Developing and Evaluating Incremental Evolution using High Quality Performance Measures for Genetic Programming

A thesis presented in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science
at Massey University, Albany, Auckland,
New Zealand

Matthew Garry William Walker
2007

To Mum and Dad

Abstract

This thesis is divided into two parts. The first part considers and develops some of the statistics used in genetic programming (GP) while the second uses those statistics to study and develop a form of incremental evolution and an early termination heuristic for GP.

The first part looks in detail at success proportion, Koza's minimum computational effort, and a measure we rename "success effort". We describe and develop methods to produce confidence intervals for these measures as well as confidence intervals for the difference and ratio of these measures.

The second part studies Jackson's fitness-based incremental evolution. If the number of fitness evaluations are considered (rather than the number of generations) then we find some potential benefit through reduction in the effort required to find a solution. We then automate the incremental evolution method and show a statistically significant improvement compared to GP with automatically defined functions (ADFs).

The success effort measure is shown to have the critical advantage over Koza's measure as it has the ability to include a decreasing cost of failure. We capitalise on this advantage by demonstrating an early termination heuristic that again offers a statistically significant advantage.

Preface

Acknowledgements

First, I'd like to thank Mum and Dad. Without their support and—equally importantly—encouragement, I wouldn't be here at the end of this experience. Thank you for your enthusiasm, for putting up with draft chapters being read to you, and for listening to ideas when mere mortals would have fallen asleep.

Chris Messom, my supervisor, has been great. My PhD-studying friends have had some horror stories to tell regarding their supervisors, but Chris was fantastic. He organised funding (thank you to the Government for their "Top Doctoral" programme), was ridiculously patient, always supportive and thorough and motivational when things weren't working. Another of Chris's wonders is his ability to respond. The university recommends forming an agreement with supervisors to return work within a specific time. They recommended a turn-around time of two weeks. I think Chris might have averaged two hours. Thanks Chris.

Howard Edwards has been a huge resource for the statistics in this thesis. He provided excellent assistance and discussion. He also provided great motivation as it was fantastic to work with someone who was excited by the ideas. Although all the work presented in this thesis is my own, it is Howard and Chris and I who are the "we" in this thesis.

Hopefully we can convince Bruce Mills to return from the United Arab Emirates. Bruce provided the most entertaining conversations (that could easily eliminate an afternoon) introducing me to evolutionary ideas that I still think about. But possibly most importantly to me was that it was his suggestion that gave me a concrete starting point. The ideas transitioned unrecognisably but finally they formed this thesis. His influence was an excellent catalyst just when I needed it.

Christian Gagné produced Open BEAGLE, GP software that I have contributed to and have used almost exclusively throughout the thesis. He provided excellent practical discussions surrounding GP and was very helpful when I requested large datasets of GP experiments.

Steffen Christensen was also of considerable assistance when he provided his absolutely enormous artificial ant datasets.

I'd also like to say a very big thank you to my little sister and my friends. You provided excellent conversations and emails, or lovely distracting phone calls and good vibes, or excellent dinners, or just a break away from this work. You also applied what was highly necessary pressure to get this work done. Thanks.

Finally, I'd like to thank the anonymous reviewers of papers I submitted, especially the reviewer who voted as best paper the work that turned into chapter 3.

Publications

Parts of this thesis have already been published. The initial work surrounding the Wilson-Dependent method in chapter 3 was published at *EuroGP 2007* [114]. The discussion of its reliability (section 3.2) was published at *GECCO* a few months later [115], as was a summary of the confidence interval method for the success effort statistic (chapter 4) [117]. The more detailed analysis of success effort that appears in chapter 4 along with parts of the literature review in chapter 2 and the comparisons that appear in chapter 5 were later published at *CEC* [116].

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis is divided into two major parts. Part I studies the statistics used in genetic programming while Part II utilises those statistics to study incremental evolution—a variation to the genetic programming algorithm.

Genetic programming is one of a class of techniques that fit under the umbrella term evolutionary computation, which is also known as evolutionary algorithms. Evolutionary computation is a subfield of machine learning (ML) which itself is a subfield of artificial intelligence (AI). The focus of machine learning is the development and use of algorithms and techniques that allow computers to learn; evolutionary computation attempts to achieve that goal through the use of artificial evolution.

We will use an incremental learning algorithm to encourage evolution of a successful individual. Incremental evolution is a process of starting from a simple evolutionary environment that is related to the problem domain, but is not as challenging. Once success is attained in the simple domain, the population is transfered to a harder domain, and evolution is allowed to continue. Ideally, the evolutionary environments become more and more challenging until finally the goal environment is reached and a successful individual is evolved.

This chapter offers an introduction to some of the main concepts in this thesis.

1.1 Genetic Programming

Genetic programming (GP) is an automated technique that produces computer programs. Although it is very expensive computationally, GP has been very successfully applied in some fields. For example, in more than twenty occasions it has been so successful that it has achieved levels of performance equivalent to patentable ideas [74, section 1.1.5].

GP was motivated by the ideas behind Darwin's theory of evolution: a population of computer programs is kept; "survival of the fittest" is implemented by assessing the computer programs and culling those that are least effective at their allocated task; computer programs are bred using both sexual reproductive operators (termed crossover) and by asexual operators (termed mutation); the selection and breeding cycles (generations) are continued until a solution is found or the allocated processing resources have been spent.

A reader familiar with Genetic Algorithms (GAs) [60] will note the similarity of the GP algorithm to GAs—indeed, GP was borne out of GAs [71].

A reader interested in an introduction to GP might turn to many of the tutorials available on the Internet. For a more thorough treatment I recommend *Genetic Programming: An Introduction* by Wolfgang Banzhaf et al. [13] or the original text by John Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection* [71]. A very recently published technical report by Riccardo Poli et al. [96] offers a freely-available excellent tutorial and review of the progress in the field.

1.2 Performance Measurements

It is typical in GP research to offer a modification to the original Koza-style (canonical) approach. A few examples of previous modifications include the ability to automatically define functions [7, 72], the use of semi-isolated islands [73], a reduction in the processing requirement to assess individuals' fitness [44] and the subdivision of evolution into a sequence of tasks [106]. The developers of each of these ideas have desired to compare their new idea to the standard approach. To do this a number of measures have been used.

In chapter 2 we review many of the statistics that have been used in the field. Two performance measurements dominate the literature: Koza's minimum computational effort (which we discuss in detail in chapter 3) and mean best-of-run fitness (which we analyse in chapter 5). Other methods include success proportion and mean generation (also analysed in chapter 5). However a primary contribution of the first part of this thesis is to re-introduce and study a measure we term success effort (see chapter 4).

The notable problem with a large portion of GP literature is a lack of consideration of sampling error. Sampling error is often measured by calculating a confidence interval, which is an indication of how confident you should be in a given measure. If you are given two measurements, say 500,000 and 600,000, how confident are you that the sources of the two measurements are in fact different? If you are given the two measurements and told that they could both be "out" by up to 200,000 then you now have reason to suspect that the two sources may, in fact, not differ. A confidence interval tells you how far "out" a measure may be. Without them, as Peter Angeline said, comparisons are inconclusive [8].

To be fair, before our studies, no one had a reliable method to generate confidence intervals for Koza's minimum computational effort. It is another primary contribution of the first part of this thesis that we offer such a method. We also offer a confidence interval method for the success effort statistic and we collate the "best practice" methods for success proportion, mean best-of-run fitness, and mean generation (in chapter 2).

1.3 Performance Comparisons

Performance comparisons between the canonical form of GP and a researcher's modification have typically been made with either what Koza termed an efficiency ratio [72] or by plots per-generation of success proportion or mean best-of-run fitness.

Another technique to compare performance of two measures is to assess whether their confidence intervals overlap. Non-overlapping confidence intervals indicate that one can have confidence that the sources of the two measures are, in reasonable likelihood, different.

That technique is not as powerful as the use of differences of two measures or the use of Koza's efficiency ratios of two measures. However we still need confidence intervals for differences and ratios. We develop methods to generate confidence intervals for differences and ratios for both minimum computational effort and success effort. We also collate best-practice (but fairly unknown in GP) methods for success proportion.

Further, in sections 3.5 and 4.3 we show that the use of per-generation plots for success proportion can produce misleading conclusions.

1.4 Incremental Evolution and Cost of Failure

In the second part of this thesis we apply the knowledge obtained from the first part to assess the performance improvement offered by the use of a form of incremental evolution. Incremental evolution is a technique (similar to layered learning [106, 107]) that breaks up the evolutionary goal into a series of sub-goals. Chapter 6 discusses the literature surrounding these ideas.

The specific form of incremental evolution is David Jackson's fitness-based incremental evolution [67]. We find that it is very difficult to get his method to outperform canonical GP—especially when using standard measures of the computational cost of individuals evaluated. However, fitness-based incremental evolution reduces the number of fitness evaluations required, and if one is prepared to measure the cost of evolution in terms of evaluations rather than individuals (a viewpoint we term *adjusted generations*), then fitness-based incremental evolution shows some potential.

We then further investigate the idea by automating the stages in fitness-based incremental evolution, and show (under adjusted generations) a statistically significant improvement in performance.

One of the keys that we required in order to demonstrate this improvement was the use of the success effort statistic. In the second part of the thesis one advantage of success effort as a measure is that (unlike minimum computational effort) success effort includes the cost of failure. We further capitalise on this ability to measure the cost of failure when (in chapter 10) we offer an early termination heuristic for genetic programming.

Without the statistical techniques developed in the first part, we would have taken the traditional route and stated the size of measured effects, but we would have been able to give no indication of the variability we would expect from our results. Thanks to the statistical methods we develop, we are able to give not only the expected size of effects, but also the confidence that one can have in the results.

1.5 Simple Problems

Throughout this thesis we study simple problems. In fact, it is a potential issue with minimum computational effort, success effort and success proportion that *success* in the problem is required. Further, for statistical comparisons, the measures require a fair proportion of runs to succeed. Appendix A considers this topic.

The use of simple problems is however very common in science. It is through the process of many experiments that we can study new ideas. In order to execute many experiments we require that they are not too expensive—and thus they are labelled simple.

The statistics offered and recommended in this thesis assume this experimental paradigm. When “hard problems” are being considered it is possible to categorise experiments as successes or failures by using a performance threshold (a technique discussed in section A.3.3). However even if this approach isn’t acceptable, we hope that, from having used these methods on simpler problems, the lessons learnt will allow for greater success when GP is applied to the real world.

1.6 Objectives and Motivation

Incremental evolution, in the fitness-based form that we study, is very widely applicable to genetic programming, and yet it is challenging to apply it successfully. The central focus of this thesis is to study and improve incremental evolution under genetic programming. To do this we required statistical methods to compare empirical results.

Specifically, we use the standard scientific method to answer the following research questions:

- What methods produce reliable confidence intervals for Koza’s Minimum Computational Effort?
- What methods produce reliable confidence intervals for Success Effort?
- What are the best-practice methods for the production of confidence intervals for Success Proportion, Mean Fitness, and Mean Generation?
- Does manual incremental evolution have the potential to outperform direct evolution when the generations are weighted based on the amount of work done per generation (a measure we term adjusted generations)?
- Does automatic incremental evolution on the even-n-parity problem domain, statistically significantly outperform direct evolution using adjusted generations?
- Does the early termination heuristic benefit direct evolution (over even-n-parity, symbolic regression, multiplexor and ant problems) using adjusted generations? And does the early termination heuristic benefit automatic incremental evolution?

1.7 How to Read this Thesis

If your primary interest is in how this thesis applies to genetic programming, then please read Part II first. This will provide a demonstration for the benefits of success effort versus minimum computational effort.

If you would like to apply the statistics used in Part II to your own experimental work, then the key algorithms are in: tables 4.2 and 4.7 (pages 75 and 80) for success effort; tables 3.4 and 3.19 (pages 33 and 64) for minimum computational effort; and equations 2.3 and 2.4 and table 2.3 (pages 13 and 16) for success proportion.

If you wish to develop the statistics used in genetic programming, or would like to consider the evidence that the recommended methods do actually work as intended, then Part I will be of interest.

Part I

High Quality Performance Measures for Evaluating Genetic Programming

Chapter 2

Review: Performance Comparison Measures

When researchers make alterations to the genetic programming algorithm they almost invariably wish to measure the change in performance of the evolutionary system. No one specific measure is standard, but Koza's minimum computational effort statistic has been frequently used [85]. Other measures that have been considered with varying popularity include: success proportion, mean fitness, mean best-of-generation fitness, mean best-of-run fitness, mean generations (or evaluations), average evaluations to success, the y-test, effective success probability, effective mean best fitness, and success effort[1].

This chapter reviews these measures and the literature that has discussed performance comparisons in genetic programming. We focus primarily on the measures that are used in this thesis: success proportion, minimum computational effort, and success effort. First, however, we diverge for a discussion on the importance of statistics.

2.1 The Importance of Statistics

> Statistics may be defined as the study of variability. If there were no variability there would be no need for the science of statistics.
>
> — *Clarke and Cooke* [24]

Genetic programming is a non-deterministic algorithm [71]. A consequence of this is that one run will most likely produce a completely different result to

[1] To avoid confusion, throughout this thesis, we refer to Miller and Thomson's "hit effort" [87] as "success effort" (see section 2.8).

the next. To answer questions like "how likely is it that I'll find a solution?", we need to execute a number of runs to estimate the answer.

With two different GP systems one can get two such answers. For example, configuration A might produce five successes out of ten runs (50% success), while configuration B might produce seven successes from ten runs (70% success). Can we conclude that configuration B is better than configuration A? As Peter Angeline remarked [8], we cannot make that conclusion if all we have are the percentages. We need to consider, for example, if we were to do the experiment again, what are the chances that both configurations produce 60% success? Or—possibly worse still—what are the chances that configuration B's performance is beaten by configuration A's?

The answer depends on the variability associated with the two measures. The use of statistics can turn the measure and its variability into a statement that we either can or cannot be confident that the two configurations are indeed different.

(It turns out that, had ten runs been executed for both configurations, Wilson's method—discussed in section 2.2.2—would have informed us that we could not be confident that they were indeed different.)

There are other techniques to indicate variability other than the use of confidence intervals. Such techniques include the use of standard deviation and standard error. However, both of these fail to allow a reader to quickly ascertain whether two results are statistically significantly different. The use of confidence intervals allows for more intuitive comprehension of results.

2.1.1 Comparison of Two Results

The comparison of two confidence intervals is however not entirely straightforward.

> It is a common statistical misconception to suppose that two quantities whose 95% confidence intervals just fail to overlap are significantly different at the 5% level.
>
> — *Goldstein and Healy* [49]

A better approach (than the direct comparison of two confidence intervals) is a confidence interval for the difference of, or the ratio of, two values. If the confidence interval for the difference includes zero then we cannot be confident that a difference exists. Equally, if the confidence interval for the ratio of two values includes one, then we cannot be confident that a difference truly exists.

Further, these are more powerful techniques (that is, they are more likely to detect a difference when a difference truly exists) than direct comparison of two confidence intervals.

However, it is too common in the literature in genetic programming that a point statistic will be quoted with absolutely no indication of its variability. The remainder of this chapter discusses the measures that have been used in the field and the statistical issues that have been raised.

2.2 Success Proportion

Success proportion is at the heart of Koza's minimum computational effort and it is also critical to effective success probability and success effort. However, it has frequently been used in its own right—commonly in the same way as Koza used it in his first and second books on GP [71, 72]—plotted per generation in order to assist comparison of two or more GP variations. An example of its use can be found in the left graph of figure 3.12 (page 66).

In Koza's parlance [71], the instantaneous probability of success, $Y(i)$, is the proportion of runs that found a solution after completing execution of generation i. Success proportion, $P(i)$, is the sum of all values of instantaneous probability of success at or before the given generation:

$$P(i) = \int_0^i Y(g) \cdot dg \tag{2.1}$$

In other words, success proportion is the number of runs that have found a solution at or before generation i divided by the total number of runs in the experiment. For GP experiments with discrete generations the discrete version of success proportion is normally used:

$$P(i) = \sum_{g=0}^{i} Y(g) \tag{2.2}$$

2.2.1 Dependence

You should note that when success proportion is plotted per generation, the results are almost certainly not independent of each other. It is most likely that only one set of experiments was executed and the success proportion at each generation was plotted. For the results to be independent one set of experiments would have to be executed up to generation i for each i from zero to the final generation. This topic was discussed by Luke and Panait [85] and we will return to it in appendix A.

The lack of statistical independence reduces our ability to make generation-to-generation comparisons. However, it does not limit the statistical validity of comparisons at a specific generation (for example, the last generation of the run).

2.2.2 Confidence Intervals

Because success proportion is a proportion, there are a number of statistical methods that can be applied to produce associated confidence intervals—in fact, Newcombe studied seven different methods [91].

Normal-Approximation

Possibly the most common technique is based on an approximation to the normal distribution. It is assumed that for cases where $np > 5$ and $n(1-p) > 5$ then confidence intervals are given by:

$$p \pm z_{\text{norm}} \sqrt{\frac{p(1-p)}{n}}$$

where p is the success proportion, n is the number of runs, and z_{norm} is the standard normal variate (1.96 for 95% confidence intervals) [24]. However this method was considered unacceptable by Newcombe: "it is strongly recommended that intervals calculated by [this method] should no longer be acceptable for scientific literature; [its use] should be restricted to ... introductory teaching purposes"[91]. He pointed out that the method suffers from overshoot as it can produce intervals that include values beyond what is possible. It can also produce intervals with zero-width: a statistical aberration. Finally, he demonstrated that there were many "totally unacceptable" instances where "95%" confidence intervals did not include the true value 95% of the time.

Wilson's "Score" Method

Instead of the normal-approximation method, Newcombe favoured Wilson's "score" method [123]. To calculate a 95% confidence interval for the true but unknown proportion of successes based on the observed sample proportion of successes, $p = r/n$, given r successes from n runs, these formulae [91] may be used (where the standard normal variable $z_{\text{norm}} = 1.96$):

1. Obtain n independent runs. Count the number of successes n_{s}, the number of failures n_{f}. $p = \frac{n_{\text{s}}}{n}$.
2. Produce B random variables that follow a beta distribution with parameters $\alpha = pn + 1$ and $\beta = (1 - p)n + 1$. Label these P.
3. Find the $\frac{\epsilon}{2}$ and $\frac{1-\epsilon}{2}$ quantiles of P. These are the limits of a $1 - \epsilon$ confidence interval for the true value of the success proportion.

Table 2.1: Simulation algorithm to produce confidence intervals for the success proportion statistic.

$$upper(p, n) = \frac{2np + {z_{\text{norm}}}^2 + z_{\text{norm}}\sqrt{{z_{\text{norm}}}^2 + 4np(1-p)}}{2(n + {z_{\text{norm}}}^2)} \tag{2.3}$$

$$lower(p, n) = \frac{2np + {z_{\text{norm}}}^2 - z_{\text{norm}}\sqrt{{z_{\text{norm}}}^2 + 4np(1-p)}}{2(n + {z_{\text{norm}}}^2)} \tag{2.4}$$

Newcombe showed that Wilson's method suffers from neither overshoot nor the possibility of producing zero-width intervals, and that it has an estimated mean coverage of 0.952 (which should be compared to the ideal of 0.95 and to the estimation of 0.88 for the normal-approximation method).[2] A continuity-corrected form was also studied but Newcombe concluded it was unnecessarily conservative.

Beta Distribution: Simulation

An alternative method based on the Beta distribution, which we will utilise in the next two chapters, is to produce random numbers based on the likelihood function. The 2.5% and 97.5% quantiles of a sufficient quantity of such random numbers represent confidence limits for a 95% confidence interval. The likelihood function of the true probability of success is proportional to a Beta distribution whose α variable is $np + 1$ and whose β variable is $n(1 - p) + 1$, where p is the success proportion and n is the number of runs [79]. Table 2.1 defines this algorithm.

Kim used another approach, but it was also based on use of the Beta distribution [70]. Kim's method was also described and used by Yannakakis et al. [43].

[2] Newcombe's study was of 96,000 random pairs of n and p with $5 \leq n \leq 100$ and $0 \leq p \leq 0.5$.

2.2.3 Confidence Intervals for Differences

Newcombe studied eleven methods for calculating confidence intervals for the difference of two proportions [90]. He noted that that two methods based on Wilson's score method were "remarkable": "They are computationally very tractable ... free from aberrations, and achieve better coverage properties than any except the most complex methods". The methods came "strongly recommended" over those commonly in use.

The two methods were a continuity-corrected version and one that was not corrected. Just as the continuity-corrected Wilson method for confidence intervals for a single proportion was quite conservative, so too was the continuity-corrected method for the difference. As Newcombe said, "[this] may be interpreted to mean the interval is simply unnecessarily wide." As a consequence we shall make use of the non-continuity-corrected version.

Table 2.2 gives an algorithm, based on Wilson's score method, for calculating a confidence interval for the difference of two proportions. Newcombe's study showed this method to have mean coverage close to the nominal $1 - \alpha$ level: 96.0% when 95% was specified, 91.6% when 90% was specified, and 99.2% at a nominal rate of 99%. Minimum coverage for those levels were 86.7%, 82.3%, and 91.7%.

2.2.4 Confidence Intervals for Ratios

Although Newcombe offered a Wilson-based method for the confidence interval of an odds-ratio of two proportions [90, appendix I], he did not offer a Wilson-based method for the confidence intervals for a ratio of two proportions (also termed a rate-ratio). Instead he referred to studies by Miettinen and Nurminen, and by Rothman. We implemented Miettinen and Nurminen's method [86] and use it throughout Part II of this thesis. R/S-PLUS code to implement their method is given in table 2.3.[3]

2.2.5 Effective Success Probability

An interested reader might also like to consider Steffen Christensen's *effective success probability* which allows comparison of runs with different population sizes [21, chapter 3].

[3] Although I have modified this code, I would like to gratefully acknowledge Brad Biggerstaff as the primary author [18].

1. Given a successes from m trials of the first experiment and b successes from n trials of the second experiment, the two proportions of interest are $p_1 = \frac{a}{m}$ and $p_2 = \frac{b}{n}$. The observed difference is $\theta = p_1 - p_2$.

2. The lower limit of a confidence interval at the $1 - \alpha$ level is given by $\theta - \delta$ and the upper limit is given by $\theta + \epsilon$, where z is the $1 - \frac{\alpha}{2}$ point of the standard Normal distribution, $q_1 = 1 - p_1$, $q_2 = 1 - p_2$, and:

$$\begin{aligned}
l_1 &= \frac{2a + z^2 - \left(z \cdot \sqrt{z^2 + 4a \cdot q_1}\right)}{2(m + z^2)} \\
u_1 &= \frac{2a + z^2 + \left(z \cdot \sqrt{z^2 + 4a \cdot q_1}\right)}{2(m + z^2)} \\
l_2 &= \frac{2b + z^2 - \left(z \cdot \sqrt{z^2 + 4b \cdot q_2}\right)}{2(n + z^2)} \\
u_2 &= \frac{2b + z^2 + \left(z \cdot \sqrt{z^2 + 4b \cdot q_2}\right)}{2(n + z^2)} \\
\delta &= \sqrt{(p_1 - l_1)^2 + (u_2 - p_2)^2} \\
\epsilon &= \sqrt{(u_1 - p_1)^2 + (p_2 - l_2)^2}
\end{aligned}$$

Table 2.2: An algorithm to produce a confidence interval at the $1 - \alpha$ level for the true difference between two proportions [82, 90].

```
# Required arguments:
# x1, x2: number of 'successes' for binomal variates
# n1, n2: number of observations for each
# Optional arguments:
# alpha: confidence coefficient
# Author:
# Brad Biggerstaff (2001)

proportion_ratio_ci <- function(x1, n1, x2, n2, alpha = 0.05)
{
  chi2RR <- function(rr, x1, n1, x2, n2, alpha) {
    n <- n1 + n2
    A <- n * rr
    B <-  - (n1 * rr + x1 + n2 + x2 * rr)
    C <- x1 + x2
    R2 <- ( - B - sqrt(B^2 - 4 * A * C))/(2 * A)
    R1 <- R2 * rr
    V <- (((R1 * (1 - R1))/n1 + ((rr^2) *
         R2 * (1 - R2))/n2) * n)/(n - 1)
    r1 <- x1/n1
    r2 <- x2/n2
    X2 <- ((r1 - r2 * rr)^2)/V
    chisq.crit <- qchisq(1 - alpha, 1)
    result <- X2 - chisq.crit
    result
  }
  point.est <- (x1/n1)/(x2/n2)
  lower.limit <- uniroot(chi2RR, c(1e-007, point.est),
                         x1 = x1, n1 = n1,
                         x2 = x2, n2 = n2, alpha = alpha)$root
  upper.limit <- uniroot(chi2RR, c(point.est, 1000),
                         x1 = x1, n1 = n1,
                         x2 = x2, n2 = n2, alpha = alpha)$root
  answer <- data.frame(point.est = point.est,
                       lower.limit = lower.limit,
                       upper.limit = upper.limit,
                       confidence = 1 - alpha)
  answer
}
```

Table 2.3: R/S-PLUS code to produce a $1 - \alpha$ confidence intervals for the ratio of two proportions.

2.3 Minimum Computational Effort

Koza's minimum computational effort has been one of the most used statistics in the field of genetic programming [85]. In was used heavily throughout Koza's first two books on GP [71, 72]. It was also utilised in his third book, but to a lesser extent as Koza has appeared to veer away from performance-oriented discussion and instead focused on discussions about the best evolved individual [73, 74]. Minimum computational effort's popularity has decreased over time. This could be due to a series of papers highlighting significant concerns. This section discusses those concerns while chapter 3 discusses confidence intervals for minimum computational effort.

2.3.1 Koza's Original Definition

In *Genetic Programming*, Koza described a statistic to assess the computational burden of using GP to find a solution [71, chapter 8]. It calculates the minimum number of individuals that must be evaluated in order to yield a solution 99% of the time and was termed minimum computational effort, E.

Computational effort is calculated from the observed success proportion $P(i)$ (equation 2.1). Given the probability of success from a number of GP runs, we can calculate how many runs would be required, $R(P(i), z)$, in order to find a solution at generation i with probability z:

$$R(p, z) = \left\lceil \frac{\log(1-z)}{\log(1-p)} \right\rceil \tag{2.5}$$

where $\lceil \cdot \rceil$ indicates the ceiling operator (also known as "rounding up"). As is typical in the literature, z will be set to 0.99 throughout this work.

Now we may calculate the computational effort (the number of individuals that need to be evaluated to find a solution 99% of the time) for generation i with a population of M individuals.

$$I(i, z) = (i+1) \cdot R(P(i), z) \cdot M \tag{2.6}$$

Koza's minimum computational effort, E, is the minimum value of $I(i, z)$ over the range of generations from 0 to the maximum in the experiment.

2.3.2 Dropping the Ceiling Operator

Christensen and Oppacher [22] suggested that “the GP community might be well served by dropping the ceiling operator, although this may be subject to debate”. They showed that use of the ceiling operator tended to overestimate the computational effort required.

Up to seven years earlier, Andre and Koza [4, 5] had already dropped the use of the ceiling operator. Their work was republished in *Genetic Programming III* in which Koza et al. said that “the rounding up is not required for computing $R(z)$” [73, page 208]. The ceiling operator was not used in at least four chapters of the book[4] but no rationale was given for dropping it.

Removing the ceiling operator makes the formula for calculating $R(p, z)$ a little more complex. There exists an issue about computational efforts calculated at generations where the cumulative probability of success is greater than z: if the only change to the definition of R is to remove the ceiling operator, then at such points the number of runs required drops below one, and it becomes very difficult to see the practical meaning of the measure. However an even more significant issue exists when every run eventually finds a solution. In such situations, the cumulative probability of success reaches 1.0 and $R(1.0) = 0$. Thus the number of runs required is zero. This makes the computational effort equal to zero, and thus the minimum computational effort also zero. This cannot be acceptable.

