2: drop off

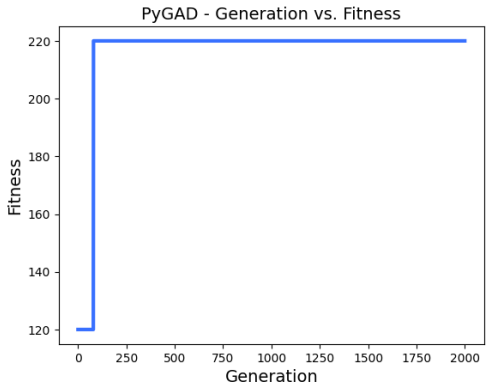


Figure 6.9: Environment 3: pick-up

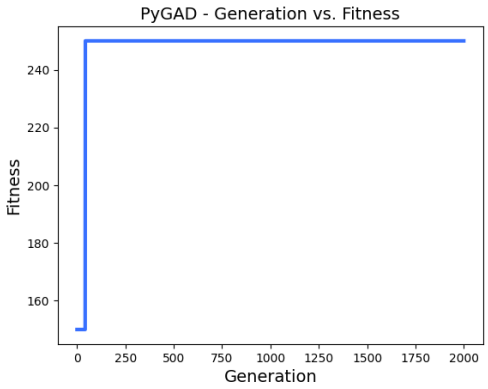


Figure 6.10: Environment 3: drop off

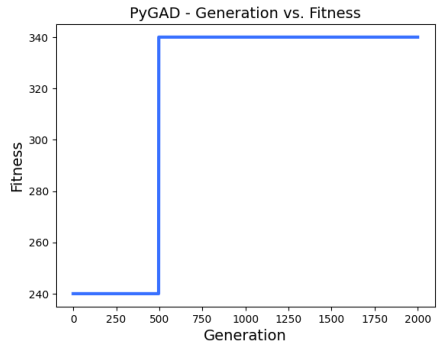


Figure 6.11: Environment 4: pick-up

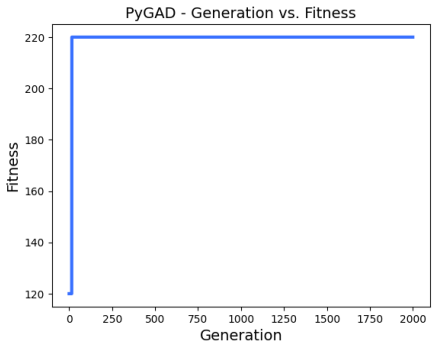


Figure 6.12: Environment 4: drop off

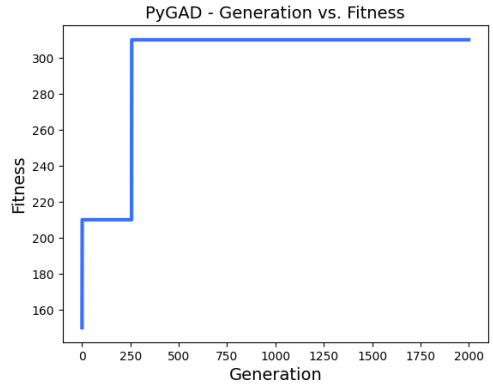


Figure 6.13: Environment 5: pick-up

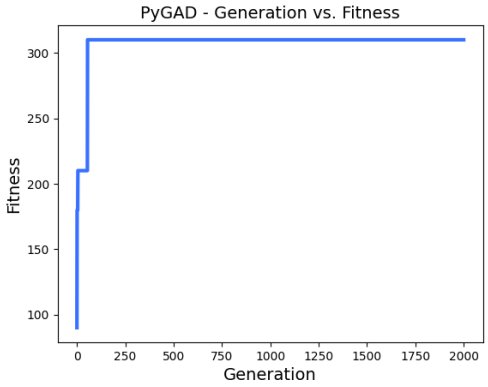


Figure 6.14: Environment 5: drop off

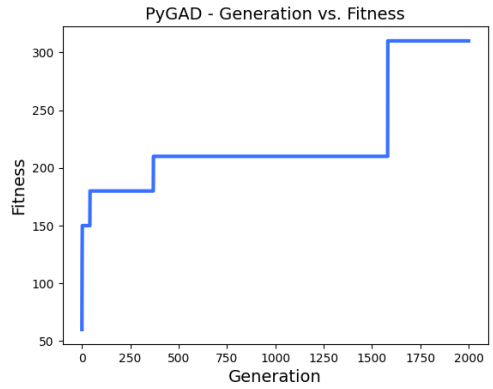


Figure 6.15: Environment 6: pick-up

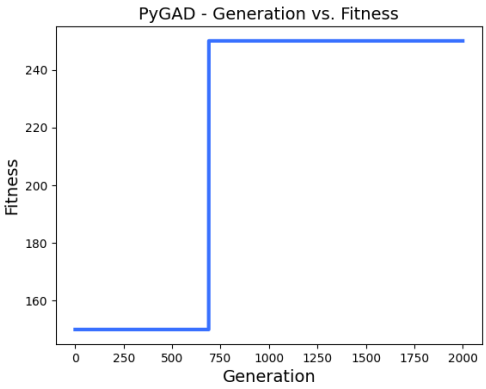


Figure 6.16: Environment 6: drop off

6.3 Experiment on Proposed GNP vs Original GNP

6.3.1 Motivation

To test the performance of our proposed GNP variant, it is necessary to compare it with the original GNP. What happens if we solve the taxi problem with both GNP variants? Which kind of GNP will have better performance?

6.3.2 Experiment Design

- Run the original GNP to solve the taxi problem against 100 different environments.
- Run the proposed GNP to solve the taxi problem against 100 different environments.
- Compare their average fitness score and success rate.

6.3.3 Target:

The proposed GNP aims to achieve better performance than the original GNP.

6.3.4 Training Results:

The tables (Fig.6.17 and 6.18) below show the performance comparison between the original GNP and the proposed GNP for both sub-problems. Overall, the proposed GNP demonstrates better performance in terms of average fitness score and success rate. In 2000 generations, the