Our opinion is that the number of runs required should have a lower bound of one. This means that if the ceiling operator is to be dropped, then:

$$R(p, z) = \begin{cases} \frac{\log(1-z)}{\log(1-p)} & \text{if} \quad p < z \\ 1 & \text{if} \quad p \geq z \end{cases} \tag{2.7}$$

Although they did not make it explicit, Koza et al. must have had a similar expectation in *Genetic Programming III* as, although the text declares elimination of the ceiling operator [73, page 330], the graph [73, figure 21.2] effectively demonstrates the statement $R(1.0) = 1$.

2.3.3 Underestimating the True Computational Effort

Although the use of the ceiling operator tends to overestimate the computational burden, Christensen and Oppacher [22] showed that the use of the minimum operator tends to produce an underestimate of the true computational effort

[4] Sections 21.1 and 62.3 and chapters 54 and 55 of *Genetic Programming III* all drop the use of the ceiling operator.

required. For their example (albeit artificial) case they demonstrated that the use of the minimum operator produced an underestimate almost 90% of the time.

Christensen and Oppacher also demonstrated this underestimation with a real GP problem, the artificial ant on the Santa Fe Trail [71]. They ran a remarkable 27,755 runs to find a best estimate of the true computational effort. They then selected 10,000 random subsamples of 50 runs from that data. They found that almost 70% of the observed values for computational effort were below their best estimate of the true value. The median computational effort of the subsamples was just 80% of their estimate of the true computational cost.

In order to reduce the magnitude of this underestimation, Christensen and Oppacher recommended that GP “practitioners choose relatively large run counts (on the order of 500 runs)”.

2.3.4 Influence of Probability of Success

Miller and Thomson [87] demonstrated results where the minimum computational effort (E) was being calculated at generations where very few of the runs had found a solution. They detailed 24 experiments with the artificial ant on the Santa Fe trail; each was run 100 times. When calculating the minimum computational effort statistics they found that half the values were obtained at generations where fewer than 10 runs had found a solution. They concluded that were their experiments to be run again, the results were “likely to vary enormously”.

Luke and Panait [85] pointed out the same issue: “changes in the Individuals to be Processed measure and its derived Computational Effort measure are both greatly exaggerated when small changes occur in ideal solution counts [number of hits]”.

Niehaus and Banzhaf [92] showed that as the probability of success decreased, the range of observed values for computational effort increases. Thus the confidence in the accuracy of the minimum computational effort should be reduced whenever the probability of success is low.

Unfortunately, some quoted minimum computational effort statistics do not state the number of runs that were successful (or even the generation at which they were obtained—which along with the population size would allow the calculation). The elimination of this information means that readers are not able to form even a feeling for the confidence they should have in the quoted statistic.

2.3.5 Number of Runs

Niehaus and Banzhaf [92] also demonstrated the impact of the number of runs. As would be expected, the greater the number of runs, the smaller the range of observed minimum computational efforts. However, they also showed that if the probability of finding a solution is low, a small number of runs can result in an enormous range of observed minimum computational efforts. For 50 runs with a probability of success of 0.2, they showed a range of observed values of more than 5-fold the theoretical minimum computational effort. Doubling the number of runs to 100 resulted in more than halving the observed range. They concluded that “calculating effort based on only 50 [runs] may lead to values quite off the theoretical values, and that even 200 [runs] often are not sufficient.”

2.3.6 Confidence Intervals

As Angeline [8] pointed out, a key problem with Koza’s computational effort statistic is that, as defined, it is a point statistic with no confidence interval. Without a confidence interval, comparisons are inconclusive.

Keijzer et al. [69] used resampling statistics to calculate confidence intervals on two problem domains. They used a bootstrap sample of 10,000 where they had executed 100 and 500 runs. However, they did not find the results very useful, “for the Santa-Fe problem . . . the width of the confidence interval (i.e. the uncertainty around the statistic) is nearly as large as the value of the computational effort itself. The confidence intervals clearly show that a straightforward comparison of computational effort, even differing in an order of magnitude, is not possible.”

Methods to generate confidence intervals for minimum computational effort are discussed and studied in chapter 3. The study includes the methods that Keijzer et al. may have used. We also offer methods to produce confidence intervals for the difference and for the ratio of two minimum computational efforts.

2.4 Mean Fitness

Mean fitness, as a measure of performance, vies with minimum computational effort as the most popular measure in the genetic programming field [85]. It is popular perhaps because the statistical issues surrounding the use of a mean are well understood. The measure and its confidence intervals may well have been introduced to the GP field by Angeline’s 1996 paper [8].

Mean best-of-run fitness is the sum of the fitness scores for the best individual in each run up to a specified generation, divided by the number of runs executed. The statistic is frequently measured at the final generation of each run, but it is possibly most common to see it graphed for every generation. Just like success proportion, when shown per generation, it is important to note that the results are typically not independent across generations.

There are at least three variations of the theme: mean average-of-generation fitness (also called mean population fitness [21]), mean best-of-generation fitness, and mean best-of-run fitness (where all the generations are considered and which we truncate to *mean best fitness*). Further, the variance of fitness is also a popular measure [13, section 8.4.3].

Mean fitness is also termed "mean number of hits", as Koza defined "hit" as success in a portion of the given problem [71]. Consequently, mean best fitness may also be termed "mean best number of hits".

Mean average-of-generation fitness has been shown to converge much more quickly. Christensen showed it to be more than three times as precise as mean best-of-generation fitness [21, page 84]. He also suggested that researchers may have preferred the measure given that "much of population genetics and GA theory refers to the behaviour of the mean fitness of the population". However, he concludes that "we are usually interested in finding the most successful individuals" and "the behaviour of an auxiliary set of solutions used during the searching process is not normally of great interest" [21]. Finally, he showed mean average-of-generation fitness is not a good predictor of mean best-of-generation fitness.

2.4.1 Confidence Intervals

Given that all forms of the measure are based on the mean, the same method can be used for the formation of confidence intervals. We will use mean best fitness as an example.

Mean best fitness is normally distributed (from the Central Limit Theorem [24]), but for the small sample sizes available from GP runs, it has been considered more appropriate to use a t-distribution [8, 24]. The parameters of the distribution can be approximated with those observed from the sample. A $1 - \alpha$ confidence interval can be obtained with the formula:

$$\text{mean}(f) \pm t_{(n-1,\alpha)} \frac{\text{sd}(f)}{\sqrt{n}} \tag{2.8}$$

where mean(f) is the mean best fitness, $t_{(n-1,\alpha)}$ is the t-statistic for the t-distribution with $n-1$ degrees of freedom and a cumulative probability of $1-\alpha$, sd(f) is the standard deviation of the fitness scores, and n is the number of runs executed.

2.4.2 Variations

Luke suggested a method "to calculate the expected maximum best-fitness-of run for N total runs", however he accepted the measure as a point statistic and thus did not offer a method to generate confidence intervals [83, 85].

Finally, just as with success proportion, an interested reader might also like to consider Christensen's *effective mean best fitness* which allows comparison of runs with different population sizes [21, chapter 3].

2.5 Mean Generation

When the vast majority of runs complete successfully, mean best fitness is not a useful statistic for differentiating between the performance of two GP variations. When this has occurred mean generation has been the preferred statistic [8, 25].

The mean generation is the sum of the generations at which termination occurred (irrespective of success or failure) divided by the number of runs that were executed.

2.5.1 Confidence Intervals

The mean of generation-to-termination follows a normal distribution (from the Central Limit Theorem [24]), however for sample sizes as small as the typical number of runs in a GP experiment, a t-distribution has been considered more appropriate [8, 24]. An approximate $1-\alpha$ confidence interval can be obtained with the formula:

$$\text{mean}(g) \pm t_{(n-1,\alpha)} \frac{\text{sd}(g)}{\sqrt{n}} \tag{2.9}$$

where mean(g) is the mean generation, $t_{(n-1,\alpha)}$ is the t-statistic for the t-distribution with $n-1$ degrees of freedom and a cumulative probability of $1-\alpha$, sd(g) is the standard deviation of the generations-to-termination, and n is the number of runs.

An alternative method to generate confidence intervals for generation-to-termination was used by Clegg et al. [25]. They used a Mann-Whitney U test (also known as a Wilcoxon rank sum test) [24] that effectively ranked the runs by generation. They gave no indication as to why they elected not to use the more traditional normal (or t-distribution) approximation. We do not recommend the use of the rank-sum or U test for this measure (unless the number of runs is very small) as it is statistically less powerful than tests based on the normal or t-distribution [24, page 397].

2.6 Average Evaluations to Success

Average evaluations to success is a measure not often used. It is defined as the average number of evaluations (or generations) of only the successful runs (i.e. the runs that failed are not included in this measure) [110, section 6.1.2].

Christensen considered this measure in his thesis and concluded it had a number of flaws [21]. It was sensitive to the cutoff used, correlated to population size, potentially unstable over the number of evaluations performed, and compromised even if used on a local scale.

If a practitioner must use it then one benefit is that, like mean fitness and mean generation, one can use a normal-approximation (or t-distribution for small samples) thanks to the Central Limit Theorem and the measure's use of a mean.

2.7 The Y-Test

In chapter 4 of his PhD thesis, Christensen considered the topic of comparing fitness values when different amounts of work had been done to obtain those values [21]:

> A common problem is that method A delivers better results than method B, but takes more fitness evaluations to do so . . . this issue can be directly answered by comparing the median performance of A to a specific cumulative proportion of B's performance that exactly compensates for the difference in evaluation count.

The y-test is a variation of the Mann-Whitney-Wilcoxon t-test that allows the user to state that two methods are either statistically significantly different or indistinguishable. For details, the interested reader is referred to Christensen's 15-page discussion of the statistic [21, pages 105–119].

The primary issue with the y-test is that, as it stands, it is inappropriate for use on experimental data where the computational efforts differ for each run. Section A.3.3 discusses this topic.

2.8 Success Effort

"Success effort" was a measure introduced by Miller and Thomson [87] under the name "hit effort". They defined a hit as a run that found a 100%-correct solution. Given that Koza had earlier defined the term "hit"—as an individual having success in a *portion* of the given problem [71]—we felt "hit effort" was a somewhat confusing name. Consequently, we have renamed Miller and Thomson's measure as "success effort".

Success effort measures the expected number of generations before a solution will be found. It may be calculated from a collection of runs by:

$$\frac{\text{mean}(g)}{p} \tag{2.10}$$

where g is the vector of generations at which the runs terminated (generations-to-termination) and p is the proportion of runs that found a solution.

In a similar vein to success effort, Lee [80] suggested "a measure of the average computing cost needed for the first successful run" [70]. This idea was extended by Kim and a method to produce confidence intervals was offered [70]. Their focus was on calculating the expected number of runs that would fail before a success was observed. They used this coefficient as a multiplier of the expected cost of failure with the expectation that the cost of failure was a constant. If it was not constant, then their method did not account for that variability. Further, they assumed that the computational effort only consisted of the runs that failed. Although they suggested a way to include the cost of success, their method again failed to include that variability.

Yannakakis et al. developed Kim's work and combined both the cost of failure and the cost of success [43]. However, again only the variability of the probability of success was accounted for in their method to produce confidence intervals.

Chapter 4 discusses the success effort statistic, and studies two techniques for the production of confidence intervals. The method of choice includes variability from the success proportion, the cost of failure, and the cost of success. We also detail methods to produce confidence intervals for the difference and for the ratio of two success efforts. Section 4.3 compares the measure to success proportion. Part II utilises this statistic's ability to include the cost of failure.

2.9 Summary and Further Reading

For this thesis we utilised Wilson's method—Newcombe's recommendation—for the production of confidence intervals for success proportion. When this was not appropriate we used the simulation algorithm in table 2.1. In section 3.5 we compare success proportion to minimum computational effort. We make a similar comparison to success effort in section 4.3. In chapter 5 we compare success proportion to a number of other statistics. Finally, in appendix A we tackle Luke and Panait's idealogical claims that ideal-solution counts (success proportions), and measures derived from them, are poor measures.

In chapter 3 we develop and analyse confidence interval methods for minimum computational effort and show that the chosen method is reliable.

We develop and analyse methods for confidence interval generation for success effort in chapter 4.

Chapter 5 compares mean best-of-run fitness, mean generation, success proportion, minimum computational effort and success effort.

Readers who are interested in further discussion of the statistics in genetic programming should consider chapter 8 of *Genetic Programming: An Introduction* [13] and chapters 3 and 4 of Christensen's thesis [21].

Chapter 3

Minimum Computational Effort

In *Genetic Programming* [71], Koza described minimum computational effort, a statistic to assess the computational burden of using GP. In section 2.3 we reviewed the literature that has shown up flaws in Koza's measure such as: it typically underestimates the true value, it is sensitive to small changes in success when success proportion is low and it is better to drop the use of the ceiling operator. But as Angeline [8] pointed out, a key problem with Koza's computational effort statistic is that, as defined, it is a point statistic with no confidence interval—and without a confidence interval, comparisons are inconclusive.

This chapter offers a number of methods to produce confidence intervals for Koza's measure. We compare them and study their performance under both real and simulated data.

3.1 Defining Confidence Intervals

3.1.1 Normal Approximation Method

This section discusses how an approximate 95% confidence interval can be generated for a computational effort statistic if the true *minimum generation* is known. The minimum generation is the generation at which the minimum computational effort occurs. The method used in this section is the textbook "normal approximation" method [24].

The cumulative probability of success statistic is calculated from the proportion of the population that has found a solution at a given generation. We may assume that this proportion is approximately normally distributed [24] and thus calculate an approximate 95% confidence interval using $p \pm e$ where $e = 2\sqrt{\frac{p(1-p)}{n}}$, p is the proportion of runs that found a solution by the specified generation and n

is the number of runs performed. Given that cumulative success cannot be below zero or above one, the confidence interval should be truncated to that range [91].

The minimum and maximum of this confidence interval can be used to generate an approximate 95% confidence interval for $\mathcal{R}$, the true number of runs required to find a solution with probability z:

$$\frac{\log(1-z)}{\log(1-(p+e))} \leq \mathcal{R} \leq \frac{\log(1-z)}{\log(1-(p-e))} \tag{3.1}$$

If the minimum and maximum of this range are used in place of $R(p, z)$ in the formula for computational effort (equation 2.6), the values can be used as an approximate 95% confidence interval for the true value of Koza's computational effort statistic, $\mathcal{I}(i)$, for generation i:

$$(i+1) \cdot M \cdot \frac{\log(1-z)}{\log(1-(p+e))} \leq \mathcal{I}(i) \leq (i+1) \cdot M \cdot \frac{\log(1-z)}{\log(1-(p-e))} \tag{3.2}$$

These confidence intervals are only valid while $np > 5$ and $n(1-p) > 5$ [24] where $p = P(i)$ and n is the number of runs that were executed.

3.1.2 Wilson-Dependent Method

This section discusses the replacement of the normal approximation method with Wilson's "score" method (which we discussed in section 2.2.2) when constructing confidence intervals for the computational effort statistic. We will call this method the *Wilson-Dependent* method—for reasons that should become clear in section 3.1.5. It is still assumed that the minimum generation is known.

Using the formulae in equations 2.3 and 2.4, a confidence interval can be established for the proportion of successful runs: the upper bound is given by $\mathit{upper}(P(i), n)$ and the lower bound is given by $\mathit{lower}(P(i), n)$. Just as was done in the previous section, the minimum and maximum of this range can then be used to calculate a maximum and minimum for the number of runs required to obtain a solution with probability z. These numbers can then be used with the known value for the population size, M, to find a 95% confidence interval for the true computational effort, $\mathcal{I}(i)$, at a given generation i:

$$(i+1) \cdot M \cdot \frac{\log(1-z)}{\log(1-\mathit{upper}(p,n))} \leq \mathcal{I}(i) \leq (i+1) \cdot M \cdot \frac{\log(1-z)}{\log(1-\mathit{lower}(p,n))} \tag{3.3}$$

1. Obtain n independent runs. Label these as the source set.
2. Repeat 10,000 times:
    (a) Select, with replacement, n runs from the source set.
    (b) Calculate the minimum computational effort statistic for the selection. If zero runs succeeded, the computational effort is infinite.
3. Find the 2.5% and 97.5% quantiles of the 10,000 computational effort statistics. These provide an upper and lower range on a 95% confidence interval for the true minimum computational effort.

Table 3.1: Algorithm for the Resampling method.

When the minimum generation is known, the use of the Wilson-Dependent method produces a valid confidence interval irrespective of the number of runs or the probability of success.

3.1.3 Resampling Statistics Method

Keijzer et al. [69] are the only group we have found who attempted to generate a confidence interval for Koza's computational effort statistic. We implemented a modified version of their method (see table 3.1).[1] When the true minimum generation is known, the minimum computational effort is calculated for the selection as the selection's computational effort at the true minimum generation. The resampling method always finds a confidence interval irrespective of the number of runs and the probability of success.

3.1.4 When Minimum Generation is Known

Testing the Validity of the Three Methods

In order to empirically test the validity of these three methods to generate confidence intervals, we ran experiments based on datasets where very large numbers of runs had been executed[2]. The four datasets were:

[1] Although Keijzer et al. did not clearly specify their method, they did state that they split the executed runs into two groups. We study that variation in section 3.2.1.

[2] The datasets and complete results are available for download. Please see Appendix D.

- *Ant*: Christensen and Oppacher's 27,755 runs [22] of the artificial ant on the Santa-Fe trail; panmictic population of 500; best estimate of the true computational effort 479,344 at generation 18[3]; $P(18) = \frac{2421}{27755} = 0.0872$
- *Parity*: 3,400 runs of even-4-parity without ADFs [71, 72]; panmictic population of 16,000; best estimate of true computational effort 421,074 at generation 23; $P(23) = \frac{3349}{3400} = 0.985$
- *Symbreg*: Gagné's 1,000 runs[4] of a symbolic regression problem ($x^4 + x^3 + x^2 + x$) [71]; panmictic population of 500; best estimate of true computational effort 33,299 at generation 12; $P(12) = \frac{593}{1000} = 0.593$
- *Multiplexor*: Gagné's 1,000 runs[5] of the 11-multiplexor problem [71]; panmictic population of 4,000; best estimate of true computational effort 163,045 at generation 25; $P(25) = \frac{947}{1000} = 0.947$

The computational effort calculations for each dataset (utilising every run) were treated as a best estimate of the true minimum generation and true minimum computational effort.

For each dataset and for each confidence interval generating method, the following method was applied. A subset of the whole dataset's runs were randomly selected (uniformly with replacement). The subset sizes were 25, 50, 75, 100, 200 and 500 runs. These sizes are typical of published work (often 25 to 100 runs, sometimes fewer [71, 72]) and recommendations by statisticians (200 to 500 runs [22, 92]). 10,000 subsets were selected and for each subset the confidence interval generating method was applied. This simulated 10,000 genetic programming experiments on each of the four problem domains for each of the six run sizes.

Results and Discussion

For each of the four problem domains and each of the three confidence interval generation methods, table 3.2 gives the average coverage and the average number of valid confidence intervals that were produced from the 10,000 simulated experiments. Table 3.3 gives the same statistics but by run size and method.

So, for example, table 3.2 shows that for the normal approximation method on the Ant problem domain, an average of 97.1% of the confidence intervals

[3] This occurred at generation 18 as, like Koza, we have counted the first generation as generation 0, whereas Christensen and Oppacher labelled it generation 1.

[4] Our thanks go to Christian Gagné for this dataset

[5] Thanks again to Christian Gagné for this dataset

Method \ Problem	Ant	Parity	Symbreg	Multiplexor	Average
Normal	97.1%	49.4%	95.3%	79.1%	80.3%
	7,049	1,787	9,954	4,752	5,885
Wilson-Dependent	95.2%	95.3%	94.9%	95.1%	95.1%
	10,000	10,000	10,000	10,000	10,000
Resampling	93.2%	69.9%	94.1%	88.9%	86.5%
	10,000	10,000	10,000	10,000	10,000

Table 3.2: Average coverage percentages and average validity statistics by problem domain when the minimum generation is known. Averages are over 25–500 runs.

Method \ Runs	25	50	75	100	200	500	Average
Normal	48.1%	70.5%	72.9%	98.0%	96.3%	95.9%	80.3%
	2,582	3,704	5,007	6,439	7,907	9,674	5,885
Wilson-Dependent	94.6%	95.7%	95.6%	94.7%	95.5%	94.7%	95.1%
	10,000	10,000	10,000	10,000	10,000	10,000	10,000
Resampling	72.0%	82.8%	86.9%	88.8%	94.4%	94.3%	86.5%
	10,000	10,000	10,000	10,000	10,000	10,000	10,000

Table 3.3: Average coverage percentages and average validity statistics by run size when the minimum generation is known. Averages are over the four problem domains.

included the true value of the minimum computational effort (compare that to the expected result of approximately 95%). This average was produced over simulated experiment sizes of 25–500 runs. The table also shows that, for the same setup, an average of 7,049 of the 10,000 simulated experiments produced valid confidence intervals.

The resampling method had a very poor minimum average coverage of 69.9% for the Parity domain (see table 3.2). The Normal method also did poorly for that domain with a coverage score of 49.4%. In contrast, the Wilson-Dependent method achieved very good coverage levels across all domains and all run sizes with a minimum coverage of 93.3% (on the Parity domain with 100 runs).

The advantage of the Wilson-Dependent method over the normal approximation method is clearly demonstrated by the validity statistics in the Parity problem. Because the probability of success is so high (0.985 over 3,400 runs), the samples with a low number of runs (25–200) were often unable to satisfy the normal method's validity criteria of $n(1 - p) > 5$. And even when the validity criteria were satisfied, for the small runs sizes (i.e. 50 and 75 runs), none of the confidence intervals included the best estimate of the true computational effort. The Wilson-Dependent method, on the other hand, produced valid confidence intervals for all 10,000 samples for every run size and with a coverage of 95.3% for the experiments in that domain. Where it was fair to make a comparison, the widths of the confidence intervals were similar.

The Ant domain exemplifies a low probability of success ($P(18) = 0.087$). In this case the Normal method had difficulty satisfying its $np > 5$ criteria, producing valid confidence intervals for only 6% of the samples with 25 runs and 43% with 50 runs. However, for the confidence intervals that it did produce, the proportions that included the true value either exceeded or were very close to the intended 95%. However, yet again the Wilson-Dependent method was the method of choice as it produced confidence intervals for every sample and with an average coverage of 95.2%. Further, for almost every run size the Wilson-Dependent method produced notably tighter confidence intervals.

Finally, the Symbreg domain, with its non-extreme cumulative probability of success ($P(12) = 0.593$), levelled the playing field for the Normal method. The Normal method produced very good average coverage of 95.3% for an average of 99.5% of the samples. The Wilson-Dependent method did only slightly better in this instance, although the widths of its confidence intervals were a little tighter.

The Resampling method did very poorly over lower (25–100) run counts for the parity problem (coverages of 32%–78%). This was due to the low probability that a sample of the population would contain a run that did *not* find a solution before the minimum generation. For data where the cumulative success rate is very high at the minimum generation, it can now be seen that the resampling method is inappropriate to use.

3.1.5 When Minimum Generation is Unknown

Changes to the Methods

Unfortunately, is it extremely unlikely that a researcher will know the number of generations at which the true minimum computational effort occurs. This section discusses how confidence intervals can be established using an estimate of the true minimum generation.

For a sample of genetic programming runs, the minimum generation can be estimated by using the technique Koza described. That is, by calculating the computational effort, $I(i)$, for every generation, i, from 0 to the maximum in the

1. Obtain n independent runs using a population of size M. Obtain the observed success proportion, p, and observed minimum generation, j, using all n runs.
2. Obtain the $1-\alpha$ confidence limits for the true success proportion using Wilson's method (equations 2.3 and 2.4). Label these upper and lower limits p_u and p_l.
3. Approximate $1-\alpha$ confidence limits for the true minimum computational effort are given by
$$E_l = (j+1) \cdot M \cdot R(p_u, z)$$
and
$$E_u = (j+1) \cdot M \cdot R(p_l, z)$$
where E_l is the approximate lower limit and E_u is the approximate upper limit.

Table 3.4: Algorithm for the Wilson-Dependent method.

experiment. The estimated minimum generation is the generation where $I(i)$ is minimal.

For the generation of confidence intervals, the estimated minimum generation is used in place of the true minimum generation, but otherwise the three methods remain unchanged. Table 3.4 describes the Wilson-Dependent algorithm.

From a statistical perspective this introduces dependence between the measurements of minimum generation and the minimum computational effort. Keijzer et al. suggested that the runs in a GP experiment could be divided into two halves; the first half used to estimate the minimum generation and the second half used to estimate the minimum computational effort. However the cost of a GP run is typically so expensive that using only half the runs to establish computational effort is not seriously considered. This work follows that pragmatic approach and accepts the dependence (although we re-consider this in section 3.2).

Because no effort has been made to account for the increased variability in the estimated computational effort that is due to estimating the minimum generation, it should be expected that the confidence intervals produced using these methods would achieve less than 95% coverage.

Results and Discussion

For each problem domain and confidence interval generation method, table 3.5 gives the average coverage and the average number of valid confidence intervals

Method \ Problem	Ant	Parity	Symbreg	Multiplexor	Average
Normal	96.1%	63.8%	94.8%	93.1%	86.9%
	7,012	1,892	9,839	3,684	5,606
Wilson-Dependent	92.9%	94.0%	94.9%	95.7%	94.4%
	9,950	10,000	10,000	10,000	9,988
Resampling	92.4%	65.3%	91.2%	72.3%	80.3%
	10,000	10,000	10,000	10,000	10,000

Table 3.5: Average coverage percentages and average validity statistics by problem domain when the minimum generation is estimated. Averages are over 25–500 runs.

Method \ Runs	25	50	75	100	200	500	Average
Normal	65.1%	72.3%	94.7%	97.3%	96.7%	95.4%	86.9%
	2,497	3,770	4,695	5,595	7,212	9,870	5,606
Wilson-Dependent	93.0%	94.4%	94.7%	93.8%	94.9%	95.3%	94.4%
	9,928	9,998	10,000	10,000	10,000	10,000	9,988
Resampling	62.2%	73.9%	80.2%	84.5%	89.0%	91.9%	80.3%
	10,000	10,000	10,000	10,000	10,000	10,000	10,000

Table 3.6: Average coverage percentages and average validity statistics by run size when the minimum generation is estimated. Averages are over the four problem domains.

that were produced. Table 3.6 gives the same statistics but by run size and method.

Figure 3.1 depicts box and whisker plots of the width of the confidence intervals produced using each of the three methods for each of the six run sizes on the Ant domain.[6] The grey line across each plot indicates the value of the best estimate of the true computational effort. This line is added to assist understanding of the magnitude of the widths. The whiskers (indicated by the dashed line) in the plots extend to the most extreme data point or 1.5 times the interquartile range from the box, whichever is smaller. In the latter case, points past the whiskers are considered outliers and are marked with a small circle. The box-plot for 25 runs using the Resampling method is incomplete as more than 50% of the simulated experiments produced infinite confidence interval widths.

Surprisingly, the use of an estimated minimum generation had very little negative impact on the coverage of the three methods. Excluding the Parity

[6] Chapter 5 contains further discussion on confidence interval widths.

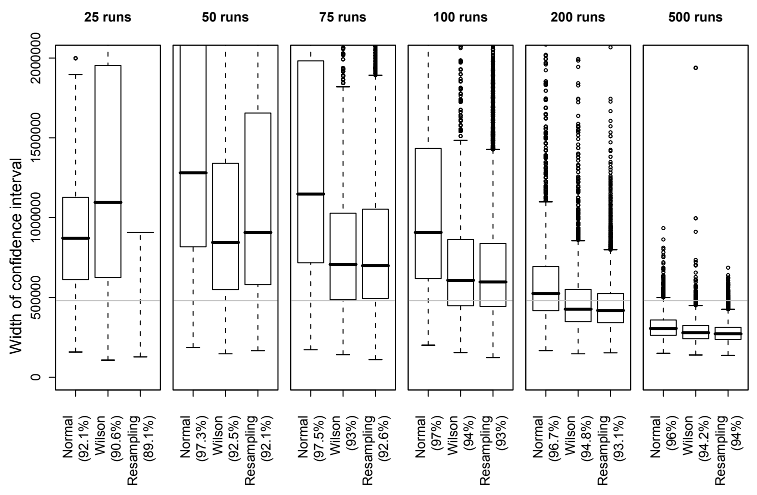


Figure 3.1: Confidence interval widths for the Ant problem domain when the minimum generation is estimated. Percentages indicate coverage for the specific configurations.

domain, the Normal method did well with an average coverage of 94.7% (as against the intended 95%). The Wilson-Dependent method did even better as, over *all* problem domains and run sizes, it dropped only slightly to an average of 94.4% (as compared to 95.2% when the true minimum generation was known). From these results it appears that, even when an estimated minimum generation is used, the confidence intervals produced by the Wilson-Dependent method are a good approximation to a 95% confidence interval.

It is hypothesised that the use of an estimated minimum generation had so little negative effect because the computational effort performance curves flatten out around the true minimum generation, and that the use of an estimate provides a result "good enough" for the production of a confidence interval.

Finally, it is worth noting that the median widths of the confidence intervals are almost always greater than the best estimate of the true value.

3.1.6 Further Analysis of the Wilson-Dependent Method

It is easy to retrospectively apply the Wilson-Dependent method to previously published results, however, because it is common for published work to fail to give the cumulative success proportion or the minimum generation at which the computational effort was calculated, the Wilson-Dependent method is not always able to be applied. We can instead consider a "best-case" confidence interval, one that gives the smallest range of computational effort given the number of runs executed. In this way we can say that a 95% confidence interval is *at least* this range.

To calculate the "best-case" for the lower bound, take the minimum defined value for each run size across the range of possible values for cumulative probability of success. More formally,

$$\min_p \frac{R(upper(p,n)) - R(p)}{R(p)} \tag{3.4}$$

where p ranges from 0 to 1; *upper* is the upper bound of a 95% confidence interval of a proportion (see equation 2.3); n is the number of runs; and

$$R(p,z) = \begin{cases} \frac{\log(1-z)}{\log(1-p)} & \text{if} \quad p < z \\ \textit{undefined} & \text{if} \quad p \geq z \end{cases} \tag{3.5}$$

The "best-case" for the upper bound is calculated in a similar way, but with *upper* replaced with *lower*, the lower bound in equation 2.4. The "best-case" scenario is only valid if $upper(P(i),n)) < z$.

This approach can be used if a computational effort value, E, has been stated for a specified number of runs, say 50, but without a value for the cumulative probability of success. In this case we can use the above formulae to calculate a "best-case" confidence interval for $\mathcal{E}$, the true minimum computation effort, as $(1-0.26)E \leq \mathcal{E} \leq (1+0.45)E$. The true 95% confidence interval will be *at least* this size.

3.1.7 Summary

The Wilson-Dependent method is an appropriate way to produce confidence intervals for Koza's computational effort statistic. From the empirical results, the use of an estimated minimum generation has little effect on the coverage and the intervals can be treated as a very good approximation to 95% confidence intervals.

The Wilson-Dependent method can often be retrospectively applied to earlier work as only the number of runs and the success proportion at the generation of minimum computational effort are required. If the number of runs is known but the success proportion is not known, then a minimum confidence interval can be generated using the "best-case" approach.

Finally, computational effort may not be the best measurement for comparison as this study has shown that results that differ by 50% or 100% may, from a statistical perspective, not be significantly different.

3.2 Reliability of Confidence Intervals

We have concluded that the Wilson-Dependent method was the best choice for the production of confidence intervals for Koza's minimum computational effort: it is fairly easy to calculate, practically always produces a valid confidence interval, and the confidence intervals included the true computational effort an appropriate proportion of the time.

However we left a few questions unanswered. This section attempts to answer some of those open questions: How does the dependence between success proportion and minimum generation affect performance? How does the method perform with different levels of confidence? Is the performance reliable for difficult problems which run for many generations?

3.2.1 Dependence Issues

In section 3.1.5 we assumed that GP practitioners would elect to use all their run-data to produce estimates of both the minimum generation (j) and the cumulative probability of success at the estimated minimum generation ($P(j)$). This is in contrast to the method proposed by Keijzer et al. [69] who used half their runs to estimate the minimum generation and the other half to estimate the cumulative probability of success. The advantage of using their method is that the estimation of the two variables is statistically independent. This section analyses these two variations on both the Wilson-Dependent and Resampling methods.

The Wilson-Dependent method has been defined in table 3.4. If we accept dependence between the estimation of $P(j)$ and j, then j is the generation at which the minimum computational effort occurs when calculated using the entire set of runs. $P(j)$ is estimated by the proportion of runs (as calculated over the entire dataset) which found a solution at or before generation j. This is the method that was used throughout section 3.1.

If independence between the estimation of $P(j)$ and j is desired, then the dataset should be divided in two. For this work we considered a division of 1:1 (as Keijzer et al. proposed), but other ratios could be used. j is estimated as the generation at which the minimum computational effort occurs when calculated using the first part of the dataset. $P(j)$ is estimated by the proportion of runs (as calculated over the second part of the dataset) which found a solution at or before generation j. We will term this method *Wilson-Independent*.

The resampling method (see section 3.1.3) where the estimation of j and $P(j)$ were dependent (*Resampling-Dependent*) was shown to be an inferior choice to

1. Obtain n independent runs. Label these as the source set.
2. Divide the source set into two parts (S_1 and S_2) with n_1 runs in S_1 and n_2 runs in S_2.
3. Repeat 10,000 times:
   (a) Select, with replacement, n_1 runs from the set S_1 (s_1).
   (b) Calculate j, the generation at which the minimum computational effort occurs, for the selection s_1.
   (c) Select, with replacement, n_2 runs from the set S_2 (s_2).
   (d) Calculate the computational effort at generation j for the selection s_2. If zero runs succeeded, the computational effort is infinite.
4. Find the 2.5% and 97.5% quantiles of the 10,000 minimum computational effort statistics. These provide an upper and lower range on a 95% confidence interval for the true minimum computational effort.

Table 3.7: Algorithm for the Resampling-Independent method.

Wilson-Dependent. Resampling-Dependent will not be considered any further in this chapter.

The *Resampling-Independent* algorithm is defined in table 3.7.

To compare the three methods, Resampling-Independent, Wilson-Dependent, and Wilson-Independent, the methods were applied to the same large datasets that were used previously: Ant, Parity, Symbreg, and Multiplexor (see page 29).

For each dataset six simulated run sizes were used: 25, 50, 75, 100, 200, and 500 runs.

For each of the 72 combinations of confidence interval method, dataset, and simulated run size, 10,000 samples of the specified number of runs were randomly selected from the problem domain's large dataset. For each sample the 95% confidence interval for the sample's computational effort was calculated using the specified method. Whether the confidence interval included the best estimate of the true computational effort (the *coverage*) was recorded, as was the width of the confidence interval relative to the best estimate of the true computational effort.

This process effectively simulates 10,000 genetic programming experiments over each of the four problem domains and each of the six run sizes, or a total of

240,000 experiments, for each of the three confidence interval methods.

The minimum computational effort, as calculated over the entire dataset, was used as the best estimate of the true computational effort for that dataset.

Figure 3.2 plots the results of these experiments. The upper graph gives the observed coverage by run size (averaged over the four problem domains). The lower graph gives the median observed confidence interval width as a ratio of the best estimate of the true computational effort (averaged over the four problem domains). Because Resampling-Independent produced an infinite-width median coverage for all bar one of the Ant experiments, only one data point for that method is plotted on the lower graph. 95% confidence intervals for the coverage results (upper graph) are smaller than ± 0.4 percentage points.

Of the three methods considered, Wilson-Dependent produces observed average coverage levels that are closest to the target 95%. This alone would make it the preferred choice, however it has two other advantages. The first advantage is that the width of the confidence intervals are notably tighter; and this is most obvious at the lower run sizes—run sizes that are most commonly used. If the data is averaged over problem domain, then Resampling's infinite-width issue is confined to just the Ant data, and in this form the Resampling method produced widths that were at least 67% larger than those produced with Wilson-Dependent. Wilson-Independent produced widths that were at least 51% larger than Wilson-Dependent. The second advantage is that confidence intervals produced by the Wilson method were significantly less computationally expensive to obtain than those produced via the Resampling method. We conclude that Wilson-Dependent is the method of choice, and is the method used throughout the rest of this chapter.

Finally, it appears that the estimated value of j and the estimated value of $R(P(j))$ may be correlated. This correlation may explain the enhanced coverage accuracy of Wilson-Dependent versus Wilson-Independent, but this remains an open issue.

3.2.2 Varying Alpha Values

In section 3.1, the target coverage level was set at 95%. This section extends our earlier work by considering the effect on observed coverage levels when the target coverage level is varied.

Six commonly used target coverage levels (also known as $1 - \alpha$ values) were selected. They were: 80%, 85%, 90%, 95%, 99% and 99.9%. The previous

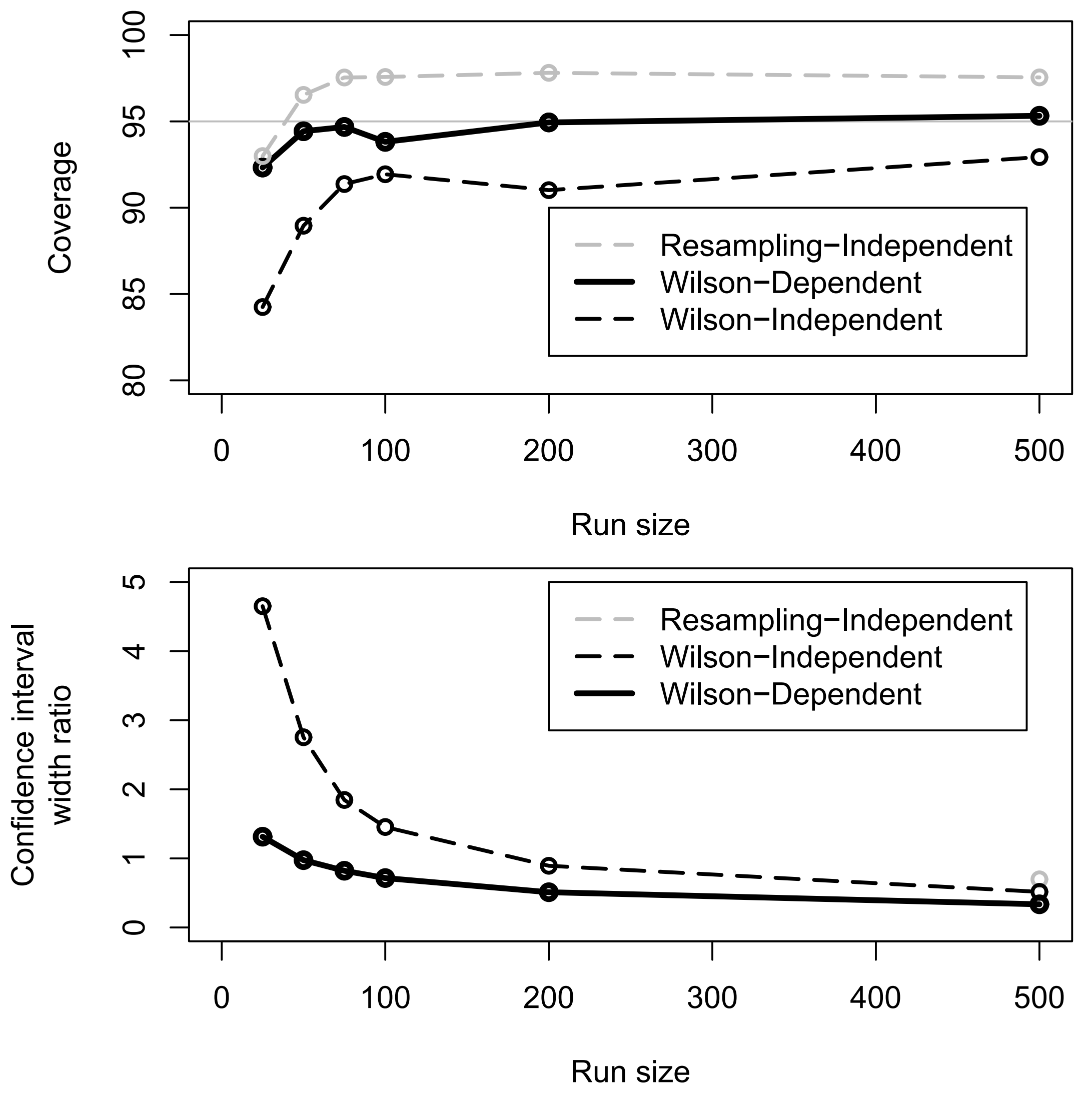


Figure 3.2: Average observed coverage (upper) and median confidence interval width ratios (lower) against run size for the three methods, averaged over the four problem domains. Missing Resampling-Independent data caused by infinite-width median coverage.

Target coverage	80%	85%	90%	95%	99%	99.9%
Ant	77.5%	82.7%	87.6%	92.4%	97.5%	99.1%
Multiplexor	79.2%	84.6%	90.7%	95.7%	97.9%	98.2%
Parity	85.2%	89.7%	91.9%	94.0%	95.1%	95.5%
Symbreg	79.1%	84.7%	90.0%	94.9%	99.0%	99.9%

Table 3.8: Coverage statistics by target coverage and problem domain

experimental setup was repeated for each target level, that is: four problem domains were used (Ant, Parity, Symbreg, and Multiplexor) and six runs sizes were used (25, 50, 75, 100, 200, and 500 runs).

For each of the 144 combinations of target coverage level, problem domain, and run size, 10,000 samples of the specified number of runs were randomly selected from the problem domain's large dataset. For each sample the confidence interval for the sample's computational effort was calculated using the Wilson-Dependent method (with a $1 - \alpha$ value as specified by the target coverage level). Whether the confidence interval included the true computational effort (the coverage) was recorded.

The cost of increasing the coverage level is an increase in the width of the confidence intervals. To assess the impact on the confidence interval width, we also recorded the width of the interval as a ratio of the true computational effort, for every sample's confidence interval.

Figure 3.3 plots the mean observed coverage and the ratio of the confidence interval width for each target coverage level (averaged over the four problem domains and six run sizes). Tables 3.8 and 3.9 give coverage and width statistics by problem domain and target coverage level (averaged over the six run sizes). Results where the sample did not include any successful runs could not produce a valid confidence interval; such samples were ignored for the calculation of the averages.

As can be seen from the upper plot of figure 3.3, the observed coverage levels are very close to the target levels up to about 95%. For the two higher cases of 99% and 99.9% the observed coverage is slightly smaller than the target. Although these results are highly statistically significant (with 95% confidence intervals of less than ± 0.2 percentage points), one could ask if the four datasets that were selected are a fair representation of the problem domains on which this method may be used. This question will always remain open, even though the domains selected are common GP problems. However, at the very least

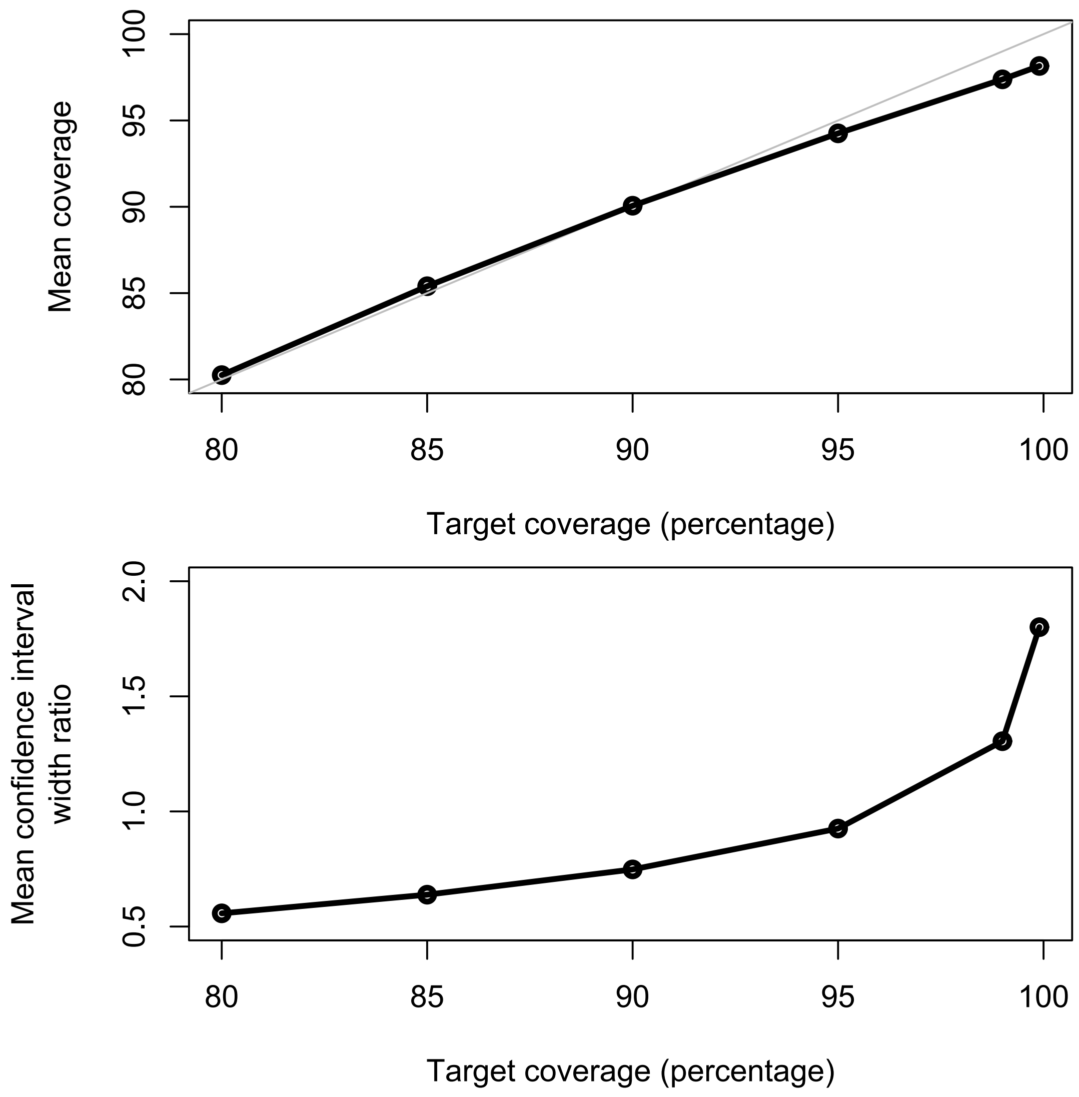


Figure 3.3: Observed coverage (upper) and confidence interval width ratios (lower) of the Wilson-Dependent method against target coverage, averaged over all four problem domains and all six run sizes.

Target coverage	80%	85%	90%	95%	99%	99.9%
Ant	1.12	1.28	1.51	1.89	2.71	3.80
Multiplexor	0.37	0.42	0.49	0.60	0.82	1.12
Parity	0.37	0.43	0.51	0.63	0.90	1.24
Symbreg	0.38	0.42	0.49	0.59	0.79	1.04

Table 3.9: Confidence interval width ratios by target coverage and problem domain

the results from the four problem domains do give a good indication that the Wilson-Dependent method performs as expected with different target coverage levels between 80% and 95%.

The lower plot of figure 3.3 shows, as expected, that as the target coverage is decreased, the width of the confidence interval also decreases. However even with a choice of 80% coverage, the mean observed width was not small: it was 56% of the true computational effort (with a range of 16% to just over two-fold). The Ant dataset has larger width ratios than the other datasets (due primarily to its low $P(j)$ value). But even with the Ant dataset removed, 80% coverage still gives a width ratio of 37%.

3.2.3 Large Minimum Generations

In section 3.1 the confidence interval generation methods were tested on a range of minimum generations that was quite tight: 12 to 25 generations. It was an open question as to whether any of the methods would continue to function acceptably if the true minimum generation were significantly outside this range.

For "difficult problems", Luke [83] concluded that a solution was more likely to be found with longer runs than with multiple shorter runs. This would produce a minimum generation that was much larger than would be obtained were the traditional approach, of terminating runs longer than 50 generations, taken.

To ensure that the Wilson-Dependent method does not deteriorate with larger minimum generations, it is important to empirically check its validity in this area. Unfortunately it is far too computationally expensive to obtain thousands of runs on problem domains that find solutions only after many hundreds of generations. Instead we have elected to simulate GP experiments of that difficulty and to check the Wilson-Dependent method using this simulated data.

Simulating GP Experiments

The distribution which models the generation at which a GP solution will appear is not known [21, page 158]. In fact, it is highly likely that the distribution is problem-domain specific. However from the four large datasets that we have studied, it can be said that the distribution is a smooth curve that peaks at a specific generation and that may have a long tail to the right.

Many such distributions have been defined that pass this description. In order to assess the match between these distributions and the large datasets, we optimised the distributions' parameters and then tested their quality as models.

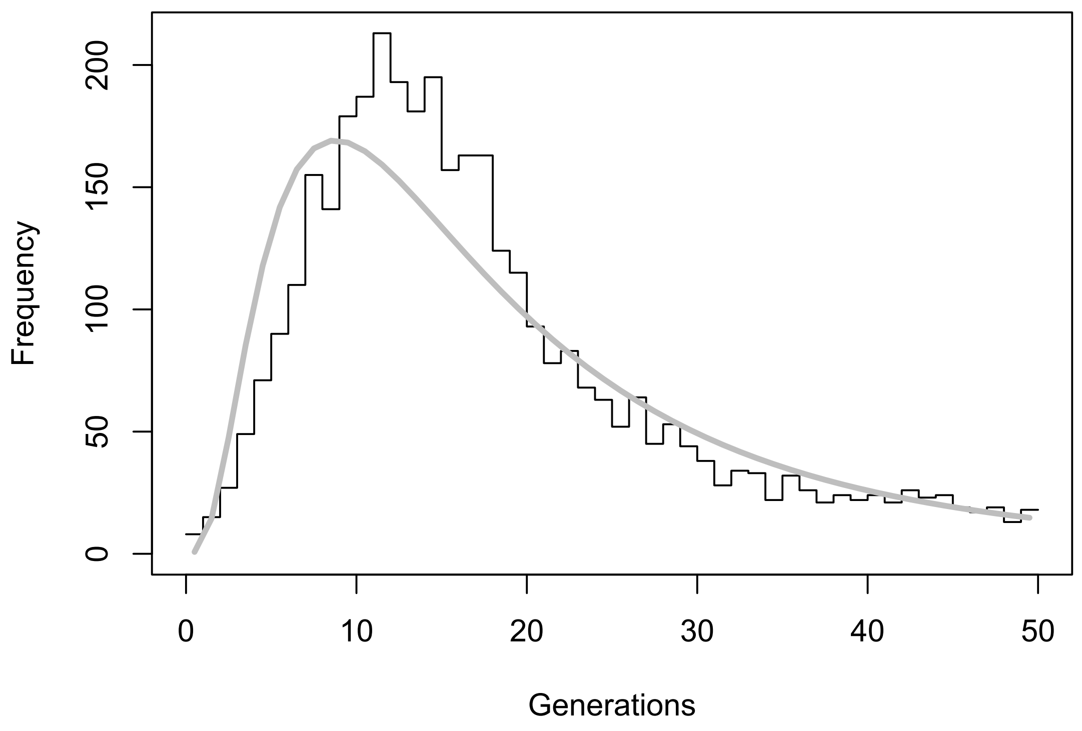


Figure 3.4: The best model (thick grey line) that was found for the Ant dataset (black line): a log-normal distribution with $\mu = 16.2$ and $\sigma = 2.2$.

Distribution Selection The distributions we considered were: Normal, Log-Normal, Gamma, Weibull, and Beta. Each distribution takes two parameters to describe its shape. The optimal values for these parameters were found with a numerical method[7]. Figure 3.4 shows one of the better models. Once the distributions' parameters were obtained, the models were then compared to the real dataset using χ^2 tests.

To quantify how well the distribution modelled the real data, χ^2 tests were executed for multiple run sizes (25, 50, 100, 200, 300, 400, and 500 runs). For each run size, 100 samples of that number of runs were randomly generated using the distribution (with its optimised parameters). For each of the 100 samples a χ^2 test was executed (on bin sizes of one generation, unless the real dataset contained fewer than 5 successes in a given bin, in which case adjacent bins were combined until the enlarged bin contained at least 5 successes).

None of the models performed acceptably. Of the five models, the best were

[7]An implementation of the Nelder and Mead method was used as defined by the optim function in the statistical software R [99].

normal and log-normal, but even for those, almost all the samples with more than 200 runs were classed by the χ^2 test as significantly different to the data they were modelling.

Although we were unable to find an acceptable model we elected to use the best two distributions (normal and log-normal) for simulating GP runs with large minimum generations. We limited the experiments to simulate no more than 100 runs.

Probability of Success When considering the cumulative probability curves of the large datasets (most notably Symbreg and Ant), it appeared that GP runs may asymptotically tend to a cumulative probability of success that is less than one. Although it seems reasonable that if a GP run was left to evolve indefinitely it would eventually find a solution, it would seem that the tails must become *very* long indeed [21, page 82].

To model this asymptotic behaviour, a "second level" was added. This asked the question, "does a given run have any chance of success?". If the answer was "yes", then the chosen distribution was used to answer the question "does success occur before the cut-off generation?".

Testing Performance on Simulated Data

Because neither the normal nor the log-normal distributions were shown to be sufficiently similar to the data after 200 runs, we elected to limit the use of the models to 25, 50, and 100 runs.

Because appropriate parameters for each model are unknown a range of parameters were used. For both models the mean was set to 25, 100, 500, and 1000 generations. For the log-normal distribution the standard deviation was set to 0.5, 1.0, and 2.0 times the mean. For the normal distribution the standard deviation was set to $\frac{1}{16}$, $\frac{1}{8}$, and $\frac{1}{4}$ times the mean. The probability of success (at the second level) was set to 0.2, 0.5, and 0.8. The cut-off was set to 1,000 generations. The number 500 was used as a population size, but this was just a scaling factor that had no bearing on the model nor the coverage results.

It was found that when the standard deviation values used for the log-normal distribution were applied to the normal distribution, they were sufficiently large to produce a non-zero probability of success at the initial generation. This non-zero probability was sufficient to set the minimum generation to generation zero. Koza studied the probability of finding a solution at generation zero for both the 11-multiplexor and 6-multiplexor problems [71, page 207]. He tested up to

Mean	25	100	500	1000
	93.6%	91.5%	90.6%	91.1%
Std. dev.	0.5	1.0	2.0	
	93.6%	91.7%	89.9%	
Success prop.	0.2	0.5	0.8	
	90.1%	92.2%	92.7%	
Runs	25	50	100	
	90.2%	92.0%	93.0%	

Table 3.10: Coverage statistics for the log-normal model

10,000,000 individuals, and found that none were successful; that is a probability of success of less than 0.000001. Because the work in this section was intended to model problems significantly harder than the two Koza studied, the standard deviations used for the normal distribution were reduced.