success rates for the pick-up sub-problem are 52 percent for the original GNP and 91 percent for the proposed GNP, respectively. The proposed GNP achieves a higher average fitness score. For the drop-off sub-problem, the success rates are 56 percent and 89 percent for the original and proposed GNP, respectively, with average fitness scores of 195.3 and 316.2.

	Original GNP	Proposed GNP
Average Fitness	210	316.2
Pick-up Success Rate	52%	91%

Figure 6.17: Original GNP vs Proposed GNP for pick-up sub-problem

	Original GNP	Proposed GNP
Average Fitness	195.3	316.2
Drop-off Success Rate	56%	89%

Figure 6.18: Original GNP vs Proposed GNP for drop-off sub-problem

Figure 6.19: Sample Original GNP vs Proposed GNP graph solution for drop-off sub-problem (Environment 11)

This image (Fig.6.19) shows the graph solutions for both the proposed GNP and the original GNP. Both solutions have a similar structure for the drop-off sub-problem in environment 11; the red "G" is the drop-off location. As we can see, both GNP solutions have some redundant connections; however, they can still solve some instances of the taxi problem. On the other hand, both GNP solutions can generate different paths; one starts by moving North, and the other one starts by moving West.

6.4 GNP solution refinement

6.4.1 Motivation

It is easy to observe that the refined GNP solution is the same as our handcrafted solution. However, we still need to verify that the extracted sub-paths are correct, ensure these sub-paths are unique, and confirm they can be combined to generate any solution.