For each of the 216 combinations of distribution, mean, standard deviation, probability of success, and number of runs, one million simulated runs were generated using the specified distribution. From these simulated runs, 10,000 samples of the specified number of runs were randomly selected. For each sample the 95% confidence interval for the sample's computational effort was calculated using the Wilson-Dependent method. Whether the confidence interval included the true computational effort (the coverage) was recorded.

The true computational effort was obtained by calculating

$$\min_i \, (i+1) \cdot R(P(i) \cdot p) \cdot M \tag{3.6}$$

where: R is the function for calculating the number of runs required (equation 2.7); $P(i)$ is the cumulative proportion given by the distribution function; p is the probability of success (at the second level); i is the generation which ranged from 0 to 1000; and M is the population size (where 500 was used).

Tables 3.10 and 3.11 summarise these results. The log-normal model had an average coverage of 91.7% and the normal model had an average of 94.6%. These should be compared with the desired coverage of 95%.

So, for the log-normal model (table 3.10), the average coverage for a mean of 25 generations (as seen in the top left cell) was 93.6%. This coverage is an average coverage for all parameter combinations where the mean was set to 25. Similarly,

Mean	25	100	500	1000
	94.8%	94.7%	94.8%	94.4%
Std. dev.	$\frac{1}{16}$	$\frac{1}{8}$	$\frac{1}{4}$	
	94.9%	94.8%	94.3%	
Success prop.	0.2	0.5	0.8	
	94.1%	94.9%	94.9%	
Runs	25	50	100	
	94.4%	94.9%	94.7%	

Table 3.11: Coverage statistics for the normal model

for the normal model (table 3.11), for all parameter combinations where run size was set to 100 runs, an average of 94.8% of the samples produced a confidence interval that included the true computational effort.

For the log-normal distribution, the minimum generation ranged from 7 to 1000 generations with a mean of 299 and an upper quartile of 523 generations. For the normal distribution the minimum generation ranged from 28 to 1000 generations with a mean of 447 and an upper quartile of 799 generations. Thus the minimum generations that were considered in this study were significantly larger than those observed in the experiments originally executed.

Both models showed reduced coverage as the standard deviation increased. Both models produced increased coverage levels as the success proportion increased, and mostly increased coverage as the number of runs increased.

From these results, if your GP data follows a normal or log-normal distribution, it appears that the confidence interval generation method based on the Wilson-Dependent method produces coverage levels that are a good approximation to a 95% confidence interval.

Arbitrary Distributions

Given the success with the normal and log-normal distributions, we continued the investigation with some arbitrarily selected distributions. The objective in this study was to see if the similar success could be obtained with the alternative distributions.

Figure 3.5 shows the four new distributions that were selected. All four distributions had zero-values between 0–49 generations and 951–1000 generations and

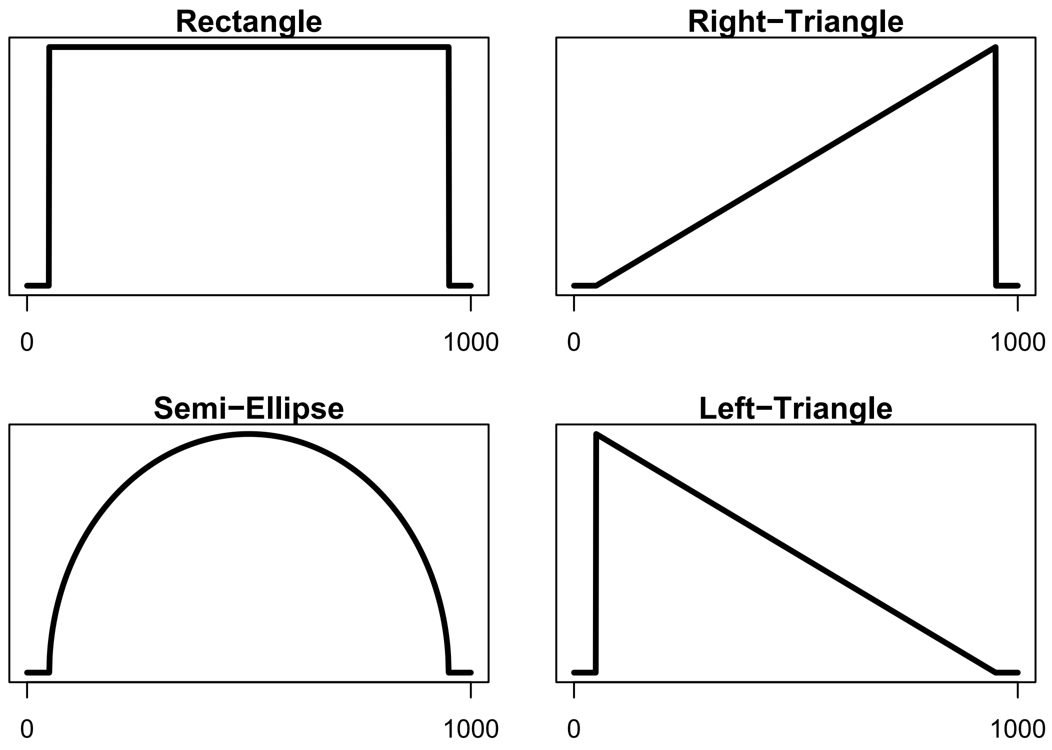


Figure 3.5: Density versus generation for the four arbitrarily-selected distributions.

had a cumulative probability (the area under the graph) of one. The four distributions were: a rectangular shape; a triangular shaped distribution that sloped from 0 at 50 generations to a peak at 950 generations (termed Right-Triangle); a triangular shape that sloped from a peak at 50 generations down to 0 at 950 generations (termed Left-Triangle); and a semi-ellipse.

To test the coverage of the Wilson-Dependent method on these distributions, three variables were required: the distribution, the success proportion (at the second level), and the number of runs. The success proportion was given the same three values as before: 0.2, 0.5, and 0.8. However, the number of runs in each sample was extended to include 25, 50, 100, 200, and 500 runs.

For each of the 60 combinations of distribution, success proportion, and runs size, 100,000 simulated runs were generated using the specified distribution. From these simulated runs, 1,000 samples of the specified number of runs were randomly selected. For each sample the 95% confidence interval for the sample's computational effort was calculated using the Wilson-Dependent method. Whether the confidence interval included the true computational effort (the coverage) was recorded.

Success prop.	0.2	0.5	0.8		
	92.7%	95.1%	95.3%		
Runs	25	50	100	200	500
	93.2%	94.4%	94.5%	94.9%	94.8%

Table 3.12: Coverage statistics for the Rectangle model

Success prop.	0.2	0.5	0.8		
	94.9%	95.3%	94.6%		
Runs	25	50	100	200	500
	95.3%	95.3%	94.0%	95.1%	95.0%

Table 3.13: Coverage statistics for the Right-Triangle model

The minimum generation was 951 for Rectangle and Right-Triangle. Left-Triangle's minimum generation ranged from 341 to 692 generations and Semi-Ellipse's ranged from 817 to 928 generations—with the specific value dependent on the success proportion.

Tables 3.12, 3.13, 3.14 and 3.15 show the coverage results for the four distributions. The Wilson-Dependent method produced an average coverage of 94.4% for Rectangle, 94.9% for Right-Triangle, 91.1% for Left-Triangle, and 94.0% for Semi-Ellipse.

Although the average coverage for Left-Triangle dips to 91.1%, this should be compared to Robert Newcombe's analysis of the performance of the normal-approximation method for confidence intervals for a proportion [91]. He showed that method to have an estimated mean coverage of 88%. In that light, the Wilson-Dependent method on the Left-Triangle data performs better than that generally-accepted and widely used method.

These results are very interesting. They show that the performance of the Wilson-Dependent method is not affected by the distribution of successes—even with these four far-from-typical distributions. The results also give further evidence that the method is not sensitive to the magnitude of the minimum generation.

Success prop.	0.2	0.5	0.8		
	89.7%	90.7%	92.8%		
Runs	25	50	100	200	500
	88.1%	89.9%	91.3%	92.5%	93.7%

Table 3.14: Coverage statistics for the Left-Triangle model

Success prop.	0.2	0.5	0.8		
	92.5%	94.0%	95.4%		
Runs	25	50	100	200	500
	93.7%	93.4%	93.4%	94.3%	95.1%

Table 3.15: Coverage statistics for the Semi-Ellipse model

3.2.4 More Large Datasets

Steffen Christensen executed some enormously large number of runs on the Ant problem domain [21]. This section analyses the performance of the confidence intervals for these real datasets.

The four datasets were all based on the artificial ant on the Santa-Fe trail, a problem domain detailed in *Genetic Programming* [71, section 3.3.2] and commonly used as a benchmark for variations of GP. The four datasets that Christensen produced vary by population size and the generation at which the runs were cut off. The four datasets are:

- *Ant m10000g25*: Panmictic population of 10,000; cutoff of 25 generations; 12,280 runs; best estimate of the true computational effort 478,506 at generation 15; $P(15) = \frac{9{,}647}{12{,}280} = 0.786$
- *Ant m1000g150*: Panmictic population of 1,000; cutoff of 150 generations; 40,010 runs; best estimate of the true computational effort 446,801 at generation 17; $P(17) = \frac{6{,}775}{40{,}010} = 0.169$
- *Ant m250g60*: Panmictic population of 250; cutoff of 60 generations; 400,625 runs; best estimate of the true computational effort 488,518 at generation 19; $P(19) = \frac{18{,}445}{400{,}625} = 0.0460$
- *Ant m250g1000*: Panmictic population of 250; cutoff of 1,000 generations; 8,000 runs; best estimate of the true computational effort 503,594 at generation 20; $P(20) = \frac{355}{8{,}000} = 0.0444$

Run size	25	50	75	100	200	500	Average
m10000g25	95.9%	95.4%	95.3%	94.7%	95.0%	94.9%	95.2%
m1000g150	92.4%	94.4%	94.0%	94.8%	95.2%	94.7%	94.2%
m250g60	74.9%	89.4%	90.1%	92.4%	93.5%	93.8%	89.0%
m250g1000	78.4%	88.8%	91.8%	92.1%	93.3%	94.6%	89.8%
Average	85.4%	92.0%	92.8%	93.5%	94.3%	94.5%	92.1%

Table 3.16: Coverage statistics, by run size and dataset, for the four extra Ant datasets.

These datasets are interesting because, whereas the experiments in section 3.1 covered $P(j)$ values that were biased towards one, these datasets are biased towards a cumulative success (at the minimum generation) of zero. To enable direct comparison with the earlier results, run sizes were simulated at 25, 50, 75, 100, 200, and 500 runs.

For each of the 24 combinations of dataset and run size, 10,000 samples of the specified number of runs were randomly selected from the specified large dataset. For each sample the 95% confidence interval for the sample's computational effort was calculated using the Wilson-Dependent method. Whether the confidence interval included the best estimate of the true computational effort (the coverage) was recorded. The best estimate of the true computational effort was the computational effort calculated over the entire dataset.

Table 3.16 shows the results of these experiments. 95% confidence intervals for these results are at most ±1 percentage point, and in most cases will be no more than ±0.5 percentage points.

For the two datasets with the higher values for $P(j)$ (m10000g25 and m1000-g150), the performance of the Wilson-Dependent method is very good, averaging 95.2% and 94.2% coverage. For the other two datasets (m250g60 and m250g1000), the performance is good except for the smaller run sizes (specifically 25 runs).

However, for the smaller run sizes, it is worth noting that the granularity of the estimate of $P(j)$ is of the same order as the best estimate of the true value. In other words, for the case where 25 runs are being sampled, we should expect just one of the runs to succeed ($\frac{1}{25} = 0.04$ and the two values for $P(j)$ were 0.0460 and 0.0444). Thus, a variation in success of just one run is a variation of approximately ±100% of the true value.

It is also worth noting that, although the only difference between m250g60 and m250g1000 was that the former had a shorter cutoff value (a variable that would not have affected the true computational effort), there was still a 3.1%

difference between their estimated computational efforts—and that was with an enormous number of runs.

So, as might be expected, if the cumulative probability of success at the minimum generation is very small compared to the granularity produced by the number of runs, the coverage of the Wilson-Dependent method deteriorates.

However, the general picture provided by these datasets is that the method to produce confidence intervals for minimum computational effort is reliable.

3.2.5 Summary

This research has extended the work in section 3.1 on the production of confidence intervals. We have shown that the Wilson-Dependent confidence interval production method is reliable; specifically that:

- It out-performs other methods, in terms of both appropriate coverage levels and tighter confidence interval widths.
- It performs well at different target coverage levels, especially those between 80% and 95%.
- It performs well across a large range of minimum generations (10 to 1,000 generations) on simulated datasets.
- It appears to be insensitive to the distribution of generations of successful runs.
- It performs well on a number of datasets collected from real GP runs.

The method should be applicable to all genetic programming runs where the generations-to-success follows a normal, log-normal, or similar distribution—we even showed the Wilson-Dependent method was reliable under four very extreme distributions. We hypothesise that these cases cover all genetic programming experiments, but with the limitation that success is sufficiently common to make the confidence intervals useful.

3.3 Computational Effort near 100% Success

There is an interesting effect on minimum computational effort confidence intervals that are calculated with the Wilson-Dependent method from cumulative success rates close to 100%. In such situations the confidence interval can be biased above the calculated value: if the observed probability of success is greater than or equal to z[8] then the lower bound of the confidence interval will be the same as the calculated value. This section discusses this phenomenon and its effect on the coverage rates of the confidence intervals.

We have already studied a system where this issue would have had an impact: the Parity domain in section 3.1.5, where 99.6% of the 3,400 runs found a solution and the cumulative probability of success at the minimum generation was 0.985. In that case the coverage rate dropped to 94.0% (see table 3.5). Although that result indicated the reliability of the Wilson-Dependent method to generate appropriate confidence intervals, we go further here by considering a range of minimum generations and variations in the cumulative success proportion curve.

For this study we are interested in the case where the lower limit of the minimum computational effort confidence interval is the same as the calculated minimum computational effort. This can only occur when the upper limit of the confidence interval for cumulative probability of success is equal to the observed cumulative probability of success, and that can only happen when the observed cumulative probability of success is greater than or equal to z because at that point the function R is clamped to one (see equation 2.7 on page 18).

Two questions that will be answered in this section. The first, "what is the coverage like when the true success rate approaches 100%?", allows comparison with the earlier studies. The second, "what is the coverage like given an observed success rate greater than z?", is perhaps the more practical question and will give an indication of the reliability of confidence intervals that are affected by this issue.

3.3.1 Coverage with True Success Rate Near 100%

This section answers the question "what is the coverage like when the true success rate approaches 100%?". We will show that the method's performance is acceptable when compared with the performance of other confidence interval generating methods.

Method

For this question it would be preferable to use data from real genetic programming runs. Obtaining such data is, unfortunately, currently too demanding computa-

[8] The variable z is introduced in section 2.3.1 and, as is common, is set to 0.99 throughout this work.

tionally. Instead we have chosen to simulate GP data using the assumption that the success proportion curves are either normally or log-normally distributed—the two most accurate approximations that we found to real GP data (see section 3.2.3).

Normal For the normally-distributed experiments, we simulated the success probability curves using three different means (25, 100, 500 generations) and three different standard deviations of $\frac{1}{16}$, $\frac{1}{8}$, and $\frac{1}{4}$ times the mean. Each of these curves were multiplicatively scaled by a "success factor"—making success factor equivalent to the true cumulative success probability after an infinite number of generations. Nine success factors were used (90%, 95%, 96%, 97%, 98%, 99%, 99.5%, 99.9%, and 100%). Each of these 81 configurations were repeated for six different run sizes (25, 50, 75, 100, 200, and 500 runs), giving a total of 486 configurations.

For each configuration of mean, standard deviation, success factor, and run size, one million simulated runs were generated using a normal distribution. For success factors of 95%, 99%, 99.5%, 99.9%, and 100%, from these simulated runs, 10,000 samples of the specified number of runs were randomly selected (with replacement). For the other success factors 1,000 samples were obtained. For each sample a 95% confidence interval for the sample's computational effort was calculated using the Wilson-Dependent method. Whether the confidence interval included the true minimum computational effort (the *coverage*) was recorded.

The true computational effort was obtained by calculating

$$\min_i \, (i+1) \cdot R(P(i) \cdot p) \cdot M$$

where: R is the function for calculating the number of runs required (equation 2.7); $P(i)$ is the cumulative proportion given by the normal distribution function; p is the specified success factor; i is the generation which ranged from 0 to 1000; and M is the population size (effectively just a scaling factor; 500 was used).

Log-Normal For the log-normally distributed experiments, a very similar procedure was followed. The only differences were that:

- The distribution was log-normal.
- Ten success factors were considered (90%, 95%, 96%, 97%, 98%, 99% 99.25%, 99.5%, 99.75% and 100%).

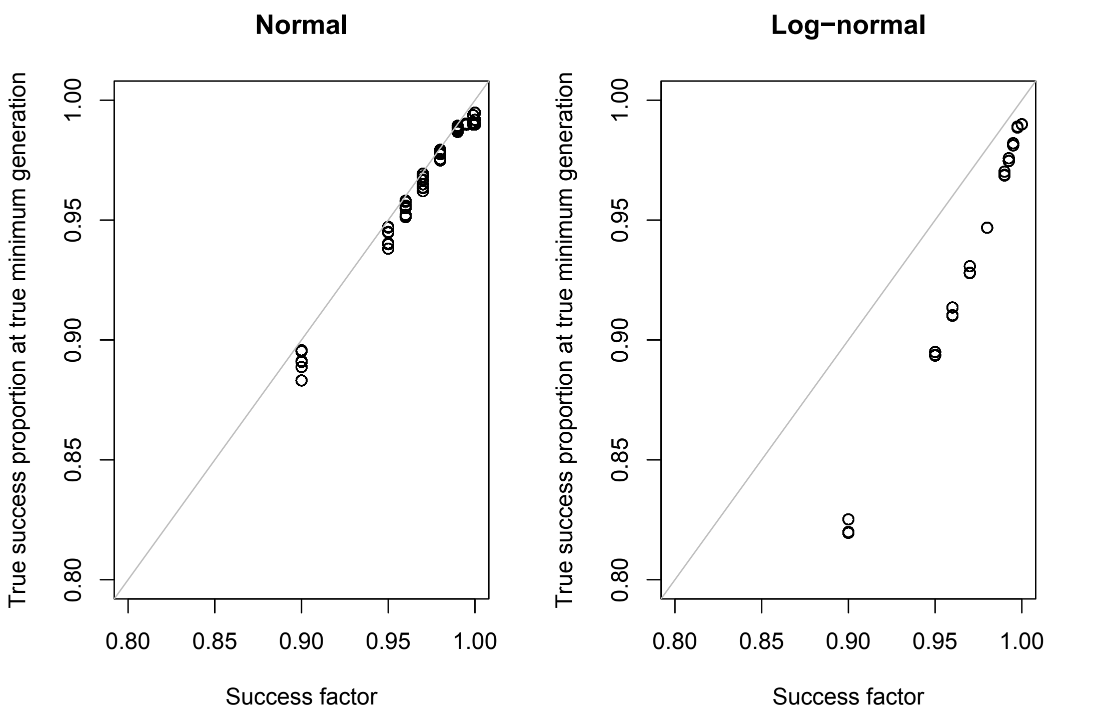


Figure 3.6: Success factor versus true success proportion at the true minimum generation for the Normal and Log-normal experiments.

- 1,000 samples were taken for each of the configurations.
- Only one standard deviation scaler was used ($\frac{1}{2}$).

Results and Discussion

It is of use to note the relationship between success factor and the true cumulative success at the true minimum generation. For the normally distributed experiments, the specified success factor is an acceptable indicator but for the log-normal experiments this approximation is less accurate. Figure 3.6 graphs these relationships. It should be noted that the true success proportion at the true minimum generation cannot be greater than the value for success factor.

Figure 3.7 plots the observed coverage rates for the different success factors for the normal and log-normal experiments. Both curves show good coverage rates near the desired 95% level up to a success effort of about 99% at which point they both drop markedly. However, when they drop, although they fall quickly, at worst the drop was to 78% when 95% was specified. Although this is far from ideal, this range of coverage is not unlike that observed by Newcombe [91]

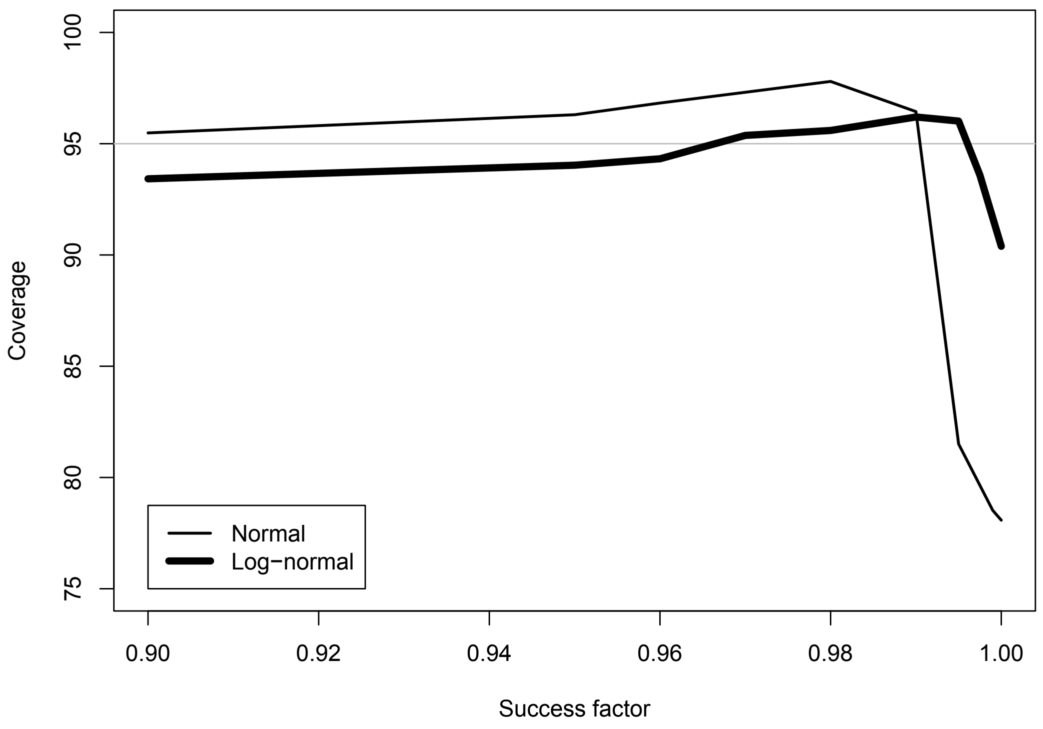


Figure 3.7: Observed coverage against success factor for the Normal and Log-normal experiments. Confidence intervals are less than ±1.5 percentage points for Normal and ±2.0 percentage points for Log-normal.

regarding Wilson's method (where the minimum observed coverage was 83%) and the "Likelihood based" method (where the minimum was 80%), and it is far superior to the frequently used normal-approximation method with a minimum observed coverage of 0.02%!

Further, it is not uncommon for confidence interval measures to be unreliable at extreme values. A commonly known example would be the rule-of-thumb for normal-approximation confidence intervals for proportions: one should satisfy the equations $np > 5$ and $n(1-p) > 5$ before the intervals are valid [24]. But for extreme values of p (close to either zero or one) the products can be very small, thus reducing the quality of the underlying assumptions which has a direct result on the quality of the confidence intervals.

An analysis by both run size and success factor shows that the drop at high success factors is greater for higher run sizes, thus for those with a high success factor, increasing the run size decreases the coverage rates. Figure 3.8 graphs this effect.

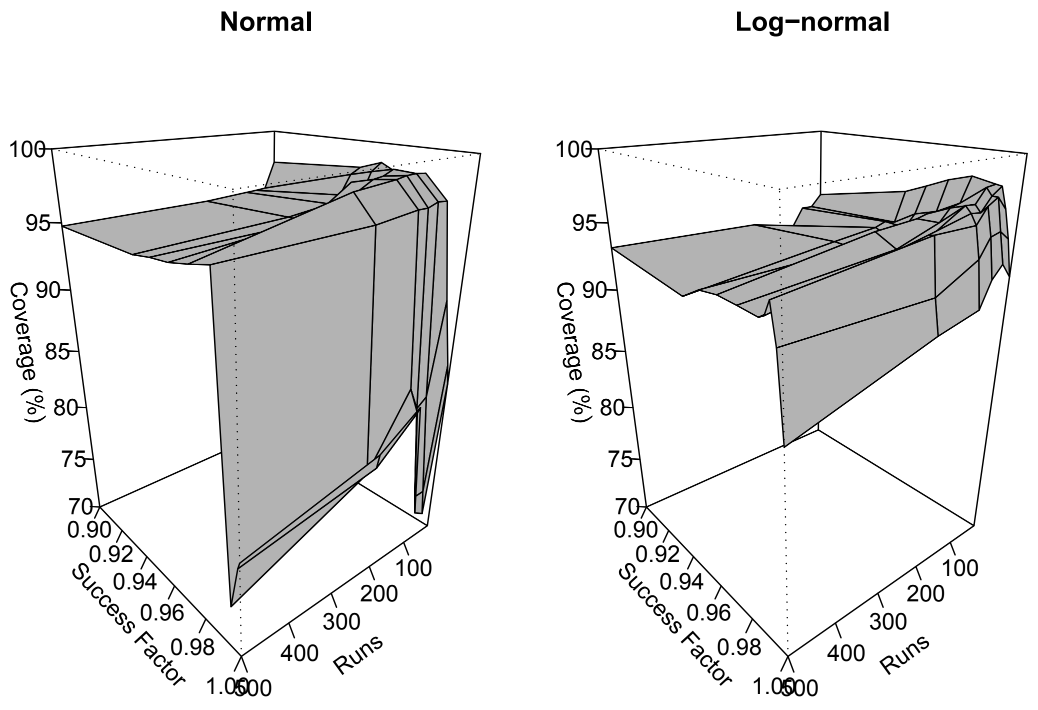


Figure 3.8: Observed coverage and run size against success factor for the Normal and Log-normal experiments.

The minimum generation ranged from 29 to 848 generations but because of the bias towards very high success proportions, the mean minimum generation was just 285.

An interesting effect occurs when the observed probability of success is greater than z and either a very large number of runs was executed or the confidence $(1 - \alpha)$ level is reduced. In some cases it is possible that the lower limit of the confidence interval for cumulative probability of success does not drop below z. When this occurs the function $R(p, z)$ is clamped to one for both the upper and lower limits, thus producing a zero width confidence interval for the minimum computational effort. Zero width intervals are an obvious indication of failure in any method to generate confidence intervals. Fortunately, this is an unlikely scenario. Of the experiments, only those at 500 runs demonstrated this issue, and even then only at very low rates: from 0.2% at a success factor of 99.5% up to 2% when the success factor was 100%.

3.3.2 Coverage with Observed Success Greater than z

This section answers the question "what is the coverage like given an observed cumulative success at the minimum generation that is greater than z?". It is perhaps a more useful question than that of the previous section as it is more pragmatic: it is highly unlikely that a practitioner will know the true success rate but instead highly likely that they will have an observation for which they are interested in the coverage rate they can expect.

Method

Ideally this study would consider a collection of experiments where the observed cumulative success was greater than or equal to z. Given their true computational effort, we could then calculate the coverage rates and analyse the experiments' configurations for trends. Unfortunately, this is not feasible. If we were to attempt to use real GP runs, then the true computational effort is not known and many many runs would have to be made to estimate it. But one configuration is not sufficient to assess the general coverage rate, so many different real GP runs would be required and many of those wouldn't produce the desired value of the observed probability of success thus resulting in an utterly unacceptably large computational requirement. The feasibility increases if we simulate GP data rather than use real runs, but we would still be left with the question of what configurations to simulate.

Instead, we will take a slightly different approach and consider a set of experiments that are both likely to produce data where the observed cumulative

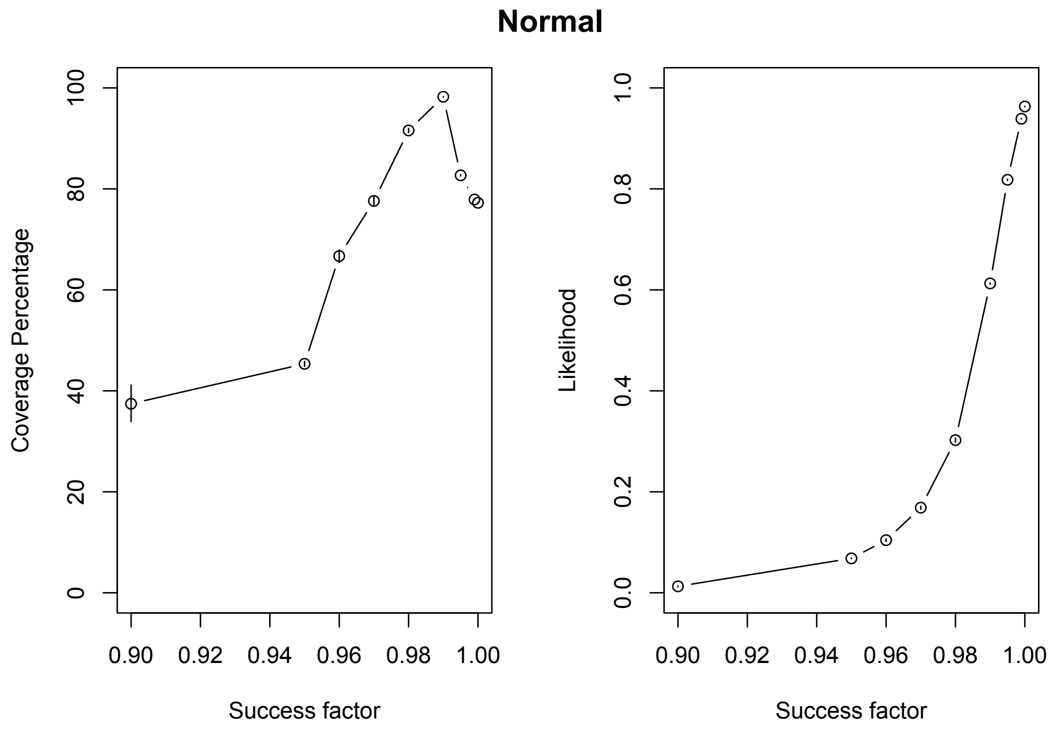


Figure 3.9: Success factor versus coverage percentage (left) and success factor versus likelihood (right) for the Normal experiments. 95% confidence intervals are shown for both graphs.

success is greater than or equal to z, and to be generally representative of real-world experiments where this effect would occur. Analysing the data will give us some indication for how the Wilson-Dependent method deteriorates in this extreme case.

For this we can re-use the experimental data from the previous section. From each of the samples obtained earlier we selected those with an observed success proportion that was larger than z at the observed minimum generation. The proportion of such cases was noted and will be discussed as the *likelihood* that the sample was affected. The *coverage* of the selected samples was also observed.

Results and Discussion

Figure 3.9 graphs the results for the normally-distributed experiments while figure 3.10 graphs the results for those log-normally distributed.

The left-hand graphs indicate the coverage rates that were observed. For both the Normal and Log-normal experiments, coverage was worst when the success

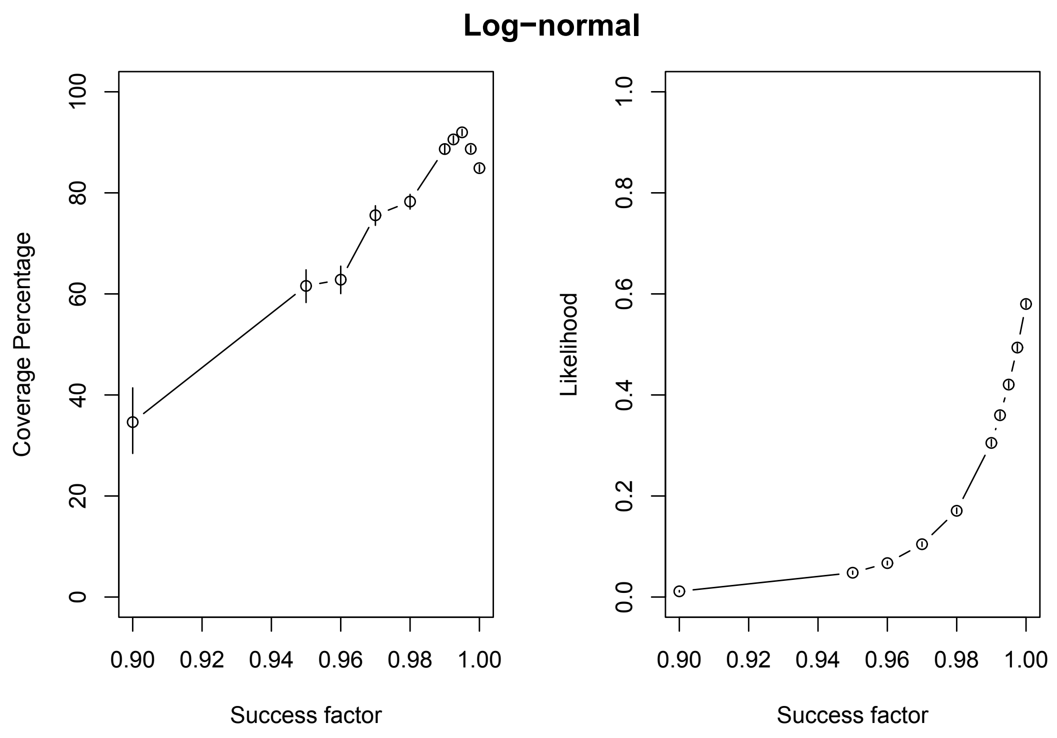


Figure 3.10: Success factor versus coverage percentage (left) and success factor versus likelihood (right) for the Log-normal experiments. 95% confidence intervals are shown for both graphs.

factor was at 90% and it peaked at a success factor of about 99%. It should be remembered that a target coverage of 95% was specified.

Thankfully for both experiments, the lowest coverage rates were correlated to the lowest likelihoods of observation (as shown in the right-hand graphs).

An average coverage can be calculated, weighting the observed values by their likelihood and by their associated widths. Doing this gives an average expected coverage of 82% for both Normal and Log-normal, although it ranges by run size (coverage generally decreased as the number of runs increased).

If the true minimum generation had been known and used then we could have expected that at a true success rate of 99.5%, 100% coverage would have been observed. This could be expected because the only variability would come from the measurement of the success rate, and Wilson's method on the two extreme possibilities (99% and 100%), for all simulated run sizes, produces confidence intervals that include 99.5%. However 100% coverage is not observed, instead the observed value plummeted to 83%. This difference is entirely explained by the variability associated with the measurement of the minimum generation.

3.4 Comparing Two Minimum Computational Efforts

In this section we first discuss a method to produce random numbers distributed according to the likelihood of the true minimum computational effort. Using that method we can then offer methods to find confidence intervals for two related measures: (i) the difference between, and (ii) the ratio of, two observed values of minimum computational effort.

3.4.1 Simulating Minimum Computational Effort

It is possible to substitute the use of Wilson's method for a Beta-distribution-based simulation method as the two methods offer the same effect—a confidence interval for a proportion. If we make that change in the Wilson-Dependent algorithm (table 3.4) we have the ability to produce random numbers distributed according to the likelihood of the true minimum computational effort. Table 3.17 describes this algorithm.

Although this method (like the Wilson-Dependent method) assumes there is no variability associated with the minimum generation, this approximation has been shown to produce acceptable coverage for typical and even atypical GP results (see section 3.2).

1. Obtain the minimum generation (j), the success proportion at the minimum generation $(P(j))$, the number of runs executed (n) and the population size (M) for a given experiment.
2. Obtain a random number which follows a Beta distribution with an α' parameter of $(P_1(j_1) \cdot n_1) + 1$ and a β' parameter of $((1 - P_1(j_1)) \cdot n_1) + 1$. Label this P_{rand}.
3. Transform P_{rand} with the function

$$E_{\text{rand}} = (j + 1) \cdot R(P_{\text{rand}}) \cdot M$$

to obtain a random number distributed according to the likelihood of the minimum computational effort for the given parameters.

Table 3.17: Algorithm to produce a random number distributed according to the likelihood of a minimum computational effort with parameters j, $P(j)$, n, and M.

Confidence Intervals

We could use the algorithm in table 3.17 to produce an approximate confidence interval for minimum computational effort. Given say 10,000 E_{rand} values, the $\frac{\alpha}{2}$ and $1 - \frac{\alpha}{2}$ quantiles would represent upper and lower limits of a confidence interval at the $(1 - \alpha)$ level for the true minimum computational effort.

The Wilson-Dependent method is however superior for our purposes, given that it produces repeatable results (as it is a deterministic algorithm), and that it is algorithmically and computationally much simpler.

3.4.2 Minimum Computational Effort Differences

We used the simulation algorithm just developed to allow us to form approximate confidence intervals for the difference of two minimum computational effort measures. Table 3.18 details the algorithm.

3.4.3 Minimum Computational Effort Ratios

In his second book, *Genetic Programming II*, Koza introduced a measure he termed the *efficiency ratio* (R_{E}) of two minimum computational effort measurements:

1. Obtain the minimum generation (j_1), the success proportion at the minimum generation $(P_1(j_1))$, the number of runs executed (n_1), and the population size (M_1) for the first experiment.
2. Obtain the same values $(j_2,\ P_2(j_2),\ n_2, M_2)$ for the second experiment.
3. The computational effort for the first experiment is:

   $$E_1 = (j_1 + 1) \cdot R(P_1(j_1)) \cdot M_1$$

   The computational effort for the second experiment is:

   $$E_2 = (j_2 + 1) \cdot R(P_2(j_2)) \cdot M_2$$

   The minimum computational effort difference is then:

   $$\Delta_{\mathrm{E}} = E_1 - E_2$$

4. Obtain X random numbers which follow the expected distribution for the first experiment's parameters (as described in table 3.17). Label them E_{R1}.
5. Obtain another X random numbers for the second experiment. Label these E_{R2}.
6. Find the $\frac{\alpha}{2}$ and $1 - \frac{\alpha}{2}$ quantiles of $E_{\mathrm{R1}} - E_{\mathrm{R2}}$. These provide an upper and lower limit for a $1-\alpha$ confidence interval for the minimum computational effort difference.

Table 3.18: An algorithm to produce a confidence interval at the $1 - \alpha$ level for the difference between two minimum computational effort measurements.

1. Obtain the minimum generation (j_1), the success proportion at the minimum generation ($P_1(j_1)$), the number of runs executed (n_1) and the population size (M_1) for a first experiment.
2. Obtain the same values (j_2, $P_2(j_2)$, n_2, M_2) for the second experiment.
3. The efficiency ratio is then:

$$R_{\mathrm{E}} = \frac{E_1}{E_2} = \frac{(j_1+1)\cdot R(P_1(j_1))\cdot M_1}{(j_2+1)\cdot R(P_2(j_2))\cdot M_2}$$

4. Obtain X random numbers which follow the expected distribution for the first experiment's parameters (as described in table 3.17). Label them E_{R1}.
5. Obtain another X random numbers for the second experiment. Label these E_{R2}.
6. Find the $\frac{\alpha}{2}$ and $1-\frac{\alpha}{2}$ quantiles of $\frac{E_{R1}}{E_{R2}}$. These provide an upper and lower limit for a $1-\alpha$ confidence interval for the efficiency ratio R_{E}.

Table 3.19: Algorithm to produce a confidence interval at the $1-\alpha$ level for the efficiency ratio of two minimum computational effort measurements.

$$R_{\mathrm{E}} = \frac{\text{Computational effort without ADFs}}{\text{Computational Effort with ADFs}} = \frac{E_{\text{without}}}{E_{\text{with}}}$$

It was used throughout the book as an aide to demonstrate the benefits of genetic programming with automatically defined functions (ADFs).

The use of a ratio could however compare any two minimum computational effort measurements and is not specific to the use of ADFs. If you have two methods ‘A’ and ‘B’ and expected ‘A’ to outperform ‘’B then, given two minimum computational effort measurements, E_{A} and E_{B}, $\frac{E_{\mathrm{B}}}{E_{\mathrm{A}}}$ will be greater than one if ‘A’ had the better measure.

Table 3.19 introduces a method to obtain an approximate confidence interval for the ratio of two computational effort statistics. If the confidence interval does not include one then we can be confident (at the $1-\alpha$ level) that the two results are statistically different.

An example of the increased power offered by the use of this method can be found in the work in chapter 9. We were experimenting with the even-4-parity

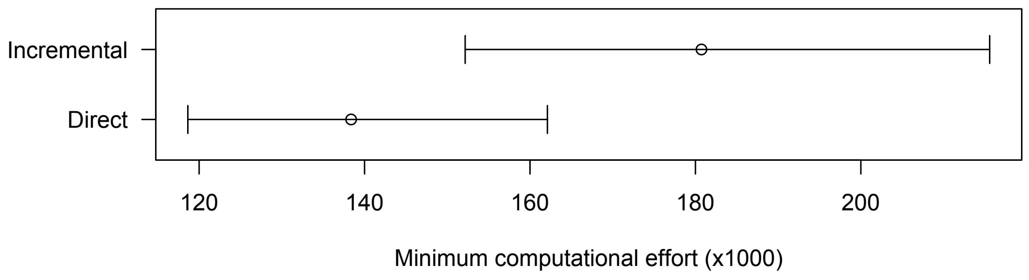


Figure 3.11: Example of two minimum computational effort measures whose individual confidence intervals overlap but whose ratio is significantly different from one.

problem and had two results whose confidence intervals overlapped.[9] Figure 3.11 graphs those two intervals. When the algorithm in table 3.19 was used, a ratio of 0.77 had a 95% confidence interval of 0.60–0.97, thus the two measurements are indeed statistically significantly different.

3.5 Computational Effort versus Success Proportion

Why would you use Koza's minimum computational effort when its confidence intervals are so wide?[10] You could instead use the cumulative probability of success—a measure with much tighter confidence intervals. Cumulative probability of success (also termed success proportion) is indeed a superior measure so long as the two success proportion curves do not cross.

A common approach under success proportion is to draw a conclusion along the lines of: the technique represented by the first curve is superior until they cross, at which point the second technique is a better choice. This section discusses this issue of "crossing" success proportions and the use of Koza's measure. We will also demonstrate why this common approach may produce a misleading analysis.

[9] The results were for even-4-parity with aggressive automatic fitness-based incremental evolution without mutation, with five generations before an automatic step.
[10] Thanks to Riccardo Poli for asking me this question.

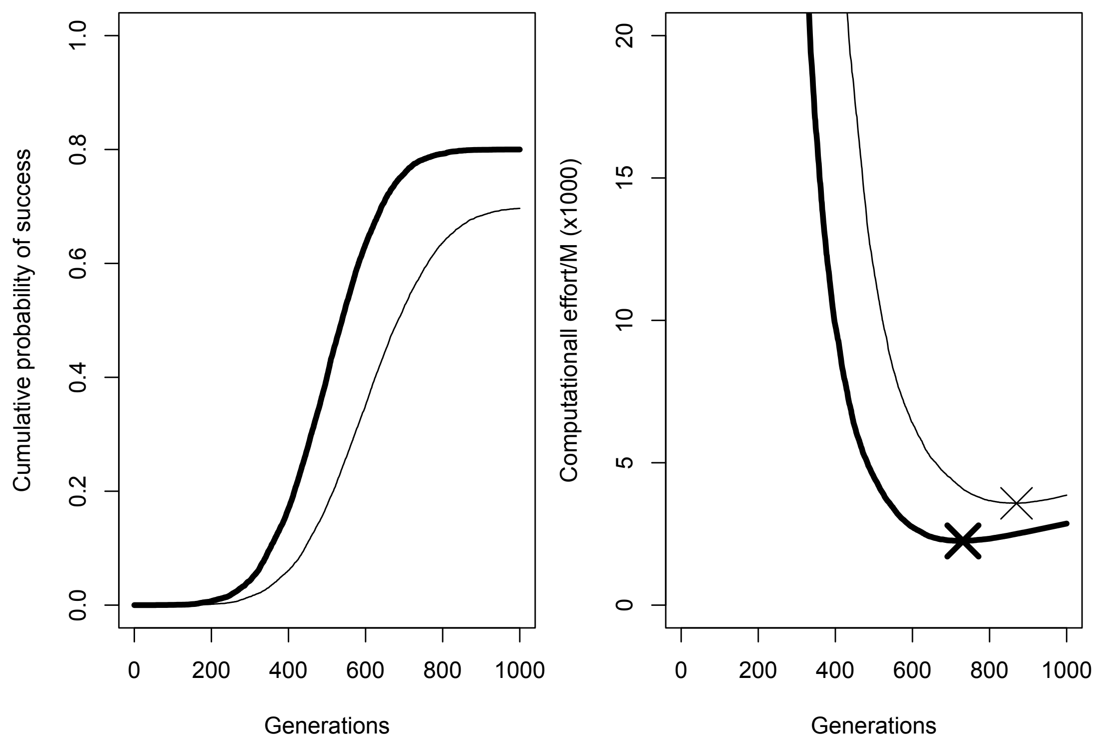


Figure 3.12: So long as the first cumulative probability of success curve (thick line) is non-dominated by the second curve for all generations then the minimum computational effort (as marked with a cross) of the first will be less than that of the second.

3.5.1 Non-Crossing Success Proportions

If one success proportion curve lies completely above a second then its minimum computational effort will be less than that of the second. Figure 3.12 graphs such a scenario.

To prove this, consider two success proportion curves, $P_1(i)$ and $P_2(i)$, where the first dominates the second:

$$P_1(i) \geq P_2(i) \; \forall i$$

Transforming both with the function R will produce two new curves, but this time the second will dominate the first:

$$-P_1(i) \leq -P_2(i) \; \forall i$$

$$1 - P_1(i) \leq 1 - P_2(i) \; \forall i$$

$$\log(1 - P_1(i)) \leq \log(1 - P_2(i)) \; \forall i$$
$$\frac{\log(1-z)}{\log(1 - P_1(i))} \leq \frac{\log(1-z)}{\log(1 - P_2(i))} \; \forall i$$
$$R(P_1(i)) \leq R(P_2(i)) \; \forall i$$

(Defending that last statement is not entirely trivial: if we say that when $P_1(i)$ is greater than z we set the left-hand side of the inequality to 1 and when $P_2(i)$ is greater than z we set the right hand side to 1, then the inequality holds: when $P_1(i) > z$ and $P_2(i) < z$ then $\frac{\log(1-z)}{\log(1-P_2(i))} < 1$; and when $P_1(i) > z$ and $P_2(i) > z$ both sides are set to 1. The case where $P_1(i) < z$ and $P_2(i) > z$ cannot occur because we know $P_1(i) \geq P_2(i) \; \forall i$.)

Multiplying both transformed curves by $M \cdot (i+1)$ does not alter the inequality:[11]

$$M \cdot (i+1) \cdot R(P_1(i)) \leq M \cdot (i+1) \cdot R(P_2(i+k)) \; \forall i$$

As a result, the minimum of the first transformed curve multiplied by $M \cdot (i+1)$ must be less than or equal to the minimum of the second multiplied by $M \cdot (i+1)$:

$$\min_i M \cdot (i+1) \cdot R(P_1(i)) \leq \min_i M \cdot (i+1) \cdot R(P_2(i))$$

Thus, the minimum computational effort calculated from the first cumulative probability curve is less than or equal to that from the second.

The Better Statistic?

In the situation where you suspect that one curve is dominated by another and you are interested in only a specific generation (for example, the final generation), it is better to use the more powerful success proportion statistic at that generation. If your assumption of domination is correct then Koza's measure will result in the same conclusion—that the dominated curve is the better choice.

However, if you *know* that one curve dominates the other (because the vast majority of cumulative success probabilities give non-overlapping confidence intervals for each generation), then you almost certainly have sufficient data to produce a statistically significant result for minimum computational effort.

To demonstrate this, consider two curves where one dominates another and

[11] Note that by definition M, the population size, is greater than or equal to one, and that i is also non-negative.

Graph	Curve	Mean	Std. Dev.	Success factor
Left	Lower-peaked	150	50	0.2
Left	Higher-peaked	600	150	0.7
Right	Lower-peaked	75	25	0.25
Right	Higher-peaked	600	150	0.8

Table 3.20: Mean and standard deviation of the normal distributions used to plot the curves in figure 3.13. Each distribution was multiplied by the specified "success factor".

the confidence intervals per generation do not overlap. Applying the function R to the confidence limits of both curves will result in two transformed confidence intervals that again do not overlap (although the distance between them may be reduced to zero if the cumulative probability is greater than z). Multiplying these two transformed confidence limits by $M \cdot (i + 1)$ will have no impact on whether they overlap, thus we now have two computational effort curves whose confidence intervals do not overlap—one computational effort band dominates the other. Consider the confidence interval of the minimum computational effort of the upper computational effort curve; it cannot overlap the confidence interval of the minimum of the other computational effort curve. If it were to do so then either one band does not dominate the other or the two confidence intervals are not at the minimum computational effort.

3.5.2 Crossing Success Proportions

If two success proportion curves cross then neither curve dominates the other and the previous argument is not useful. It was for this case that Poli argued for a "two-part" conclusion: that the first is superior to the second before the intersection of the curves, but after the intersection the second becomes the superior.

The graphs in figure 3.13 should now be considered.

All four curves were produced from normal distributions. The left graph's lower peaked curve has a mean of 150 generations with a standard deviation of 50; the distribution was multiplied by a "success factor" of 0.2. Table 3.20 gives the details of the other curves.

If you were to use success proportion to analyse these two graphs then there would be very little difference in the conclusions. One might say that, when compared to the left graph, the right graph shows a slight but general performance

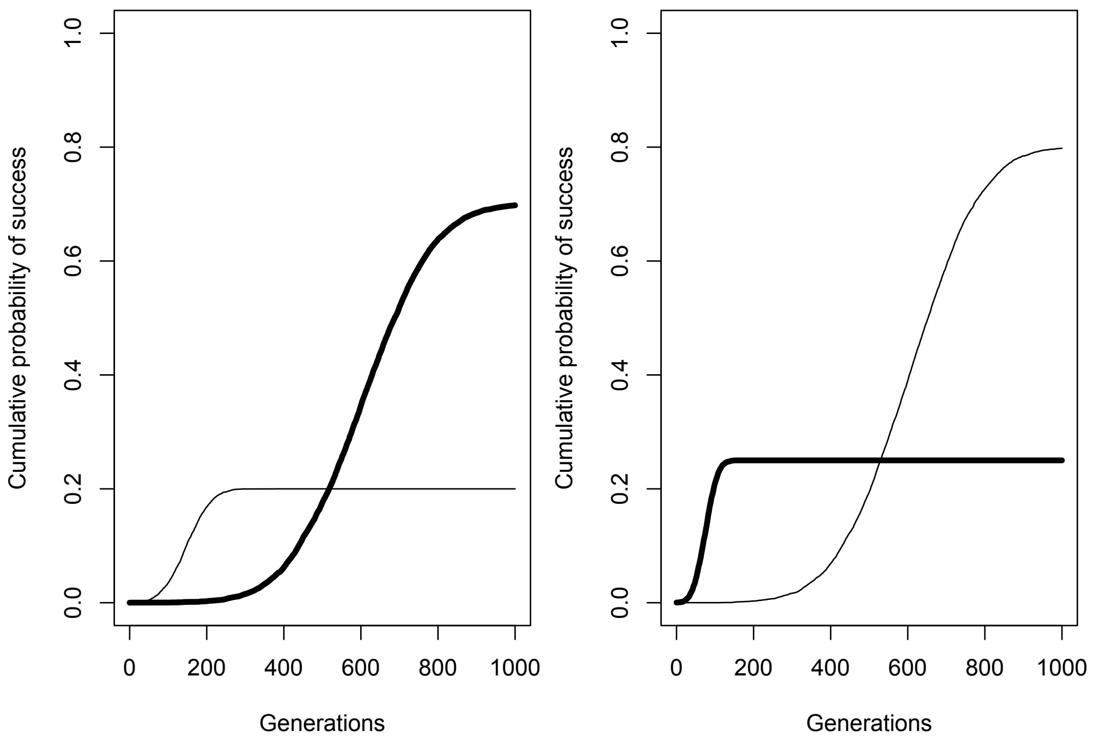


Figure 3.13: It is this scenario that demonstrates the greatest benefit of minimum computational effort over success proportion. The thicker line represents the cumulative success probability curve that generates a lower minimum computational effort.

Graph	Curve	Min. Gen. (j)	$R(j)$	E/M
Left	Lower-peaked	211	23.5	4,988
Left	Higher-peaked	858	4.2	3,567
Right	Lower-peaked	106	18.2	1,951
Right	Higher-peaked	878	3.1	2,718

Table 3.21: Generation at which the minimum computational effort occurs (minimum generation, j), number of runs required ($R(j)$), and minimum computational effort (E) for each of the curves plotted in figure 3.13.

improvement for both curves. You could also say that despite the slight improvement the cross-over point has stayed relatively static at about 500 generations. It is most likely that one would draw the same conclusion for both graphs: that the technique with the higher performance at the final generation was superior.

If, on the other hand, you were to use Koza's minimum computational effort you would conclude that the graph on the left was very different to the graph on the right. For the left graph, Koza's measure says that superior curve is the one that has the higher cumulative probability of success peak, while for the right graph it is the one with the lower peak.

Using Koza's measure you would conclude that, for the lower-peaked curve in the left graph, for a 99% chance of finding a solution, you would execute "23.5" runs of that GP system to approximately 210 generations. Thus, for a 99% chance of finding a solution, you would have processed approximately 5,000M individuals. (So if $M = 500$, Koza's minimum computational effort $E \approx$ 2,500,000.) Table 3.21 gives results for Koza's measure for the other curves.

What is important about these examples is not the detail but instead the fact that there exist cases where a comparison based solely on the use of cumulative success proportion may produce a misleading analysis when compared to an analysis that utilises Koza's measure.

3.5.3 Summary

The use of cumulative success proportion produces a result consistent with Koza's measure whenever one curve dominates another. However to demonstrate that requires statistically significant results for the vast majority of generations, which means you almost certainly have sufficient data to produce a statistically significant result for minimum computational effort.

When the cumulative success probability curves cross, we have shown that Koza's measure can give a different conclusion to an analysis based only on a per-generation comparison of cumulative success proportion. Which result you pay more attention to depends on what you want to measure. Koza's minimum computational effort tries to answer the question "how much effort would be required?"—surely a more useful focus than success proportion's "which one's better at a specific generation?".