6.4.2 Experiment Design

- Traverse the handcrafted solution to obtain all possible sub-paths.
- Compare the extracted sub-paths with the sub-paths obtained from traversing the handcrafted solution.
- Given any complete optimal path from any environment, verify that this optimal path is composed of those extracted sub-paths.

Target:

- The extracted sub-paths should match the sub-paths obtained from traversing the handcrafted solution.
- The complete optimal path should be composed of the extracted sub-paths.

The function(Algorithm 9) finds a specific number of unique paths of a given length in a graph, starting from a specific node, it implements a Depth-first search with a set to find unique paths, and the function will be broken when reaches to a certain number of paths.

The first code (Algorithm 10) searches sub-paths within the given complete path and records the index of each found sub-path. The second code (Algorithm 11) extracts elements from the extracted sub-path list based on the positions specified in the indices and then concatenates these elements into a single list (result). After this, we will compare the given complete path and the result to check if they are identical.

Here is an example:

- Result after traversing the hand-crafted solution: [[1, 2, 4],[2, 4, 5], [1, 3, 1]].

Algorithm 9 Find Unique Paths in a Graph

```
procedure TraverseGraph(Graph, StartNode, Lengths, pathNumber)
    Paths ← empty set
    Stack ← deque with (StartNode, [StartNode])
    while Stack is not empty do
        (Node, Path) ← Stack.pop()
        if length(Path) = Lengths + 1 then
            Paths.add(Path as tuple)
            if length(Paths) = pathNumber then
                break
            end if
        else
            for Neighbor in Graph[Node] do
                Stack.append((Neighbor, Path + [Neighbor]))
            end for
        end if
    end while
    return [list(Path) for Path in Paths]
end procedure
```

Algorithm 10 Find Positions of sub-paths in a complete path

```
procedure FindParhPositions(LongArray, ChunkArray)
    ChunkPositions ← empty list
    for i ← 0 to length(LongArray) − 1 do
        for j ← 0 to length(ChunkArray) − 1 do
            if slice(LongArray, i, i + length(ChunkArray[j])) = ChunkArray[j]
then
                Append j to ChunkPositions
                break
            end if
        end for
    end for
    return ChunkPositions
end procedure
```

Algorithm 11 Extract and Concatenate Sub-arrays

```
procedure ExtractSubarrays(Arr, Indices)
    Result ← empty list
    for each Index in Indices do
        if 0 ≤ Index < length(Arr) then
            Result.extend(Arr[Index])
        end if
    end for
    return Result
end procedure
```

- Long Array: [1, 2, 4, 1, 3, 1], Imagine this is a complete path.
- Chunk Array: [[1, 2, 4], [1, 3, 1]], This is the output from our graph traversal(sub-paths).

(Algorithm 10) will return the position of these two sub-paths within the result after traversing the handcrafted solution, which is [0, 3]. (Algorithm 11) will extract sub-paths from the chunk array based on the positions found and extend it to a new array A, which is [1, 2, 4, 1, 3, 1]. Up to this point, we compare the new array A with the Long Array (a complete path). If they are identical, we can claim that the proposed method is working properly.

6.4.3 Results

Overall, the extracted sub-paths are the same as the sub-paths from the traversed handcrafted solution, and the complete optimal path is composed of the extracted sub-paths.

6.5 Experiment on Finding the Minimum Training Patterns

6.5.1 Motivation

We have successfully extracted a perfect GNP solution using 400 training patterns from a black box system. Now, if we reduce the number of training patterns to 350, what will the output be? Can we still achieve a perfect GNP solution? Similarly, what if we further reduce the training patterns to 200? Our goal is to determine the minimum number of training patterns required to achieve a perfect solution.

6.5.2 Experiment Design

1. Reduce the number of training patterns by 100 each time and generate the dataset.
2. Reconstruct the GNP solution with the current dataset.
3. Verify if the constructed GNP solution matches the target GNP solution.

4. Keep reducing the number of training patterns until we can no longer get the perfect GNP solution.

6.5.3 Observations

When the training patterns (also referred to as training environments in this work) are reduced up to a certain level, we cannot generate the perfect GNP solution.

6.5.4 Results

The table (Fig.6.20) below shows the test result. The minimum number of training patterns required to find the perfect solution is 9. The result shows that the proposed methodology only needs a small dataset to generate the perfect GNP solution.

Training Pattern Number	Perfect GNP Solution
300	Yes
200	Yes
100	Yes
50	Yes
25	Yes
13	Yes
7	No
8	No
9	Yes

Figure 6.20: Proposed Methodology test result

The image (Fig.6.21) below shows the incorrect computer graph. When the training patterns are reduced up to 8, we can not generate the perfect GNP solution. For the Judgement node 4, it should connect to nodes 6, 7, and 8, however, node 6 is missing, it only connects to nodes 7 and 8.

The experiments conducted in this study were performed on a computer equipped with a 12th Gen Intel(R) Core(TM) i5-12400F CPU, operating at 2.50 GHz, and supported by 16 GB of RAM. All the implementation is written in Python.

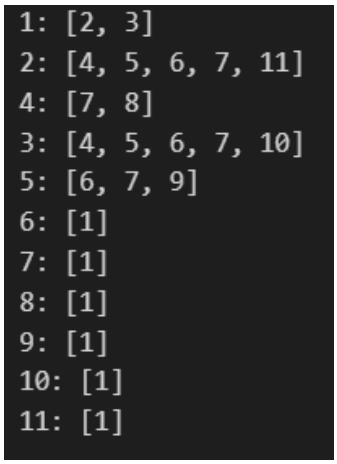


Figure 6.21: Incorrect computer graph

This section provides a summary of key points that were missed in other chapters. This research proposed a methodology that converts the "black box" system into a reusable computer graph capable of solving the taxi problem across all possible configurations (500 different environment settings). The methodology relies solely on the output of the "black box" and predefined judgment nodes for the GNP solution conversion. The "black box" output is the best action for various inputs, where the input is the current state of the environment. We can only access the inputs and outputs of the "black box" system, and nothing more.

In this research, we created a hand-crafted solution to serve as the "black box." This hand-crafted solution is the ground truth of the research, but it could be replaced by any deep learning system as long as they can return the best actions. The process of obtaining the perfect GNP solution involves three steps: dataset generation, sub-path extraction, and GNP graph construction. These approaches are general enough to be applicable to any deep learning system.

From the experiments, we observed that the proposed GNP significantly improved the efficiency compared to the original GNP. In the fitness function of the proposed GNP, the "black box" provides a list of possible actions for the chromosome, increasing the likelihood that the current executing node will find the correct next node in its connections. This enhancement allows the proposed GNP to find the optimal solution with fewer generations

6.6 Limitation of this research

Using a priori knowledge about the problem domain, we have formulated a set of judgment and processing nodes. The hypothesis is that they suffice to solve the problem, but the way they should be utilized in a graph is unknown. Therefore, the links and any reusable nodes must be automatically determined by the algorithm, and the target output is a complete computational graph.

As mentioned in the previous chapter, there are two possible approaches to generating the dataset for GNP solution refinement. The algorithm generates 500 unique GNP solutions, but they were not condensed into one ultimate solution. The missing step was only addressed by the Black Box to GNP conversion algorithm.

.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This research has three main contributions:

First: This work embarks on addressing the "black box" system interpretability problem by utilizing a "perfect solution" (target model) from a "black box" system and then transforming it into a computational graph that is elegant, highly interpretable, and reusable.