However, Koza's measure is not perfect:

- Luke and Panait have already registered their concern that both cumulative probability of success and Koza's minimum computational effort ignore the dependence typically present across generations [85].
- From a practitioner's perspective, it is somewhat dubious to expect to execute exactly the optimal number of runs when that number can only be calculated after the fact.
- Obtaining statistically significant results using the Wilson-Dependent method requires a large number of runs. To find significance between the two curves in the left graph of figure 3.13 requires 495 runs at 95% confidence and 212 runs at 80%. Similarly to obtain significance between the two graphs on the right of figure 3.13 requires at least 365 runs for 95% and 156 for 80%.
- Further, in section 8.5.2 we show minimum computational effort is actually an upper bound and sometimes unable to measure a reduction of the cost of failure.

The "Success effort" measure addresses the first two items in that list. It is considered next.

Chapter 4

Success Effort

Success effort, as we discussed in section 2.8, measures the expected number of generations before a solution will be found. It may be calculated from a collection of runs by:

$$\frac{\text{mean}(g)}{p} \tag{4.1}$$

where g is the vector of generations at which the runs terminated (generations-to-termination) and p is the proportion of runs that found a solution.

Success effort can be compared to Koza's well known computational effort statistic (see chapter 3). Minimum computational effort answers the question "what is the smallest total expected number of generations in order for a solution to be found 99% of the time, if the optimal number of runs are performed to a pre-specified number of generations?". Success effort answers the question "if I execute one run after another, what is the average number of generations that will be executed before a solution will be found?". A comparison of the philosophical values of these two questions can be found in section 5.3.

4.1 Confidence Intervals

Table 4.1 defines a method, based on a standard resampling technique, for producing confidence intervals for the success effort statistic. It has, however, been indicated that such a method may not be reliable [46, 64].

We now introduce another method based on the simulation of the two likelihood functions for the true mean generation and the true probability of success.[1]

[1] Code for the implementation of this algorithm and complete results for all experiments are available from the website listed in appendix D.

1. Obtain n independent runs. Label these as the source set.
2. Repeat $B = 10,000$ times:
   (a) Select, with replacement, n runs from the source set. Label these S.
   (b) Calculate the mean of the generations in S.
   (c) Divide the mean by the proportion of successes in S.
3. Find the 2.5% and 97.5% quantiles of the B ratios. These provide an upper and lower range on a 95% confidence interval for the true value of the success effort statistic.

Table 4.1: Resampling algorithm to produce 95% confidence intervals for the success effort statistic.

The numerator of the success effort ratio (equation 4.1) is the observed (sample) mean of the generations-to-termination and estimates the true mean number of generations-to-termination for a given problem domain and genetic programming configuration. The likelihood function of the true mean is proportional to a normal distribution with a mean equal to the observed mean and a standard deviation equal to $\frac{s}{\sqrt{n}}$, where s is the observed standard deviation and n is the number of trials (runs), provided n is not too small ($n > 25$ say). This follows from the Central Limit Theorem [24].

The denominator of the success effort ratio is the observed probability of success (the proportion of successes) and similarly estimates the true probability of success. The likelihood function of the true probability of success is proportional to a Beta distribution whose α variable is $np + 1$ and whose β variable is $n(1 - p) + 1$, where p is the success proportion and n is the number of runs [79].

Each likelihood function may be used to simulate the corresponding quantity. This corresponds to a commonly-used Bayesian statistical approach using a non-informative prior distribution for each true parameter, which results in posterior distributions (the probability distribution of each true parameter, given the observed data) given by the above normal and Beta distributions respectively [79].

The (true) mean generation and the (true) proportion of successes are dependent variables because, as the proportion of successes decreases, the chance of observing a cut-off generation increases and this in turn impacts the mean

1. Obtain n independent runs. Count the number of successes n_{s}, the number of failures n_{f}. $p = \frac{n_{\mathrm{s}}}{n}$. Extract the vector of generations for the successful runs g_{s}, and the vector for the failures g_{f}.
2. Produce $B = 10,000$ random variables that are normally distributed with mean equal to $\mathrm{mean}(g_{\mathrm{s}})$ and with a standard deviation of $\frac{\mathrm{sd}()}{\sqrt{n_{\mathrm{s}}}}$. Label these G_{s}.
3. Produce B random variables that are normally distributed with mean equal to $\mathrm{mean}(g_{\mathrm{f}})$ and with a standard deviation of $\frac{\mathrm{sd}(g_{\mathrm{f}})}{\sqrt{n_{\mathrm{f}}}}$. Label these G_{f}.
4. $G = p \cdot G_{\mathrm{s}} + (1 - p) \cdot G_{\mathrm{f}}$
5. Produce B random variables that follow a beta distribution with parameters $\alpha = pn + 1$ and $\beta = (1 - p)n + 1$. Label these P.
6. Find the 2.5% and 97.5% quantiles of $\frac{G}{P}$. These are the limits of a 95% confidence interval for the true value of the success effort statistic.

Table 4.2: Simulation algorithm to produce 95% confidence intervals for the success effort statistic.

generation. The relationship between the two variables is given by the formula[2]:

$$\mathrm{mean}(g) = p \cdot \mathrm{mean}(g_{\mathrm{s}}) + (1 - p) \cdot \mathrm{mean}(g_{\mathrm{f}}) \tag{4.2}$$

where g represents the vector of generations-to-termination for each run, g_{s} represents the generations-to-termination of the successful runs, and g_{f} represents the generations-to-termination of the failed runs.

The parameters g_{s} and g_{f} have likelihood functions which are normally distributed too (following the same argument as was used for the mean of generations-to-termination above).

Each likelihood function may be used to simulate its corresponding quantity. Thus, the simulation algorithm given in table 4.2 can be used to obtain a confidence interval for success effort ratios.

The simulation algorithm works correctly even if g_{f} is comprised of multiple instances of a single number, as would be obtained if the traditional GP approach were used where the maximum generation was set to 50 generations. If only one success or one failure was observed, then the standard deviation of that observation should be considered to be zero. If zero failures were observed, then

[2]The formula is proved in appendix B.

the cut-off generation should be used in place of mean(g_f) and zero should be used in place of sd(g_f). Finally, if zero successes were observed, the use of this statistic should not even be considered.

It is worth noting that any proportion will follow a normal distribution provided sufficient samples are obtained. Thus an alternative to the Beta distribution would be to model the success proportion with a normal distribution. If this were a valid approximation we could then apply Fieller's theorem [38, 111] to find a confidence interval for the success effort ratio. Very unfortunately, we found the confidence intervals produced with Fieller's theorem to be uselessly wide given the binary success-or-failure data available with GP runs. Swapping the Beta distribution with a normal distribution in the simulation algorithm (table 4.2) results in unacceptably poor coverage for typical GP run sizes (25–200 runs). Neither of these approaches are discussed any further in this thesis.

4.1.1 Coverage

The most important attribute of a statistic's confidence interval is its coverage. Coverage is the proportion of confidence intervals that include the true value. A 95% confidence interval should include the true value 95% of the time. Just as we studied the coverage rates for minimum computational effort in chapter 3, here we study the coverage for success effort.

To assess the level of coverage attained by the two confidence interval methods for success effort (section 4.1) we simulated a large number of GP experiments on different problem domains and at different run sizes.

The problem domains were taken from four large datasets of real GP runs. They are the same datasets as were used in the earlier chapter on computational effort (see page 29) but this time we consider the calculation of success effort:

- *Ant*: Christensen and Oppacher's 27,755 runs [22] of the artificial ant on the Santa-Fe trail; panmictic population of 500; cut-off of 50 generations; mean(g) = 46.7, $p = 0.133$; best estimate of the true success effort 351.8 generations (95% confidence interval[3] 334.2–356.8)

- *Parity*: 3,400 runs of even-4-parity without ADFs [71, 72]; panmictic population of 16,000; cut-off of 50 generations; mean(g) = 16.9, $p = 0.996$; best estimate of true success effort 16.97 generations (95% confidence interval 16.8–17.1)

[3] Calculated using the simulation method (see section 4.1) with B set to 1,000,000.

Problem Domain	Ant	Multiplexor	Parity	Symbreg	Average
Rsampling	93.3%	87.0%	89.6%	94.3%	91.0%
Simulation	94.9%	94.6%	92.4%	95.2%	94.3%

Table 4.3: Coverage statistics by problem domain, averaged over the six run sizes, for the two confidence interval techniques for success effort.

Run Size	25	50	75	100	200	500	Average
Resampling	86.7%	90.4%	90.7%	91.3%	92.8%	94.3%	91.0%
Simulation	92.6%	94.3%	94.2%	94.6%	94.9%	95.1%	94.3%

Table 4.4: Coverage statistics by run size, averaged over the four problem domains, for the two confidence interval techniques for success effort.

- *Symbreg*: Gagné's 1,000 runs of a symbolic regression problem ($x^4 + x^3 + x^2 + x$) [71]; panmictic population of 500; cut-off after 25,000 evaluations (approximately 50 generations); mean(g) = 24.2, $p = 0.726$; best estimate of true success effort 33.3 generations (95% confidence interval 30.3–36.7)

- *Multiplexor*: Gagné's 1,000 runs of the 11-multiplexor problem [71]; panmictic population of 4,000; cut-off after 200,000 evaluations (approximately 50 generations); mean(g) = 18.6, $p = 0.985$; best estimate of true success effort 18.9 generations (95% confidence interval 18.5–19.5)

Simulated run sizes were chosen to be 25, 50, 75, 100, 200 and 500 runs to allow direct comparison to the experiments in the previous chapter.

For each of the 48 combinations of confidence interval method, problem domain, and run size, 10,000 samples of the specified number of runs were randomly selected from the specified large dataset. For each sample the 95% confidence interval for the sample's success effort was calculated using the specified method. Whether the confidence interval included the best estimate of the true success effort (the coverage) was recorded. The best estimate of the true success effort was the success effort calculated over the entire dataset.

This process effectively simulated 10,000 genetic programming experiments over each of the four problem domains and each of the six run sizes, or a total of 240,000 simulated experiments, for both of the confidence interval methods.

Tables 4.3 and 4.4 give the averaged results of these experiments.

The resampling method did not perform well on either the Multiplexor or Parity domains (the two domains where the probability of success was very close to one). Both domains had an average coverage of less than 90%, but for the Multiplexor domain with 25 runs, coverage dropped to an unacceptably low 78.5%. These results are in line with others' work where it was concluded that resampling techniques for ratios did not have good coverage rates [46, 64].

In contrast the simulation method performed well with a coverage of 94.3%, averaged over the four problem domains and six run sizes. It had a minimum coverage of 87.8% on the Parity domain with 25 runs, but by 50 runs the coverage was up at the 91.4% mark (a level the resampling method did not achieve until 500 runs).

4.1.2 Conclusions

Given these results, we conclude the simulation method has appropriate coverage levels and can be used to provide confidence intervals for the success effort statistic. The resampling method, on the other hand, cannot be considered reliable.

The simulation method relies on the distributions of mean-generations-to-success and mean-generations-to-failure and success proportion. The assumptions of these distributions are all underpinned theoretically (thanks to the Central Limit Theorem, and that proportions can be simulated with a Beta distribution), so the method relies only on sufficient runs (say, at least 25) and again, like minimum computational effort, that success is sufficiently frequent that the confidence intervals are useful.

4.2 Comparing Two Success Efforts

Just as it was important to establish confidence intervals for the difference between two minimum computational efforts, so it is important for success effort. We can extend the method for the generation of success effort confidence intervals to confidence intervals for the difference of two success efforts and for the ratio of two success efforts.

Table 4.5 offers a method to produce random numbers distributed according to the likelihood of the true success effort given a set of GP runs.

Table 4.6 describes an algorithm to obtain a confidence interval for the difference of two observed success efforts, while table 4.7 contains an algorithm for the ratio of two success efforts.

It is worth noting that one can easily obtain the confidence one should have in a ratio or difference being above a specified threshold. Rather than finding

1. Obtain n independent runs. Count the number of successes n_s, the number of failures n_f. $p = \frac{n_s}{n}$. Extract the vector of generations for the successful runs g_s, and the vector for the failures g_f.
2. Produce $B = 10,000$ random variables that are normally distributed with mean equal to $\mathrm{mean}(g_s)$ and with a standard deviation of $\frac{\mathrm{sd}(g_s)}{\sqrt{n_s}}$. Label these G_s.
3. Produce B random variables that are normally distributed with mean equal to $\mathrm{mean}(g_f)$ and with a standard deviation of $\frac{\mathrm{sd}(g_f)}{\sqrt{n_f}}$. Label these G_f.
4. $G = pG_s + (1-p)G_f$
5. Produce B random variables that follow a beta distribution with parameters $\alpha = pn + 1$ and $\beta = (1-p)n + 1$. Label these P.
6. Calculate $\frac{G}{P}$. These are B random numbers distributed according to the likelihood of the true success effort given the observed data.

Table 4.5: A simulation algorithm to generate random numbers distributed according to the likelihood of the true success effort.

the values at certain quantiles, all that is necessary is to find the quantile of the certain value. Thus, if you are interested in whether the ratio is above one, find the quantile of the value one; the quantile is equivalent to the confidence one should have that the value is above (or, if subtracted from one, below) one.

4.3 Success Effort versus Success Proportion

Riccardo Poli's question regarding minimum computational effort versus success proportion (from section 3.5) can be reconsidered with success effort in place of Koza's measure: why would one use success effort when success proportion could do the job?

As before, we will consider the impact of whether the success proportion curves cross. We will find very similar results to our analysis of minimum computational effort: that analysis based solely on success proportion can produce a misleading conclusion.

1. Obtain B random numbers distributed according to the likelihood of the true success effort given the parameters of the first experiment (as described in the algorithm in table 4.5). Label the random numbers SE_{r1}.
2. Obtain B random numbers distributed according to the likelihood of the true success effort given the parameters of the second experiment. Label the random numbers SE_{r2}.
3. $\Delta_{SE} = SE_{r1} - SE_{r2}$ are B random numbers distributed according to the likelihood of the difference between the true success effort of the first experiment and the second.
4. The $\frac{\alpha}{2}$ and $1 - \frac{\alpha}{2}$ quantiles of Δ_{SE} represent the limits of a $1 - \alpha$ confidence interval for the true difference.

Table 4.6: An algorithm to produce a confidence interval at the $1 - \alpha$ level for the true difference between two success effort measurements.

1. Obtain B random numbers distributed according to the likelihood of the true success effort given the parameters of the first experiment (as described in the algorithm in table 4.5). Label the random numbers SE_{r1}.
2. Obtain B random numbers distributed according to the likelihood of the true success effort given the parameters of the second experiment. Label the random numbers SE_{r2}.
3. $R_{SE} = \frac{SE_{r1}}{SE_{r2}}$ are B random numbers distributed according to the likelihood of the ratio of the true success effort of the first experiment to that of the second.
4. The $\frac{\alpha}{2}$ and $1 - \frac{\alpha}{2}$ quantiles of R_{SE} represent the limits of a $1 - \alpha$ confidence interval for the true ratio.

Table 4.7: An algorithm to produce a confidence interval at the $1 - \alpha$ level for the true ratio of two success effort measurements.

4.3.1 Non-Crossing Success Proportions

If you have two success proportion curves with one dominating[4] the other then, if the cost of failure is equal to F, the final generation, the dominant curve will have a smaller success effort.

To prove this, consider two success proportion curves $P_1(i)$ and $P_2(i)$ where $P_1(i) \geq P_2(i)\ \forall i$. The cumulative probability of success $P_1(i)$ is the sum from $g = 0$ to i generations of $Y_{\mathrm{S}1}(g)$, the instantaneous probability of successfully finding a solution in generation g [71, page 193]:

$$P_1(i) = \sum_{g=0}^{i} Y_{\mathrm{S}1}(g)$$

A similar statement can be made for P_2 and $Y_{\mathrm{S}2}$. So:

$$\sum_{g=0}^{i} Y_{\mathrm{S}1}(g) \geq \sum_{g=0}^{i} Y_{\mathrm{S}2}(g)\ \forall i \tag{4.3}$$

If we now also consider the instantaneous probability of failure, $Y_{\mathrm{F}1}(g)$ and $Y_{\mathrm{F}2}(g)$, then we can say:

$$\sum_{g=0}^{F}(Y_{\mathrm{S}1}(g) + Y_{\mathrm{F}1}(g)) = \sum_{g=0}^{F}(Y_{\mathrm{S}2}(g) + Y_{\mathrm{F}2}(g)) = 1$$

This next step will be proved in appendix C; we can say:

$$\sum_{g=0}^{F}(Y_{\mathrm{S}1}(g) + Y_{\mathrm{F}1}(g)) \cdot g \leq \sum_{g=0}^{F}(Y_{\mathrm{S}2}(g) + Y_{\mathrm{F}2}(g)) \cdot g \tag{4.4}$$

If we note that $\sum_0^F(Y_{\mathrm{S}1}(g) + Y_{\mathrm{F}1}(g)) = \sum_0^F(Y_{\mathrm{S}2}(g) + Y_{\mathrm{F}2}(g)) = 1$ then:

$$\frac{\sum_0^F(Y_{\mathrm{S}1}(g) + Y_{\mathrm{F}1}(g)) \cdot g}{\sum_0^F(Y_{\mathrm{S}1}(g) + Y_{\mathrm{F}1}(g))} \leq \frac{\sum_0^F(Y_{\mathrm{S}2}(g) + Y_{\mathrm{F}2}(g)) \cdot g}{\sum_0^F(Y_{\mathrm{S}2}(g) + Y_{\mathrm{F}2}(g))}$$

And if we consider g_1 to be the vector of generations at which the runs of P_1 terminated (with either success or failure), and g_2 to be similarly defined for P_2, then the above can be re-written:

$$\mathrm{mean}(g_1) \leq \mathrm{mean}(g_2)$$

[4] For this discussion we will define *dominate* such that: for two curves, $f(x)$ and $g(x)$, $f(x)$ dominates $g(x)$ if and only if $f(x) \geq g(x)\ \forall x$

Graph	Curve	Mean	Std. Dev.	Success factor
Left	Lower-peaked	350	1/3	0.5
Left	Higher-peaked	850	1/15	0.7
Right	Lower-peaked	200	1/4	0.5
Right	Higher-peaked	800	1/15	0.7

Table 4.8: Mean and standard deviation (as a proportion of the mean) of the normal distributions used to plot the curves in figure 4.1. Each distribution was multiplied by the specified "success factor".

If we then chose $p' = \sum_0^F Y_{S1}(g) = P_1(F)$ and $p'' = \sum_0^F Y_{S2}(g) = P_2(F)$ and note that $p' \geq p''$, then:

$$\frac{\text{mean}(g_1)}{p'} \leq \frac{\text{mean}(g_2)}{p''}$$

Thus, the success effort of a dominant curve is less than or equal to the success effort of the dominated curve.

4.3.2 Crossing Success Proportions

The previous section allows us to say that if one success proportion curve dominates another then its success effort will be smaller. But what if one does not dominate the other but instead the two curves cross? We will now show that for such a case analysis using only success proportion can result in misleading conclusions when compared to conclusions based on success effort.

Consider the two graphs in figure 4.1. The only difference between the curves is that those in the right graph are a stretched version of those on the left.

The curves are all formed from a normal distribution scaled by a "success factor"—just as was done for minimum computational effort in section 3.5. Table 4.8 gives the parameters for the curves.

If you were to analyse the curves using success proportion as a measure, you might note that, although both sets of curves plateau at the same final success proportion, those in the right graph achieve the performance earlier. You might also note that the lower of the two curves seems to be most affected, but that the intersection is fairly consistent at about 800 generations. You would most probably conclude that there was little to distinguish the curves on the left from those on the right.

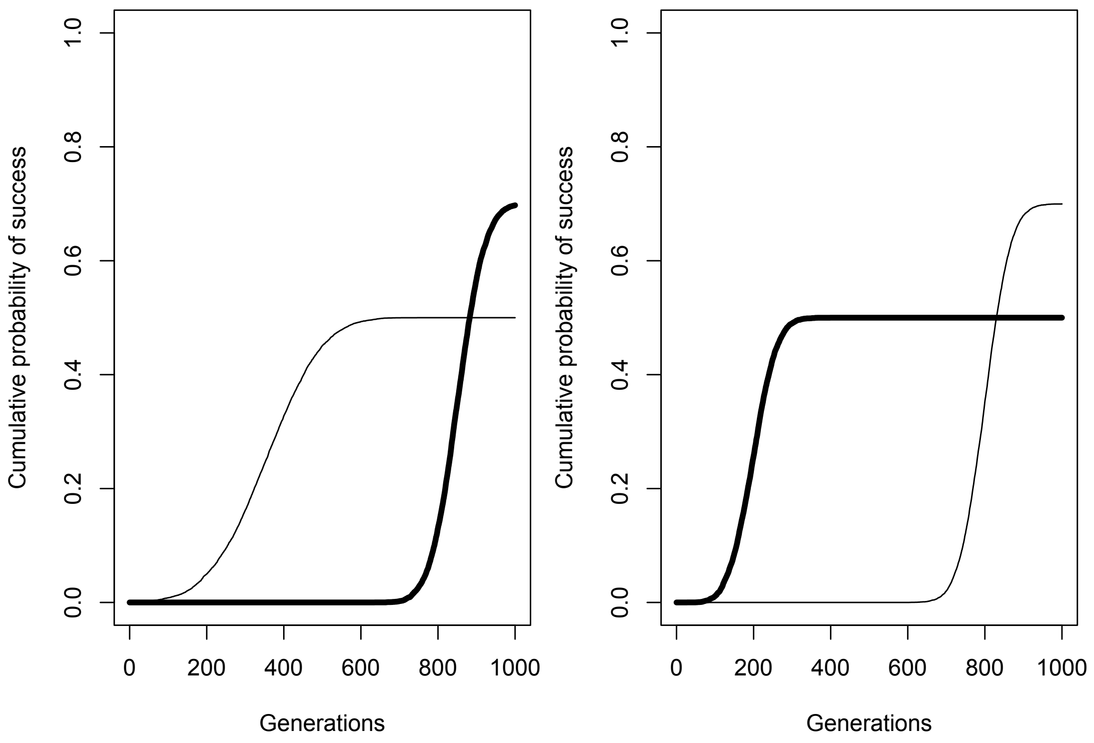


Figure 4.1: It is this scenario that demonstrates the greatest benefit of success effort over success proportion. The thicker line represents the cumulative success probability curve that generates a lower success effort.

Graph	Curve	mean(g)	p	Success Effort
Left	Lower-peaked	673	0.5	1,347
Left	Higher-peaked	895	0.7	1,282
Right	Lower-peaked	601	0.5	1,201
Right	Higher-peaked	859	0.7	1,227

Table 4.9: The mean generation, success proportion, and resulting success effort for each of the curves plotted in figure 4.1.

However, using success effort as a measure would result in quite a different conclusion: for the left graph the higher-peaked curve is the more efficient choice while the lower-peaked curve is superior in the right graph.

Using success effort you would conclude that, for the left graph, were you to execute one run after another until you found a solution, you would execute an average of 1,347 generations for the technique that generated the lower-peaked curve and an average of 1,282 generations for the higher peaked curve. Table 4.9 gives the success effort for each curve.

Although the details are unimportant, what we have shown is an example where the analysis based solely on success proportion was misleading.

4.3.3 Summary

For two success proportion curves, if one dominates the other then the dominant curve will have a success effort that is less than the success effort of the other. In this situation analysis with success proportion will produce the same conclusions as analysis based on success effort. However if the two success proportion curves cross then analysis based solely on success proportion may result in a different, and potentially misleading, conclusion to that based on success effort.

Because success effort measures the expected amount of work that would be required to find a solution, it is almost certainly a more useful measure than knowing the proportion of successes at an arbitrary generation.

Success effort has other advantages:

- Success effort does not suffer the issue of dependency across generations that cumulative success probability curves typically suffer and ignore. Although this may not have a considerable impact on the quality of results, success effort is a “cleaner” measure from a statistical perspective.

- Success effort also gives a very natural measure for a practitioner. The result can be used to predict the number of runs required to be executed back-to-back. That is a far more useful measure than that offered by minimum computational effort.
- Further, success effort is naturally able to assess the benefit of a decreased cost of failure—a benefit we will utilise in Part II.

However, because it considers both the probability of success and the number of generations, when compared to the use of success proportion, success effort will most likely require a larger number of runs for results to be statistically significant.

Chapter 5

Comparison of the Statistics

This chapter compares minimum computational effort, success effort and three other commonly-used single-variable statistics: mean best-of-run fitness, mean generation, and success proportion. We conclude that the five statistics have fairly reliable coverage rates but, although they have different power, we argue that the measures that combine both the probability and the cost of finding a solution (that is, minimum computational effort and success effort) are the most useful for a GP practitioner.

5.1 Coverage

The most important attribute of a statistic's confidence interval is its coverage. Coverage is the proportion of confidence intervals that include the true value. A 95% confidence interval should include the true value 95% of the time.

To assess the level of coverage attained by the five statistics, we simulated a large number of GP experiments on different problem domains and at different run sizes. The problem domains were taken from four large datasets of real GP runs. They are the same datasets as were used in our previous experiments: Ant, Parity, Symbreg, and Multiplexor. Simulated run sizes were chosen to be 25, 50, 75, 100, 200 and 500 runs to match the earlier experiments in minimum computational effort and success effort.

The confidence interval methods can be found in: section 2.4.1 (page 21) for mean best-of-run fitness, section 2.5.1 (page 22) for mean generation, section 2.2.2 (page 12) for success proportion, table 3.4 (page 33) for minimum computational effort, and table 4.2 (page 75) for success effort.

For each of the 120 combinations of the five statistics (and associated confi-

	Ant	Multiplexor	Parity	Symbreg
Success Effort	351.8	16.97	33.3	18.9
	(334.2–356.8)	(16.8–17.1)	(30.3–36.7)	(18.5–19.5)
Comp. Effort	479,344	421,074	33,299	163,045
	(460,637–498,847)	(395,457–450,109)	(30,682–36234)	(149,650–178,817)
Mean Best Fitness	36.8 [gen. 0]	0.645 [gen. 0]	0.744 [gen. 0]	0.633 [gen. 0]
	(36.65–36.85) to	(0.644–0.646)	(0.740–0.748)	(0.632–0.634)
	70.5 [gen. 50]	to 0.9998 [gen. 50]	0.996 [gen. 50]	0.999 [gen. 50]
	(70.40–70.63)	(0.9996–0.9999)	(0.995–0.996)	(0.999–1.000)
Mean Generation	46.7	16.90	24.2	18.6
	(46.54–46.81)	(16.78–17.03)	(22.7–25.6)	(18.2–19.0)
Success Proportion	0 [gen. 0]	0 [gen. 0]	0 [gen. 0]	0 [gen. 0]
	(0–0.0001) to	(0–0.001) to	(0–0.004) to	(0–0.004) to
	0.133 [final gen.]	0.996 [final gen.]	0.726 [final gen.]	0.985 [final gen.]
	(0.129–0.137)	(0.993–0.998)	(0.698–0.753)	(0.975–0.991)

Table 5.1: Best estimate of the true value for each statistic for each problem domain. 95% confidence intervals are shown in parentheses.

dence interval methods), problem domain, and run size, 10,000 samples of the specified number of runs were randomly selected from the specified large dataset[1]. For each sample the 95% confidence interval for the sample's statistic was calculated using the specified method. Whether the confidence interval included the best estimate of the true value of the statistic (the coverage) was recorded.

The best estimate of the true value of the statistic was the statistic calculated over the entire dataset. Table 5.1 gives the best estimate of the true values for each statistic and problem domain. It also includes 95% confidence intervals as calculated using the associated method.

Success effort, minimum computational effort, and mean generation, all produce one value (and one confidence interval) per GP experiment. Because success proportion and mean best fitness are typically plotted against generations, we elected to calculate these statistics (and their confidence intervals) for each generation.