To the best of the author's knowledge, no existing research has converted a black box into a GNP graph solution. The chromosome structure of the GNP is networked. The GNP does not "bloat" as the problem complexity increases; instead, it incorporates more interpretable function nodes and combinations of nodes. It also includes a function library with judgment and processing functions to enhance interpretability. As a result, we have embarked on pioneering work in this area of GNP research with a "top-down" approach and a "divide-and-conquer" strategy to further improve interpretability. As the ground truth of our research, the handcrafted solution has achieved a 100 per cent problem-solving rate. This study utilized only the input and output, not the internal computational details, of a "perfect solution" from a "black box" system (the target model), using a "top-bottom" approach, and then transformed it into a GNP computational graph.

Second: This research proposes a new Genetic Network Programming variant that utilizes a "black box" system to improve evolution efficiency significantly.

The utilization of the "black box" as a fitness function prevents the evolution from being stuck in "convergence to a suboptimal solution." The output of the "black box" guides the evolutionary algorithm in choosing the next appropriate node for connection to a given node, increasing the exploration ability of the GNP. Given an environment configuration or state of the world (e.g., location of taxi, pick-up and drop-off locations, distance to pick-up location, distance from pick-up location to drop-off location), the Black Box system provides a set of candidate actions that can be taken. This Black Box could be any system (e.g., deep reinforcement learning) that could return a probability distribution representing the likelihood of taking certain actions in a given state. While the proposed GNP is very effective in evolving the solutions for the training environments, the method is not suitable for handling unseen environments, prompting us to develop the Black Box to GNP solution conversion and refinement algorithm.

Third: This research proposes a methodology for converting a Black Box system into a GNP solution. Furthermore, the algorithm can refine the GNP solution. The refined GNP solution guarantees that it can arrive at a perfect solution to a problem while reducing the number of computational nodes.

It is crucial to have a perfect solution for a multi-goal pathfinding problem in real life; the navigation should be precise. Our experiments demonstrate that the proposed method succeeds in solving the Taxi problem. The refined GNP solution can solve the taxi problem in 500 different environments (complete dataset), and there are no redundant or unnecessary nodes in the refined GNP solution.

In general, the proposed method offers a completely different approach to building a GNP graph solution by utilizing a perfect model in the form of a Black Box system to guide the formulation of a solution. While the scope of the research was limited only to a single agent with multiple goals, the contributed algorithms serve as the building blocks for developing more advanced interpretable algorithms that can address more challenging problems in the future (e.g., multi-objective, multi-agent adversarial problems with elements of uncertainty).

7.2 Future Work

Four objects can be identified for future work:

7.2.1 Scalability Studies

Test the proposed method by applying it to more complex problems. This would help in understanding the limits of the current method and also test the generalizability of the method. The Black Box to GNP conversion algorithm also lends itself amenable to solving other challenging multi-agent problems, such as the TileWorld problem, which cannot be solved 100 percent using existing GNP algorithms.

7.2.2 Error Handling

Test the "black box" system against errors or unexpected inputs. This involves developing strategies to handle situations where the "black box" system provides incomplete or inaccurate information.

7.2.3 Reduce the Need for a Priori Knowledge

In this work, we utilized a priori knowledge of the problem domain and formulated a set of judgment and processing nodes that can be utilized by the algorithm for building the computational graph solution. We envision that these judgment and processing nodes could also be replaced by machine learning-based models that can be fused together in a graph architecture.

7.2.4 Interpretability and Transparency Enhancements

For future work, visualization tools or methods can be developed to simplify the representation of the computational process, the whole process can be more interpretable and transparent in this way.

Bibliography

[1] F. Xu, H. Uszkoreit, Y. Du, W. Fan, D. Zhao, and J. Zhu, *Explainable AI: A Brief Survey on History, Research Areas, Approaches and Challenges*, 09 2019, pp. 563–574.

[2] M. Lent, W. Fisher, and M. Mancuso, "An explainable artificial intelligence system for small-unit tactical behavior." 01 2004, pp. 900–907.

[3] A. Tickle, R. Andrews, M. Golea, and J. Diederich, "The truth will come to light: directions and challenges in extracting the knowledge embedded within trained artificial neural networks," *IEEE Transactions on Neural Networks*, vol. 9, no. 6, pp. 1057–1068, 1998.