For some samples it was not possible to produce a value for a statistic. An example of this was when zero runs succeed in the selection. In this case, although a success proportion and confidence interval can be calculated, Koza's minimum computational effort balks. In these situations undefined values were just ignored.[2]

[1] Unfortunately, 1,000 runs of Parity were lost before the fitness data could be collected, thus only 2,400 runs were used for the experiments with mean best fitness.

[2] The worst occurrence of this was for 25 runs on the Ant domain where for both minimum computational effort and success effort we ignored just under 3% of their 10,000 confidence intervals.

Problem Domain	Ant	Multiplexor	Parity	Symbreg	Average
Success Effort	94.9%	94.6%	92.4%	95.2%	94.3%
Comp. Effort	92.4%	95.7%	94.0%	94.9%	94.3%
Mean Best Fitness	94.2%	84.2%	66.5%	92.3%	84.3%
Mean Generation	93.0%	90.3%	92.9%	94.7%	92.7%
Success Prop.	95.3%	96.2%	95.5%	95.1%	95.5%

Table 5.2: Coverage statistics by problem domain, averaged over the six run sizes, for the five statistics.

Run Size	25	50	75	100	200	500	Average
Success Effort	92.6%	94.3%	94.2%	94.6%	94.9%	95.1%	94.3%
Comp. Effort	92.3%	94.4%	94.7%	93.8%	94.9%	95.3%	94.3%
Mean Best Fitness	74.0%	80.3%	83.2%	85.5%	89.2%	93.6%	84.3%
Mean Generation	90.2%	91.9%	92.9%	93.1%	93.9%	94.4%	92.7%
Success Prop.	95.5%	95.7%	95.4%	95.6%	95.5%	95.4%	95.5%

Table 5.3: Coverage statistics by run size, averaged over the four problem domains, for the five statistics.

Table 5.2 gives the results of these experiments by problem domain, averaged over the six run sizes. Table 5.3 gives the results by run size, averaged over the problem domains. For mean best fitness and success proportion the results are also averaged over the generations.

Success effort, minimum computational effort, and success proportion all have excellent average coverage statistics. Mean generation's coverage is a little on the low side, but its performance is still quite acceptable.

Mean best fitness on the other hand performs poorly for the Multiplexor domain, and exceptionally poorly on the Parity dataset. Its performance is worst, a meagre 47% coverage, on the Parity domain with run sizes of 25 runs.

The explanation lies in the number of zero-width confidence intervals. Because the Parity and Multiplexor domains achieve such high success proportions, the majority of samples in the later generations contain only fitness scores of one (i.e. a solution has been found). This complete lack of variance in the observed data leads to a zero-width confidence interval (see section 2.4.1), which does not include the true value of very slightly less than one. Figure 5.1 graphs this phenomenon per generation.

The conclusion that should be made is that although the coverage levels for mean best fitness are poor, they are perhaps not completely representative of the truth given that the majority of GP practitioners would intuitively question the

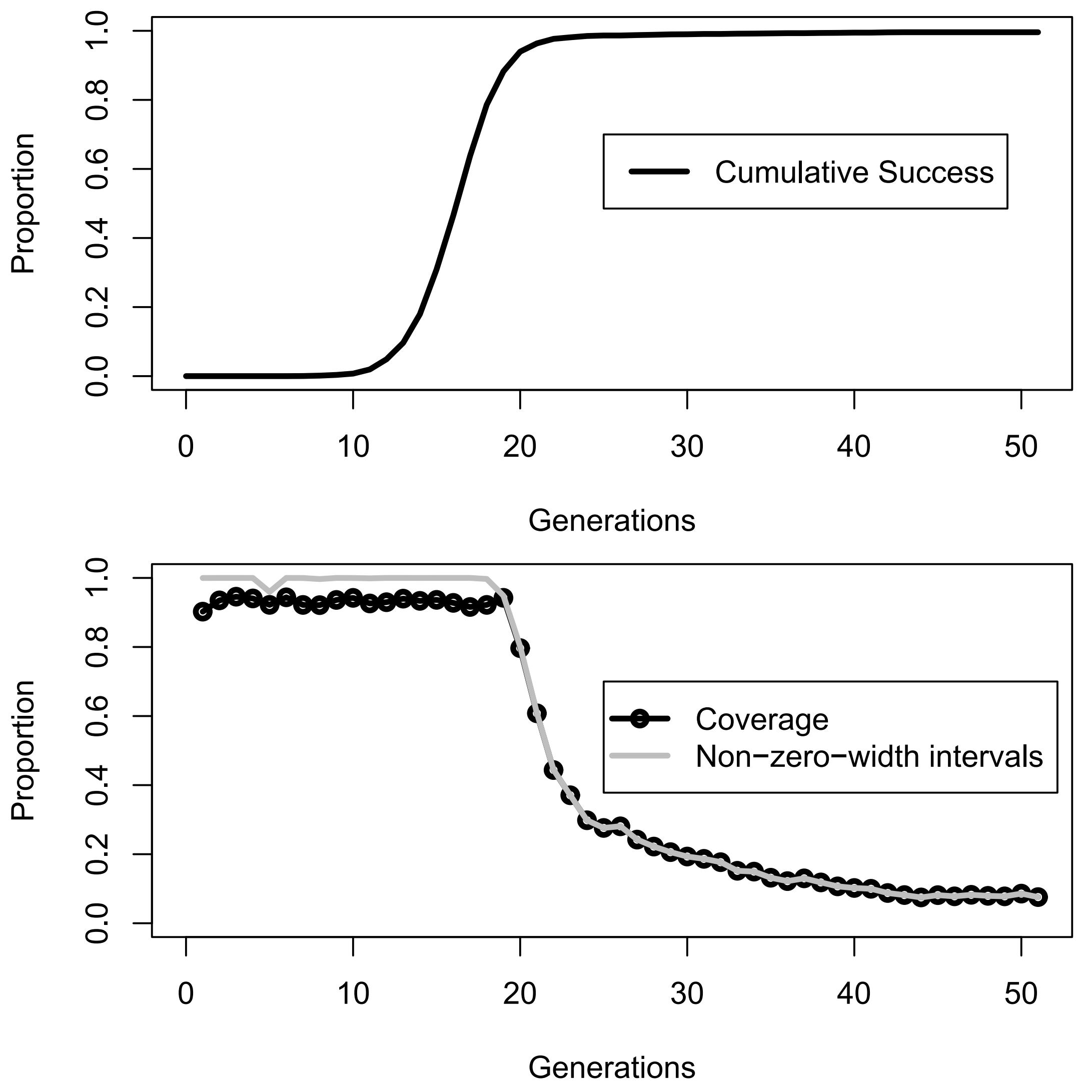


Figure 5.1: Cumulative success (upper) and mean best fitness coverage levels and non-zero-width intervals (lower) for the Parity domain with a run size of 25 runs.

validity of a zero-width confidence interval. Indeed, if zero-width intervals are ignored, mean best fitness has an average coverage, over all problem domains and run sizes, of 94.4%.

5.2 Power

A well as coverage, another statistical attribute of interest is *power* [27]. The power of a statistic measures its ability to distinguish a difference between two values. Typically in GP we are offering a variation in the methodology and then comparing the performance of the variation against Koza's canonical approach. The comparison is achieved by measuring a value (for example minimum computational effort) for each method and asking the question "are the two measurements significantly different?". A more powerful statistic can detect a difference more frequently.

One way to assess the relative power of two techniques is to consider the width of the confidence intervals. Tighter confidence intervals are an indication of a more powerful statistic (given that the coverage levels are the same). In this section we look at the width of each of the five statistics as a ratio of the true value, a measure that allows for easier comparison between problem domains.

The width-ratio is a natural measure, indicating the width of the confidence interval relative to the value being measured. It has also previously been used in GP by Keijzer et al. [69]. However, the measure is not ideal. One need only shift the origin of the values being measured and the width-ratio will be affected. We will however accept the measure as it is intended only for the purposes of informing a user of the potential variance associated with each of the five statistics.

For each of the 120 combinations of the five statistics (and associated confidence interval methods), problem domain, and run size, 10,000 samples of the specified number of runs were randomly selected from the specified large dataset. For each sample the 95% confidence interval for the sample's statistic was calculated using the specified method. The width of the confidence interval as a ratio of the best estimate of the true value was recorded.

Table 5.4 gives the width-ratios averaged over the four problem domains. Table 5.5 shows the results averaged over the six runs sizes.

From these results we can conclude that mean best fitness produced by far the narrowest width-ratios, with an overall average of 0.02. This should be interpreted as: averaged over the four problem domains and run sizes, mean best

Problem Domain	Ant	Multiplexor	Parity	Symbreg	Average
Success Effort	1.50	0.23	0.18	0.68	0.65
Comp. Effort	1.72	0.58	0.59	0.56	0.86
Mean Best Fitness	0.07	0.00	0.00	0.01	0.02
Mean Generation	0.11	0.14	0.09	0.41	0.19
Success Prop.	1.40	0.16	0.14	0.29	0.50

Table 5.4: Median confidence interval widths as a ratio of the best estimate of the true value, by problem domain, averaged over the six run sizes, for the five statistics.

Run Size	25	50	75	100	200	500	Average
Success Effort	1.00	0.63	0.49	0.42	0.29	0.18	0.50
Comp. Effort	1.11	0.76	0.63	0.55	0.39	0.26	0.62
Mean Best Fitness	0.02	0.01	0.01	0.01	0.01	0.00	0.01
Mean Generation	0.22	0.16	0.13	0.11	0.08	0.05	0.13
Success Prop.	0.57	0.42	0.35	0.30	0.22	0.14	0.33

Table 5.5: Median confidence interval widths as a ratio of the best estimate of the true value, by run size, averaged over the four problem domains, for the five statistics.

fitness produced confidence intervals whose median widths were just 2% of the value being measured.[3]

The statistics, in increasing order of width-ratio, were: mean best fitness, mean generation, success proportion, success effort, and computational effort. That ordering held for all run sizes (averaged over the four problem domains) and for three out of the four problem domains (averaged over the six run sizes).

The problem domain where the ordering was not consistent was Symbreg. In that domain mean generation swaps positions with success proportion and success effort swaps positions with minimum computational effort. However this result seemed to be the exception.

Success effort demonstrates its greatest difference in width-ratio, when compared to computational effort, with the Parity and Multiplexor domains. These are the domains where the success probability is highest. It is conceivable that this condition may be a requirement for success effort to out-power computational effort.

From these results one might conclude that, if all else is equal, mean fitness

[3] There was no notable difference between the width-ratios as stated (2.31%), and the width-ratios where the zero-width intervals were removed (2.37%).

is the ideal choice of statistic while minimum computational effort is the least desirable. However, all else is *not* equal.

5.3 A Better Statistic?

Although mean generation tells you how long, on average, it takes to terminate a run, it does not tell you how often a solution was found. And although success proportion tells you how often a solution was found, if it is stated for only the final generation, then it gives no idea how long the runs spent evolving.

Mean best fitness is similar to success proportion except that its sensitivity is greater. Mean best fitness can say how the runs are improving even if none has found a solution. But like success proportion, if it is only stated for the final generation, the measure gives no idea how much effort was required to obtain that level of fitness.

There are partial solutions to these issues. If the vast majority of runs completed successfully, then that may be sufficient information to give meaning to a mean generation statistic. But if it's desirable to quantify "vast majority", then both mean generation and success proportion can be quoted together, thus giving an indication of both the success rate and the amount of evolution required. Equally, mean generation and mean best fitness could be paired too.

However it can be quite tricky to compare two pairs of values. If both statistics are better or worse than their competitor, then it makes for an obvious comparison. Indeed, even if one of the statistics is equal, then the other statistic can be easily compared. However, what if the comparison is against a result that has a higher mean generation (i.e. it takes longer) and a higher success rate? In this case a conclusive comparison is not obvious.

Another approach is to quote success proportion or mean best fitness for every generation. This is commonly achieved through the use of a graph. As well as being cumbersome, this approach does not achieve its purpose. How should one compare two graphs that intersect with one another? Such a situation occurs when one GP variation performs well early on but is out-performed later. In such a scenario there is no obvious choice for which is the better. Indeed, in sections 3.5 and 4.3 we demonstrated that analysis based on this approach may produce a misleading conclusion.

The problem lies in how the level of success and the length of time should be *combined*. Both minimum computational effort and success effort attempt to find an acceptable answer to this.

It is possible to consider the question that is answered by minimum computational effort and success effort. Koza's statistic tells you how much effort would be required to find a solution 99% of the time were you to execute the optimal number of runs to a fixed generation (the minimum generation) irrespective of the success or failure of any run.

Success effort in comparison answers the question: given the specific settings how many generations will be required (on average) before a solution will be found. As a consequence, success effort includes the cut-off generation, and therefore, if the cost of failure is constant, the number of restarts that will be required. For the statistic to be meaningful, runs would have to be performed sequentially.

If genetic programming is to be used on hard problems, Luke has shown that longer run lengths are to be preferred over many shorter runs [83]. As a result we could expect practitioners to dedicate their resources to a single run, rather than split them into an "optimal" number of runs. Such practitioners will be very interested in the cut-off generation which tells them when their effort on the current run should be aborted. Practitioners will be interested in a statistic that offers a direct indication of the cost that they will incur if they use GP. In this light, success effort can be seen to be a more desirable measure than computational effort.

5.4 Summary

In this part we have:

- Introduced methods to produce confidence intervals for Koza's minimum computational effort measure and concluded that the Wilson-Dependent is reliable.

- Re-introduced the success effort statistic and defined two confidence interval methods for it. We concluded the simulated parametric approach was reliable.

- Shown that, for Koza's minimum computational effort, mean best-of-run fitness, mean generation, and success proportion, the confidence intervals produced are all reliable (bar the zero-width intervals of mean best fitness).

- Shown that success effort and minimum computational effort are philosophically more desirable than the other statistics if you are interested in both the proportion of success and the length of time it took to find solutions.

- Shown that success effort had generally narrower confidence interval width ratios and is a somewhat more desirable statistic than computational effort.

Because mean best fitness, mean generation, and success proportion only deal with one of the two parameters of general interest, their confidence intervals are notably tighter than those for computational effort and success effort. If you are in the unlikely position of being interested in only one of the two variables, then using one of mean best fitness, mean generation, or success proportion is a good choice.

If you are in the typical situation of being interested in both the proportion of runs that find a solution and the number of generations that were required to find the solutions, then the use of minimum computational effort or success effort is preferable. We have shown that success effort is philosophically more desirable, and statistically a possibly more powerful measure, than computational effort. We thus recommend the use of success effort be at least considered.

In the following chapters we further compare the practicality of success effort and minimum computational effort.

Part II

Developing Incremental Evolution

Chapter 6

Review: Incremental Evolution

This chapter provides an introduction to incremental evolution. It offers a review of some of the uses and previous research in the area. This review acts as a starting point for the following chapters where incremental evolution techniques are developed.

6.1 Introduction

Incremental evolution is the sequential use of simpler evolutionary environments that gradually increase in difficulty until the goal environment is reached. Depending on the researcher, the motivation for incrementally increasing the difficulty of the evolutionary environment is: to increase the likelihood of finding a solution, or to decrease the cost of finding a solution, or to increase the quality of solutions, or—ideally—all of these. Incremental evolution offers the human an opportunity to coach the evolutionary system's development. It can be seen as a way to add domain-specific knowledge.

Throughout this thesis we will refer to each of the evolutionary environments as a *stage*. By definition, incremental evolution has a minimum of two stages—the most common use in the literature—but the number of stages can be much larger, with automated options sometimes using hundreds of stages to evolve a solution [53, 122].

6.2 Detail of Incremental Evolution

6.2.1 Terminology

Harvey and his colleagues were possibly the first to use the term “incremental evolution” (in a 1992 technical report [26]), although similar ideas do pre-date their work [19, 30, 121, 122]. Unfortunately, however, many terms have been used for very similar ideas.

Layered Learning

The most common term in competition with “incremental evolution” is almost certainly “layered learning”. Layered learning was the subject of Stone’s 1998 PhD thesis [106] and although there were originally specifications that separated the two ideas (see section 6.3.1), later research blurred those distinctions [57, 61, 63, 120].

Großmann’s use of “incremental learning” [55] was very similar in approach to Stone’s layered learning.

Shaping

A significant body of research exists on the use of “shaping”—a technique used to train animals (and indeed humans) that was pioneered from the 1930’s by experimental psychologist Skinner [59, chapter 7]. The concept is to expose the “student” to the material to be learnt in graded steps of increased difficulty.

Learning by Easy Missions

Asada et al. used the idea of shaping and termed it “Learning from Easy Missions” or “LEM” [9, 10]. Again, if the evolutionary process is seen as learning, then incremental evolution and LEM do not differ.

GP-ISLES

Hsu and his colleagues [57, 63] used the acronym “GP-ISLES”, which stood for “Genetic Programming—Incrementally Staged Learning from Easier Subtasks”, in what amounted to experiments in incremental evolution. Fortunately, nobody else seems to have used this term.

Seeding

Finally, the concept of "seeding" is very similar to incremental evolution. Typically a seed-individual will be hand-coded and inserted into the initial generation. If the seed has come from an evolutionary process then this use of seeding is equivalent to incremental evolution.

6.2.2 Functional versus Environmental

Barlow [14] divided the use of incremental evolution into two types: functional and environmental. In functional incremental evolution, the fitness function is varied across the stages. In environmental incremental evolution the environment, in which the individuals evolve, is changed across the stages.

Barlow claimed functional incremental evolution to be less popular, but that both had been used successfully. He used both approaches in his work on evolving controllers for unmanned aerial vehicles (UAVs) [14].

6.2.3 Transferring the Population Between Stages

One topic that remains only very lightly studied is how the population should transition from stage to stage. There exist a number of options. The population of the $(n + 1)$th stage can be formed by copying the entire final population of the nth stage [58, 112]. This seems a very popular option. Other techniques include either the introduction of new genetic material via randomly generated individuals or the mutation of individuals [113] (both techniques that encourage diversity), or biasing selection of the individuals that performed better in the previous stage [61].

Hsu and Gustafson [61] looked at the first two suggestions using a two-stage "keep-away soccer" domain: *LLGP-Best* took the best individual of the first stage and cloned it to make up the entire initial population of the second stage; *LLGP-All* transfered the final population of the first stage so that it formed the initial population of the second stage. Their comparison showed *LLGP-All* notably outperformed *LLGP-Best* (although no statistical significance information was provided).

6.2.4 Hierarchies, Directions and Concurrency

Incremental evolution is typically a sequential process of moving from one stage to the next until the goal is reached. This, however is not a requirement. Researchers have studied the use of hierarchies, bidirectional connections and concurrency in incremental evolution stages, each with an amount of success.

Winkeler and Manjunath's "mixed increments" are interesting because they combined the use of *demes*—a technique known to be beneficial for GP [73, chapter 62]—and different incremental evolution stages [124]. They found this to be beneficial when compared to "standard" incremental evolution.

Kalganova found benefit in evolution in two directions with results performing significantly better than direct evolution [68]. Fukunaga also noticed the potential bidirectionally in incremental evolution [42].

Whiteson et al. [119, 120] developed "concurrent layered learning" by extending Stone's work on layered learning. Their hypothesis was that there exists layered learning scenarios where it is necessary to learn two behaviours (layers) simultaneously. They applied this to the Keep-Away Soccer domain and found concurrent layered learning significantly improved average fitness.

6.2.5 Diversity and Overtraining

Diversity can be a concern when incremental evolution is used. Diversity is the range of genetic material in the population; when a population evolves in an environment it is possible for the genetic material to converge to a common state. If the environment were to change (as one expects in incremental evolution) then the population may not have the ability to adapt. Incremental evolution may require diversity-enhancing operators such as mutation [113].

Winkeler and Manjunath [124] concluded that the optimum strategy for incremental evolution was for each stage to evolve until it solves that stage's problem. Their work, however, considered only one problem domain, so the extrapolation of their conclusion into a general rule might be inappropriate.

Their problem domain was visual tracking; the objective was to keep an object in the centre of a camera's image. In the first stage of evolution, the object was stationary. This proved to be easily solved within 20 generations. If the first stage evolved far past the point of finding a solution (50 generations), the resulting population was overtrained and was demonstrated to be a worse starting point for the second stage than if the first stage had been terminated when it had first found a solution.

It was also shown that undertraining on the first stage produced a better starting population than the overtrained group. However, if measured in generations, the undertrained population was less undertrained (10 generations) than the overtrained population was overtrained (30 generations). To achieve under-training, the first stage was terminated when the best individual had surpassed a "partially solved" threshold.

Barlow suggested that the use of a crowding distance in his multi-objective fitness function encouraged diversity [14, page 144]. In his research, diversity was not an issue even though evolution in each stage was terminated when it reach the allocated number of generations (400 or 600)—a strategy not recommended by Winkeler and Manjunath. His crowding distance measure was like that of the NSGA-II algorithm [32]: it encouraged selection of (otherwise equivalent) individuals that were in a less populated area of the search space.

As a counter-example to Barlow, but to reinforce Winkeler and Manjunath's results, Fukunaga [42] demonstrated that spending too many generations in the first of two stages was almost certainly detrimental to the performance of the evader in his pursuit-evasion domain.

Eriksson [35] studied the possibility that including a learning component with the evolutionary system may increase the diversity in the population thus improving the performance of incremental evolution. His work was not conclusive and although he hoped to continue this research, his later publications did not cover this subject.

6.2.6 Manual Incremental Evolution

Easily the most common use of incremental evolution in the literature is to fix the number of stages, the environments, and their fitness functions before evolution begins. Along with this specification is the implicit requirement for when the individuals will transition from stage to stage.

Despite Winkeler and Manjunath's work, the most common approach appears to be that each stage is allocated a fixed number of generations. The other approach is the use of a success-based transition criteria—but Andre and Teller used a unique variation.

Andre and Teller used multiple fitness measurements [6] but unlike the more common studies on Pareto optimisation, their fitness functions had an ordering such that any success in a higher order fitness measure would completely swamp fitnesses in all lower orders. The effect of this was that as the population improved the fitness function automatically became more challenging. This specification of fitness functions can be seen as a way to coach the evolutionary system and in a way that is automatically appropriate for the level of the individuals (even if they are at different levels in the same generation).

We study a manual incremental evolution approach using a success-based transition criteria in chapter 8.

6.2.7 Automatic Incremental Evolution

We are aware of two papers that used an automatic technique to specify a series of stages for genetic programming [53, 119]. Both papers used an "environmental" form of incremental evolution that automatically increased the difficulty of the environment. They used a success-based trigger to scale up the difficulty of the problem domain—once the population reached a predefined success criteria, the parameter was changed so that the domain became more difficult. This topic is developed in chapter 9 of this thesis and others' work is discussed further in section 9.2.

Another approach that could be considered "automatic" is that of Cliff et al. [26]. Their system was given the ability, through mutation, to increase the complexity of the search space (neural network architectures). Using this mutation operator an individual would enter what might be more complex problem-space terrain but, if successful, its architecture would propagate through the population and the population would eventually transition to the new space.

6.2.8 Success Rates

Incremental evolution is certainly not a guaranteed success. Our earlier [112] work plus the work in chapters 8 and 9 demonstrates this. Others too have either found difficulty in getting incremental evolution to outperform direct evolution [20, 67] or have shown that it is very parameter specific [41, 42].

Fukunaga hypothesised that incremental evolution will be successful if the early stages are "smoothed" versions of the goal problem [41, page 434]. He suggested that to develop a theory as to when incremental evolution will be successful, "attention should be focused not on easy/hard 'problems' (in the intuitive sense), but on the analysis of easy/hard cost surfaces for a particular search algorithm". Although he stated some interest in developing this theory, we are ten years on and Fukunaga has not published anything more on the topic.

6.2.9 Naysayers

There is an argument—the No Free Lunch Theorem—that "no single algorithm outperforms memorising random search or enumeration when amortised over all possible functions" [21, page 121]. This argument can be used to show that incremental evolution will not, in general, be any better than a random search.

Schmidhuber presented a way of searching for a "universal algorithm" that solved a sequence of tasks [102]. His defence against the No Free Lunch Theorem is worth considering. His opinion was that we are not typically interested in *general* problems, but rather real world problems. His argument applies equally well to incremental evolution.

> Successive real world problems ... tend to be closely related. In particular, teachers usually provide sequences of more and more complex tasks with very similar solutions. Problem sequences that humans consider to be *interesting* are *atypical* ... For all *interesting* problems the consideration of previous work is justified, to the extent that *interestingness* implies relatedness to what's already known. [102, section 3.3.2]

6.3 Related Techniques

There are a number of techniques that are related to incremental evolution, the most important of which is probably layered learning (as originally specified). Shaping, seeding and co-evolution also deserve consideration.

6.3.1 Layered Learning

The term "layered learning" was coined by Stone [106, 107]. It is a specification for learning complex behaviours.

Layered learning defines a multi-stage learning process. Each stage (termed a layer) is conceived by the human "coach". Any learning technique may be used at any layer, but the resulting behaviour should be useful to the next layer in the learning process.

The effectiveness of layered learning was demonstrated via the performance of the 1996–8 robotic-soccer teams named CMUnited. In the small-robot competition of RoboCup-97, CMUnited-97 finished first of four teams. In the simulator competition of RoboCup-98, CMUnited-98 won all eight of its games with a combined score of 66–0!

Stone's thesis [106] described the implementation of a three-layer training system for robotic soccer. The first layer used a neural-network that learnt to intercept the ball. The second layer used a decision tree to evaluate whether a given pass would be successful. The second layer built upon the results of the first by using the learnt interception skill for both pass-receivers and the opponents that were used to make the task more difficult. The third layer used a learning algorithm that Stone himself developed. It built upon the success of the second layer by learning pass-selection given a competing team.

Stone described two further layers that were not implemented in his thesis, strategic positioning and strategic adaption, that would have further built upon the skills learnt in previous layers.

Layered learning (as originally conceived) and incremental evolution differ in the following ways:

- Incremental evolution uses an evolutionary learning algorithm at every stage; layered learning may use any learning algorithm at any layer.

- Layered learning learns a behaviour at each layer. These behaviours are combined to solve the goal problem. Incremental evolution learns a behaviour in an initially simple environment (the first stage), this behaviour is honed by later stages until a solution is found for the goal problem.

  For example, in the case of Stone's first layer the neural network learnt how best to intercept the ball. Once this was learnt, the network's weights were never reconsidered. It was expected that later layers would learn not only what to do with this interception skill, but also how to reduce any errors it may potentially contain [106, page 99].

- Incremental evolution directly uses the results of previous layers: unless "frozen"[1], it can manipulate their genetic representation with further evolution. In layered learning the results become a block that can be reused but may not receive further development.

- Layered learning suffers from the potential misalignment of learnt behaviours. Whiteson wrote, "no matter how carefully the special training environments for the lower layers are designed, there are bound to be imperfections. Discrepancies will inevitably exist between the behaviours that those [early-layer] environments encourage and the behaviours that are optimal in the target [goal] domain" [119, page 20]. Because incremental evolution builds upon the genetic code of earlier solutions, this potential misalignment may not be so significant; eventually, individuals will be evaluated and hence honed in the goal domain. It may be that early stages were not helpful,

[1] An example of such freezing can be found in the work of Hsu et al. [63]

> but once in the goal stage, evolution has the potential to overcome a poor starting point.

Thus layered learning in its original form differs from incremental evolution. However, since the publication of Stone's thesis, continued research has blurred these distinctions. Hsu et al. used layered learning in a way that cannot be distinguished from incremental evolution [61–63] as did Whiteson and Stone when they developed "concurrent layered learning" [120].