[4] J. Dombi and O. Csiszár, *Explainable Neural Networks Based on Fuzzy Logic and Multi-criteria Decision Tools*, 01 2021.

[5] ——, *Interpretable Neural Networks Based on Continuous-Valued Logic and Multi-criteria Decision Operators*, 04 2021, pp. 147–169.

[6] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 4768–4777.

[7] R. Guidotti, A. Monreale, S. Ruggieri, F. Turini, F. Giannotti, and D. Pedreschi, "A survey of methods for explaining black box models," *ACM Comput. Surv.*, vol. 51, no. 5, aug 2018. [Online]. Available: https://doi.org/10.1145/3236009

[8] V. Chahar, J. Chhabra, and D. Kumar, "Parameter adaptive harmony search algorithm for unimodal and multimodal optimization problems," *Journal of Computational Science*, vol. 5, 01 2013.

[9] V. Chahar and D. Kumar, "An astrophysics-inspired grey wolf algorithm for numerical optimization and its application to engineering design problems," *Advances in Engineering Software (1978)*, vol. 112, 05 2017.

[10] J. H. Holland, *Adaptation in natural and artificial systems.* The MIT Press, 1975.

[11] J. R. Koza, *Genetic Programming On the Programming of Computers by Means of Natural Selection.* The MIT Press, 1992.

[12] B. Li, S. Mabu, and K. Hirasawa, "Genetic network programming with automatic program generation for agent control," *Transaction of the Japanese Society for Evolutionary Computation*, vol. 1, no. 1, pp. 43–53, 2010.

[13] L. W. S. E. Fengming Ye, Shingo Mabu and K. Hirasawa, "Genetic network programming with reconstructed individuals," *SICE Journal of Control, Measurement, and System Integration*, vol. 3, no. 2, pp. 121–129, 2010.

[14] X. Li, H. Yang, and M. Yang, "Revisiting genetic network programming (gnp): Towards the simplified genetic operators," *IEEE Access*, vol. 6, pp. 43 274–43 289, 2018.

[15] M. E. Pollack and M. Ringuette, "Introducing the tileworld: Experimentally evaluating agent architectures," in *Proceedings of the Eighth National Conference on Artificial Intelligence - Volume 1*, ser. AAAI'90. AAAI Press, 1990, p. 183–189.

[16] X. Li, W. He, and K. Hirasawa, "Adaptive genetic network programming," in *2014 IEEE Congress on Evolutionary Computation (CEC)*, 2014, pp. 1808–1815.

[17] S. Eto, S. Mabu, K. Hirasawa, and T. Huruzuki, "Genetic network programming with control nodes," in *2007 IEEE Congress on Evolutionary Computation*, 2007, pp. 1023–1028.

[18] L. Bi and S. Zhang, "Analysis and research models of the estimation of distribution algorithms," in *Proceedings of 2011 International Conference on Computer Science and Network Technology*, vol. 3, 2011, pp. 2014–2018.

[19] X. Li, S. Mabu, H. Zhou, K. Shimada, and K. Hirasawa, "Genetic network programming with estimation of distribution algorithms for class association rule mining in traffic prediction," in *IEEE Congress on Evolutionary Computation*, 2010, pp. 1–8.

[20] X. Li, S. Mabu, and K. Hirasawa, "Towards the maintenance of population diversity: A hybrid probabilistic model building genetic network programming," *Transaction of the Japanese Society for Evolutionary Computation*, pp. 89–101, 09 2011.

[21] X. Li, S. Mabu, B. Li, and K. Hirasawa, "Probabilistic model building genetic network programming using reinforcement learning," *Transaction of the Japanese Society for Evolutionary Computation*, pp. 29–40, 10 2011.

[22] S. Mabu, K. Hirasawa, J. Hu, and J. Murata, "Online learning of genetic network programming," *IEEJ Transactions on Electronics, Information and Systems*, vol. 122, pp. 355–362, 03 2002.

[23] S. Mabu, K. Hirasawa, and J. Hu, "Genetic network programming with reinforcement learning and its performance evaluation," 06 2004, pp. 710–711.

[24] S. G. Park, S. Mabu, and K. Hirasawa, "Robust genetic network programming using sarsa learning for autonomous robots," in *2009 ICCAS-SICE*, 2009, pp. 523–527.

[25] S. Mabu, H. Hatakeyama, K. Hirasawa, and J. Hu, "Genetic network programming with reinforcement learning using sarsa algorithm," in *2006 IEEE International Conference on Evolutionary Computation*, 2006, pp. 463–469.

[26] S. Mabu, K. Hirasawa, and J. Hu, "A graph-based evolutionary algorithm: Genetic network programming (gnp) and its extension using reinforcement learning," *Evolutionary computation*, vol. 15, pp. 369–98, 02 2007.

[27] S. Mabu and K. Hirasawa, "Evolving plural programs by genetic network programming with multi-start nodes," in *2009 IEEE International Conference on Systems, Man and Cybernetics*, 2009, pp. 1382–1387.

[28] L. Yu, J. Zhou, S. Mabu, K. Hirasawa, J. Hu, and S. Markon, "Elevator group control system using genetic network programming with aco considering transitions," in *SICE Annual Conference 2007*, 2007, pp. 1330–1336.

[29] l. yu, J. Zhou, S. Mabu, K. Hirasawa, J. Hu, and S. Markon, "Double-deck elevator group supervisory control system using genetic network programming with ant colony optimization with evaporation." *JACIII*, vol. 11, pp. 1149–1158, 01 2007.

[30] X. Li, G. Yang, and K. Hirasawa, "Evolving directed graphs with artificial bee colony algorithm," in *2014 14th International Conference on Intelligent Systems Design and Applications*, 2014, pp. 89–94.

[31] "Graph structure optimization of genetic network programming with ant colony mechanism in deterministic and stochastic environments," *Swarm and Evolutionary Computation*, vol. 51, p. 100581, 2019.

[32] M. Ganeshmurthy and G. Suresh, "Path planning algorithm for autonomous mobile robot in dynamic environment," in *2015 3rd International Conference on Signal Processing, Communication and Networking (ICSCN)*, 2015, pp. 1–6.

[33] R. Nemani, N. Cherukuri, G. R. K. Rao, P. V. V. S. Srinivas, J. J. Pujari, and C. Prasad, "Algorithms and optimization techniques for solving tsp," in *2021 Fifth International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, 2021, pp. 809–814.

[34] S.-Y. Fu, L.-W. Han, Y. Tian, and G.-S. Yang, "Path planning for unmanned aerial vehicle based on genetic algorithm," in *2012 IEEE 11th International Conference on Cognitive Informatics and Cognitive Computing*, 2012, pp. 140–144.

[35] X. Yang, M. Cai, and J. Li, "Path planning for unmanned aerial vehicles based on genetic programming," in *2016 Chinese Control and Decision Conference (CCDC)*, 2016, pp. 717–722.

[36] K. Vicencio, B. Davis, and I. Gentilini, "Multi-goal path planning based on the generalized traveling salesman problem with neighborhoods," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2014, pp. 2985–2990.

[37] H. Burchardt and R. Salomon, "Implementation of path planning using genetic algorithms on mobile robots," in *2006 IEEE International Conference on Evolutionary Computation*, 2006, pp. 1831–1836.

[38] A. F. d. V. Machado, U. O. Santos, H. Vale, R. Gonçalvez, T. Neves, L. S. Ochi, and E. W. G. Clua, "Real time pathfinding with genetic algorithm," in *2011 Brazilian Symposium on Games and Digital Entertainment*, 2011, pp. 215–221.

[39] F. Lampathaki, C. Agostinho, Y. Glikman, and M. Sesana, "Moving from 'black box' to 'glass box' artificial intelligence in manufacturing with xmanai," in *2021 IEEE International Conference on Engineering, Technology and Innovation (ICE/ITMC)*, 2021, pp. 1–6.

[40] F. Doshi-Velez and B. Kim, "Towards a rigorous science of interpretable machine learning," *arXiv: Machine Learning*, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:11319376

[41] R. Andrews, J. Diederich, and A. B. Tickle, "Survey and critique of techniques for extracting rules from trained artificial neural networks," *Knowledge-Based Systems*, vol. 8, no. 6, pp. 373–389, 1995, knowledge-based neural networks. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0950705196819204

[42] A. A. Freitas, "Comprehensible classification models: a position paper," *SIGKDD Explor. Newsl.*, vol. 15, no. 1, p. 1–10, mar 2014. [Online]. Available: https://doi.org/10.1145/2594473.2594475

[43] E. Frank and I. Witten, "Generating accurate rule sets without global optimization," *Machine Learning: Proceedings of the Fifteenth International Conference*, 06 1998.

[44] Y. Mei, Q. Chen, A. Lensen, B. Xue, and M. Zhang, "Explainable artificial intelligence by genetic programming: A survey," *IEEE Transactions on Evolutionary Computation*, vol. 27, no. 3, pp. 621–641, 2023.

[45] G. F. Smits and M. Kotanchek, *Pareto-Front Exploitation in Symbolic Regression.* Boston, MA: Springer US, 2005, pp. 283–299. [Online]. Available: https://doi.org/10.1007/0-387-23254-0_17

[46] M. Keijzer and J. Foster, "Crossover bias in genetic programming," in *Genetic Programming*, M. Ebner, M. O'Neill, A. Ekárt, L. Vanneschi, and A. I. Esparcia-Alcázar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 33–44.

[47] M. Kommenda, G. Kronberger, M. Affenzeller, S. M. Winkler, and B. Burlacu, *Evolving Simple Symbolic Regression Models by Multi-Objective Genetic Programming.* Cham: Springer International Publishing, 2016, pp. 1–19. [Online]. Available: https://doi.org/10.1007/978-3-319-34223-8_1

[48] E. J. Vladislavleva, G. F. Smits, and D. den Hertog, "Order of nonlinearity as a complexity measure for models generated by symbolic regression via pareto genetic programming," *IEEE Transactions on Evolutionary Computation*, vol. 13, no. 2, pp. 333–349, 2009.

[49] L. Vanneschi, M. Castelli, and S. Silva, "Measuring bloat, overfitting and functional complexity in genetic programming," 07 2010, pp. 877–884.

[50] A. Eiben and J. Smith, *Introduction To Evolutionary Computing*, 01 2003, vol. 45.

[51] K. Miettinen, "Introduction to multiobjective optimization: Noninteractive approaches," in *Multiobjective Optimization*, 2008. [Online]. Available: https://api.semanticscholar.org/CorpusID:43521644

[52] C. M. Fonseca and P. J. Fleming, "An overview of evolutionary algorithms in multiobjective optimization," *Evolutionary Computation*, vol. 3, no. 1, pp. 1–16, 1995.

[53] H. Katagiri, K. Hirasama, and J. Hu, "Genetic network programming - application to intelligent agents," in *Smc 2000 conference proceedings. 2000 ieee international conference on systems, man and cybernetics. 'cybernetics evolving to systems, humans, organizations, and their complex interactions' (cat. no.0*, vol. 5, 2000, pp. 3829–3834 vol.5.

[54] R. Kumar, "Analysis of shape alignment using euclidean and manhattan distance metrics," in *2017 International Conference on Recent Innovations in Signal processing and Embedded Systems (RISE)*, 2017, pp. 326–331.