6.3.2 Shaping

An entertaining example of shaping was carried out in 1970 on primary school classes [59, pages 240–1]. First, "a token economy was instituted" where students could use tokens to buy rewards. They earned tokens if the whole class was "paying attention" to the teacher for a set percentage of the lesson. A cue-light "prominently displayed on the teacher's desk served to inform the class when the teacher thought they were all paying attention". The teacher would switch the light without comment, but when the light was red the class knew that they were not accumulating "attention time" and so might not get their tokens. Shaping occurred by increasing the proportion of "attention time" required before the class obtained their tokens. Initially the level was set at just 12 minutes of the half-hour class, but after a number of successes the level was increased. The experiment was a "spectacular success" with class attention levels reaching 90%. Without the reinforcement, attention time plummeted to 10–20%.

The use of shaping is sufficiently pervasive that it has also been used for machine learning [33, 52, 94, 100, 122] (and even the guidance of conversations [29]). If the evolutionary system can be considered as the entity that learns, then incremental evolution can easily be seen as a form of shaping.

The references in section 2.5 of Gomez's PhD thesis [53] should be consulted by a reader interested in further work on shaping as applied to machine learning. Also, section 5.1.3 of Perkin's thesis gives a "taxonomy of shaping" [94].

6.3.3 Seeding

Seeding is a concept similar to incremental evolution. It is often implemented as a single seed-individual added to an otherwise traditionally-initialised population. One concern with this method is that the seed is typically so successful it quickly propagates throughout the population eliminating all genetic competition. Consequently, the seed is sometimes used exclusively to populate the initial generation—although this method suffers from a lack of diversity. If the source of the seed was an evolutionary run then the use of seeding is equivalent to incremental evolution.

Andre and Teller used a form of seeding, because "GP is remarkably slow to learn generalisable routines to reliably run to and kick the [robot-soccer] ball when given only the most basic of primitives" [6]. They gave every individual hand-coded "automatically defined functions" (ADFs) that encoded that basic functionality. They did not discuss the effect of their technique but the mere fact they entered into a RoboCup competition [36] a team that was almost completely evolved from scratch was a success in itself.

From the literature, seeding the initial population has been said to: improve the seed provided [1–3, 108, 109], produce results more quickly [95, 108, 118], produce higher performance individuals in early generations [54, 103], be beneficial in difficult problem domains [103], improve the rate of convergence [54, 76, 77, 95], result in a higher fitness level for the best individual [1–3, 28, 37, 54, 76, 93, 103], and produce individuals that are more robust [95].

6.3.4 Co-evolution

Incremental evolution can be seen as similar to co-evolution. In co-evolution two groups of individuals compete against each other and thus ramp up both groups' abilities. The fitness function (or environment) in incremental evolution can be seen as equivalent to the competition available through co-evolution; as the individuals improve so too does the fitness function (or environment). This is especially true when the transition between incremental evolution stages is success-based and even more apparent when the stages are automatically defined. Harvey discussed this topic, considering co-evolution as evolution versus the human experimenter [58, page 190].

6.4 Incremental versus Direct Evolution

When comparing incremental evolution to direct evolution there are three comparisons often considered: (i) the change in the success proportion at the final generation, (ii) the cost of finding comparable solutions, and (iii) the quality of solutions found. Incremental evolution has been said to increase the success proportion, decrease the cost of finding a solution, and increase the quality of solutions (although not necessarily all at once). We next discuss research in each of those areas.

However, when researchers have compared incremental evolution to direct evolution they fall into two camps: those who include the cost of the non-goal stages, and those who do not. That anyone falls into the second camp is quite remarkable given that such comparisons are almost certainly unfair—how can you justify failing to include the cost of the preparation when it is the effect of that preparation that you are studying?

6.4.1 Success Proportion

Barlow applied incremental evolution to developing a controller for an unmanned aerial vehicle (UAV) [14, 16, 17]. His comparisons with direct evolution did not include the full cost of incremental evolution as the evolutionary computation associated with any non-goal stages were not counted when comparisons were made with direct evolution. At worst, direct evolution experiments that ran for 600 generations were compared to incremental evolution runs that ran for 1800 generations [14, 15]. It is not possible to make adjustments to allow more fair comparisons as the direct evolution runs were allocated an insufficient number of generations. However, if it is assumed that incremental evolution may freely start with the population produced from the final non-goal state, then Barlow's results showed that, "the use of incremental evolution increased evolution's chances of evolving fit controllers" [14, page 127]

Chapters 8 and 9 of this thesis consider the impact of incremental evolution from a success proportion perspective.

6.4.2 Quality of a Solution

Further to Barlow's increased probability of success in evolving UAV controllers, those successful controllers that were evolved through incremental evolution were tested on all the problem domains of the non-goal states. It was found that the controllers were still able to successfully solve the non-goal problem domains [14]. This was considered an advantage not shared by direct evolution (although the direct-evolution controllers were tested only informally [15]).

In his PhD thesis, Harvey [58] compared direct and incremental evolution using the travelling salesman problem (TSP). Given the same number of evaluations, Harvey's three variations of incremental evolution all performed as well as direct evolution, even though incremental evolution solved up to 123 extra problems en route.

Winkeler and Manjunath [124] compared incremental and direct evolution. They said that, on training data, programs evolved through incremental evolution were often statistically superior to their direct evolution counterparts. On untrained data, most of the programs evolved were as robust as those evolved with direct evolution. Although it is not entirely clear, it appears that they have ensured that, when making comparisons, each technique was allowed the same number of program evaluations.

Fukunaga and Kahng considered two application domains in their 1995 paper [42]: a pursuit-evasion problem, and a “Tracker” problem that was based on Koza’s artificial ant and its “Santa Fe” food trail [71]. For both domains they used a two-stage incremental approach.

For their pursuit-evasion problem they found that the first stage could be both easier or harder than the goal stage and still incremental evolution could produce individuals with better fitness that those produced by direct evolution.

Similar results were found with the artificial ant domain. There were three training environments: easy, intermediate, and the original (hard) Santa Fe trail. When the goal was the easy trail, direct evolution always outperformed incremental evolution (when considered from a mean best-fitness perspective): it did not help to spend time on either the intermediate or the Santa Fe trails. When the goal was either the intermediate or Santa Fe trail, it was most beneficial for the first stage to train on the easy trail. However, it was also beneficial, when compared to direct evolution, to start with the intermediate trail and move to the Santa Fe trail. Notably, incremental evolution would also outperform direct evolution if a small number of generations was initially spent on the (hard) Santa Fe trail even when the goal was the intermediate trail.

In his thesis, Gomez [53] compared direct and incremental evolution using three problem domains: double pole balancing, capturing prey, and guidance of a finless rocket. The double-pole balancing problem is discussed in section 6.4.3 as it showed the cost of using incremental evolution can be less than that of direct evolution. Gomez’s other two domains however are excellent examples of incremental evolution producing higher-performance individuals.

Prey capture was the second problem domain in Gomez’s thesis [53] where direct evolution was compared to incremental evolution. The objective was to evolve a controller that chased prey.

Direct evolution was applied to evolving the predator’s controller. The prey was randomly placed just within the sensing range of the predator. The prey was then allowed to make four moves while the predator was stationary. After the four moves, the prey moved at the same speed as the predator. With direct evolution the controllers improved slightly over the first 20 generations. The population quickly converged, where the best individual moved around in a mechanical fashion. “Direct evolution failed in every simulation.”

Incremental evolution’s first stage was against a stationary prey. Once a capable controller had evolved, the problem was increased in difficulty. In the second stage the prey was allowed to make two moves while the predator was stationary. The prey then stayed still. The prey’s first two moves potentially took it out of the predator’s sensing range. This increased the problem’s difficulty as it sometimes required the controller to remember the general direction of the prey. The next two stages incremented the number of initial moves so that by the end of the fourth stage the prey made four initial moves. The fifth stage had the prey making four initial moves and then moving at about one-third the speed of the predator. Once a successful controller had evolved, the speed increased to 0.6, 0.8, and finally to 1.0. Thus, the final stage was the same problem that direct evolution tried to solve. The fitness-based performance of the controller evolved through incremental evolution was about nine times that of the controller evolved using direct evolution [53, figure 7.4].

Very similar results in the same domain are reported by Gomez in a paper that pre-dates his thesis [51].

Gomez also experimented with evolving the prey (rather than the predator) in a still-earlier paper [50]. There he considered multiple predators that moved at up to half the speed of the prey. The first stage started with one predator that moved at one-third the speed of the prey. Next a second predator was added. Later stages increased their speeds to 40% and then 50% of the speed of the prey. Again, incremental evolution drastically outperformed direct evolution.

In Gomez’s thesis [53], the third comparison between incremental and direct evolution involved the active guidance of a finless rocket. The aim was to evolve a controller that could guide a simulated, highly-unstable, rocket as high as possible. This task was too difficult for direct evolution: all members of the initial population performed so poorly that evolution stalled and converged to a local maxima. It is, however, unclear that direct evolution was allowed as much computational time as incremental evolution. Incremental evolution started with a rocket with small fins; it solved that problem in approximately 600,000 evaluations. A further 50,000 evaluations were required to successfully transition to the finless rocket. Gomez concludes "incremental evolution was critical to the success . . . Evolving a controller for the finless rocket directly would have required much greater computational resources and allowed for much less experimentation in the domain."

Hsu and Gustafson [61] used "keep-away soccer" [56] to compare the performance of direct and incremental evolution (under the terminology of "layered learning"). In the first stage, three offensive agents (which were clones of the same GP individual) were taught to pass a ball to each other. The fitness function was directly proportional to the number of successful passes. The second stage added a defender that could move at twice the speed of an offensive agent and the fitness function became directly proportional to the number of turnovers that occurred. The defender's strategy was hand-coded and not evolved.

In a preliminary experiment, the first layer was allocated 40 generations and the second had 61. Direct and incremental evolution produced solutions that were very similar in quality. The only noteworthy difference was that individuals produced through direct evolution were one-third larger in size.

An effort was then made to tune the number of generations allocated to the two stages of incremental evolution. They found the best results were obtained when ten generations were spent on the first stage and 91 on the second. With this tuning, the average of the best-of-run controllers was under six turnovers per simulation. This compared to direct evolution's average of nine turnovers when evolved for the same number of generations. They likened their success to training human soccer teams where "individuals first learnt to play well together in pairs and small groups, then as a coordinated team"

Two years later, Hsu et al. [63] reimplemented Hsu and Gustafson's previous study. Again, incremental evolution outperformed direct evolution. In a footnote they commented that a three-stage configuration was trialled. The extra stage was inserted between the original two; it included the three offensive agents as well as the defensive one, but the fitness function remained directly proportional to the number of successful passes. They stated that improved results were seen, but no further information was provided.

Although Stone [106] did not experimentally compare his work in layered learning with direct evolution, his first principle for applying layered learning was that "layered learning is designed for domains that are too complex for learning a mapping directly from an agent's sensory inputs to its actuator outputs." He claimed robotic soccer was such a problem domain and cited two papers as empirical evidence.

The first, by Luke et al. [84], used GP to evolve a team. They provided hand-coded functions and terminals such as “is a team mate closer to the ball than I am?” and “block the goal from the ball”—a starting point that was at a significantly higher level than Stone’s. This team won two of its four games at RoboCup-97, losing in the second round. Stone claimed his competing team to be “qualitatively clearly a better team”.

The second, by Andre and Teller [6], also used GP to evolve a team. Stone said, “This time, the agents were indeed allowed to learn directly from their sensory input representation. While making some impressive progress given the challenging nature of the approach, this entry was unable to advance past the first round in the tournament.”

Stone [106, page 100] wrote that this anecdotal evidence allowed him to claim his layered-learning method generated “more complex and successful learnt behaviours than possible if learning straight from the agents’ sensory inputs.”

In contrast to all those examples of increased solution quality, de Garis noticed a concerning effect [31]. Although incremental evolution produced better results within fewer generations for his neural network problem, the quality of solutions was comparably not as high if evolution was allowed to continue. He wondered if other incremental evolution researchers would also notice this “better sooner, worst later” effect. Although we have not read any such comment, it is certainly possible that this effect may have occurred in others’ work had their runs been extended.

6.4.3 Cost of Finding a Solution

After his studies of the artificial ant and the pursuit-evasion domain, Fukunaga applied both incremental and direct evolution to the design of a soil probe for a NASA mission to Mars [41]. Incremental evolution outperformed direct if the first of the two stages were evolved with softer (easier) soil than the goal stage, but direct evolution would outperform incremental if the incremental evolution started with firmer (more challenging) simulated soil. Fukunaga concluded, “incremental evolution was able to find higher-quality solutions in less time than [direct evolution] on the Mars microprobe design problem.”

For the double pole balancing problem [52] in Gomez’z thesis [53], two poles were mounted on a cart and the objective was to balance both poles simultaneously. Both direct and incremental evolution were allocated 1000 generations per evolutionary run. Direct evolution had a 20% success rate at evolving a successful controller when the shorter pole was 40% the length of the longer pole ($e_{0.4}$). For $e_{0.45}$ direct evolution evolved a controller only once in fifty runs. At $e_{0.5}$, direct evolution failed to evolve a controller within 1000 generations. Incremental evolution was far more successful. It had a 100% success rate for $e_{0.44}$, and 96% at $e_{0.5}$. One run even found a solution for $e_{0.66}$. Compared to direct evolution, incremental evolution required 75% fewer evaluations to solve $e_{0.4}$.

Chapters 8, 9 and 10 of this thesis look at reductions in cost associated with the use of incremental evolution.

6.5 Summary

Chapter 8 studies a form of incremental evolution introduced by Jackson. We utilise the statistics developed in Part I to show that, if generations are weighted according to the amount of work done, then the technique can sometimes be beneficial. In chapter 9 an automated version of the technique is developed and we show it produces a statistically-significant improvement. We then extend the study and consider, in chapter 10, a very interesting early-termination heuristic. However we must first study direct evolution's performance in our chosen problem domain and it is that topic that we turn to in the next chapter.

Chapter 7

Direct Evolution

This chapter's primary goal is to provide base-line comparisons for the following chapters. Even-n-parity problems are executed both with and without ADFs and the results are given. The chapter concludes with a replication of Koza's computational efficiency analysis—but with the addition of multiple measures and confidence intervals.

7.1 Even-n-Parity

The even-n-parity problem domain was introduced to genetic programming in Koza's first book [71] and further studied in his second [72]. The domain has become one of the more studied of problems used in genetic programming research.

The concept is simple: given n boolean inputs, the task is to produce an electronic circuit that returns true if an even number of the inputs are true, and returns false otherwise. The available gates are: AND, OR, NAND, and NOR. Note that NOT and XOR and not included in this list as they enormously simplify the problem.

Koza demonstrated that as n increases linearly, the problem difficulty increases exponentially and that the use of ADFs drastically improves the tractability of the domain [72, chapter 6].

Since Koza's books, other researchers have shown that mutation is more significant than Koza claimed, that there is benefit in evolving the main-tree and ADF-trees individually, in assessing only the fitness cases that are most difficult, in the use of population demes, in a sub-machine-code implementation, in smooth uniform crossover and in smooth point mutation [97]. Although, like others, we will use even-n-parity to demonstrate potential improvements available via our

Problem	Population (M)	ADFs	Args
Even-4	500	2	{2,3}
Even-5	1,000	3	{2,3,4}
Even-6	1,000	3	{2,3,4}
Even-7	2,000	3	{2,3,4}

Table 7.1: Population sizes, number of ADFs, and number of ADF arguments for the different problem sizes used in the experiments.

methods, except for our use of sub-machine-code GP, we will leave it to future research to identify the best combination of all the improvements that have been offered.

7.2 Method

Standard—or *direct* evolution—genetic programming runs were executed for the even-n-parity problem domain for values of n of 4, 5, 6 and 7. They were executed both with and without Koza's automatically defined functions up to 150 generations.

Given that there is evidence that the difficulty of this problem rises exponentially as n increases [72, page 192], we increased the population size as n increased. The population sizes specified were an attempt to match the performance that Jackson observed [67]. Table 7.1 gives the population sizes used for each value of n.

Whenever ADFs are used there is the standard problem of how to configure them [89]. We considered Koza's rule-of-thumb, that there should be $n - 2$ automatically defined functions (thus $n - 1$ trees per individual) with each ADF taking one more argument than the last, with the first taking two arguments [71, page 535]. For even-7-parity this would result in individuals with five trees taking two to six arguments. Executing runs with such a set up is considerably expensive from a computational perspective, so although Koza's suggestion was used for even-4 and even-5, we scaled it back for even-6 and even-7. Table 7.1 details the ADF configuration used for each value of n.

The following "minor" parameters were used:[1]

[1] We say "minor" here because that is what they are traditionally called. It happens that we are not particularly interested in optimising these parameters, but our experiments indicated that doing so could possibly have more of a positive influence than any of the novel techniques offered in this part of the thesis.

- A pre-release of Open BEAGLE version 3.1.0 was used.
- The initialisation operator was Koza's ramped half-and-half with a maximum depth of five and a minimum depth of two (without the kozagrow patch).
- The selection operator was tournament selection with a group size of seven.
- No mutation was employed, instead crossover was the sole genetic operator with resulting trees limited to a maximum depth of 17. Two attempts were made to satisfy this depth criteria before two new parents were chosen for crossover.
- Evolution occurred over a single deme.
- Finally, the "fast" version of the parity code was used.

7.3 Results without ADFs

Figures 7.1, 7.2, and 7.3 plot Koza-style graphs of computational effort and cumulative success probability. The numbers in the hexagonal boxes relate to the calculation of minimum computational effort (see chapter 3) while the label attached to the end of the cumulative success proportion curve gives the level of success at the final generation. Table 7.2 gives the values of the three measures from Part I.

The plot of even-4's performance (figure 7.1) has no conspicuously interesting features. The generation at which the minimum computational effort occurs is sufficiently before the maximum number of generations that one can be confident that longer runs would not impact this measure. Instantaneous success proportion has well and truly peaked (at about generation 35) so lengthening the runs would likely have only an increasingly small impact on the final success proportion.

Even-5's performance curves (figure 7.2) indicate that the instantaneous success proportion may still be rising, thus it is likely that the observation of minimum computational effort has been affected by the choice of maximum generation. Longer runs would likely show notably different values for this measure. Success proportion too would be influenced.

Figure 7.3 (Even-6) is an excellent example of computational effort's sensitivity to low success proportions. Only six of the 500 runs found a 100%-correct

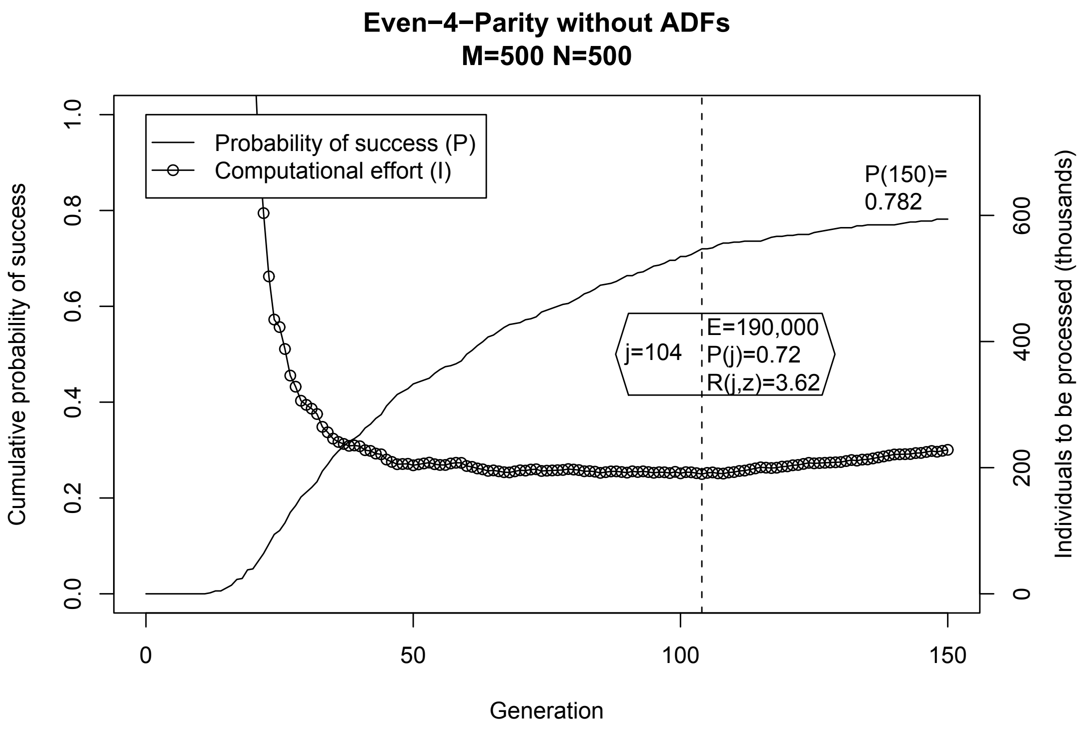


Figure 7.1: Computational effort and cumulative probability of success for even-4-parity without ADFs.

solution, and each success is clearly visible as a step on the computational effort line. The six successes influence the computational effort by between 16% and 50% (when compared to the computational effort for the generation before the occurrence of the success). The confidence interval (table 7.2) demonstrates this: it is nearly twice as wide as the observed value—and that's with 500 runs.

500 runs on Even-7 failed to find even a single 100%-correct solution. As a result neither computational effort nor success effort are able to be calculated. The final success proportion of 0% is the only one of our three measures that we can get from this data.

7.4 Results with ADFs

Figures 7.4 to 7.7 graph computational effort and cumulative success proportion curves for even-4, 5, 6, and 7 when ADFs were used. Table 7.3 lists the minimum computational effort, success effort and final success proportion plus confidence intervals for all four problems.

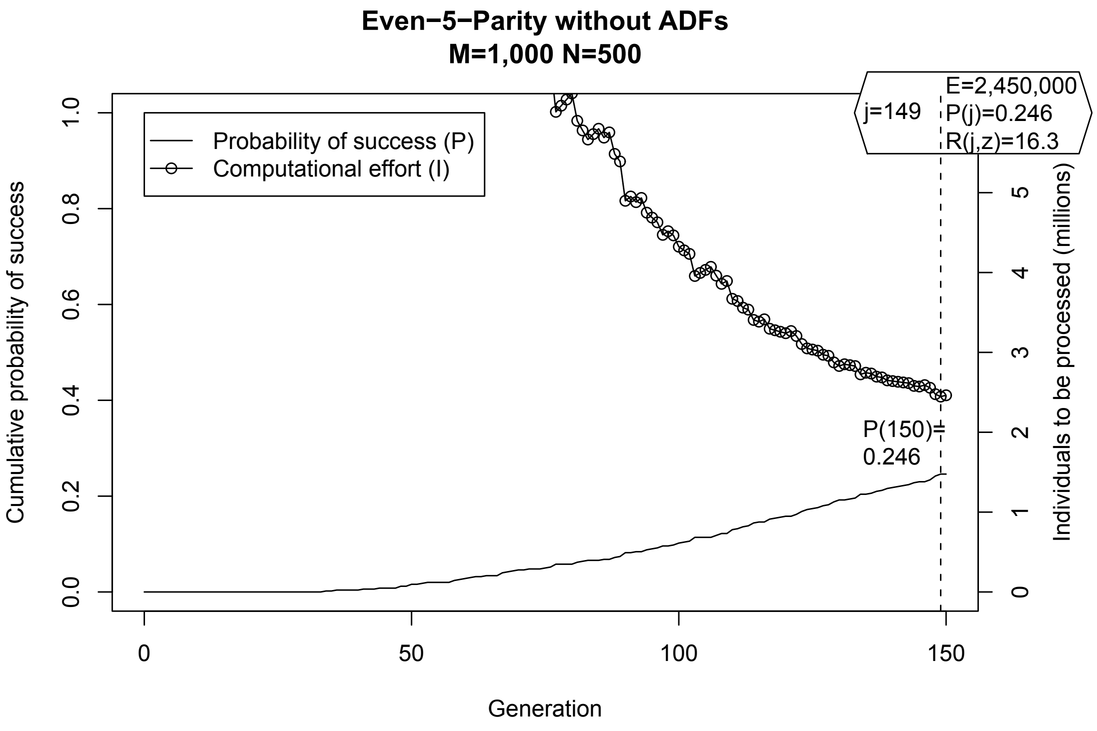


Figure 7.2: Computational effort and cumulative probability of success for even-5-parity without ADFs.

Problem	Min. Comp. Effort	Success Effort	Success Prop. $P(150)$
Even-4	190,000	97	78%
	(171,000–213,000)	(88–107)	(74–82)
Even-5	2,450,000	564	25%
	(2,050,000–2,930,000)	(481–668)	(21–29)
Even-6	50,900,000	12,500	1.2%
	(21.8–119 million)	(5,700–26,000)	(0.6–2.6)
Even-7	no successes	no successes	0%
			(0–0.8)

Table 7.2: Minimum computational effort, success effort, and final success proportion for the different problem sizes used in the experiments without ADFs. 95% confidence intervals are shown in parentheses.

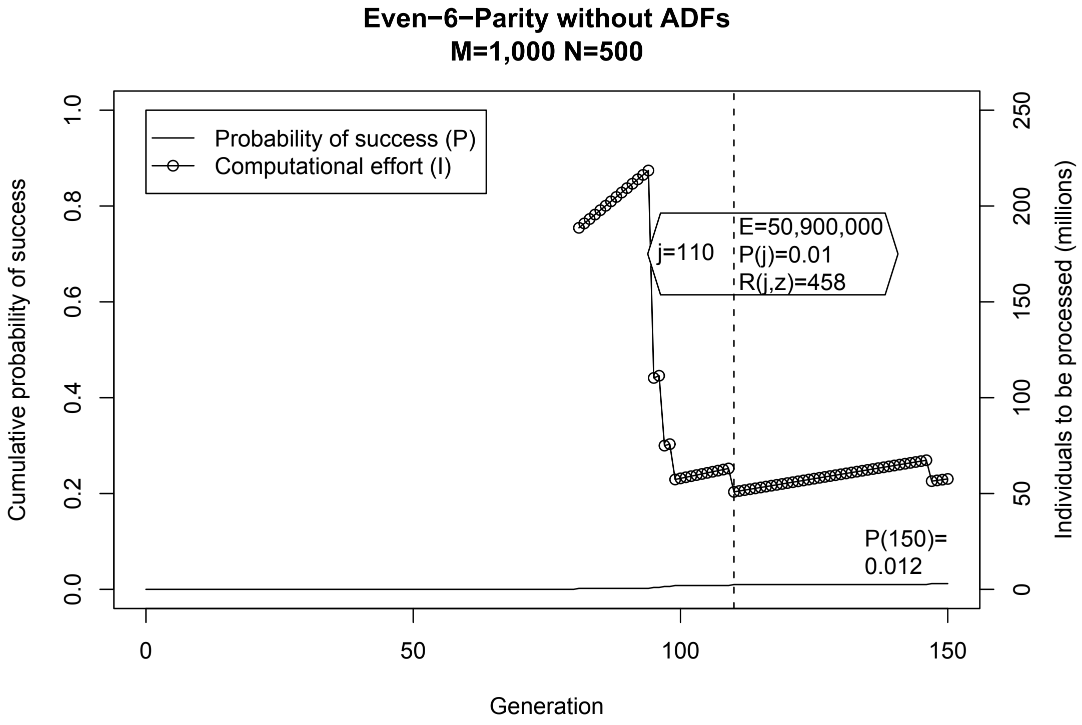


Figure 7.3: Computational effort and cumulative probability of success for even-6-parity without ADFs.

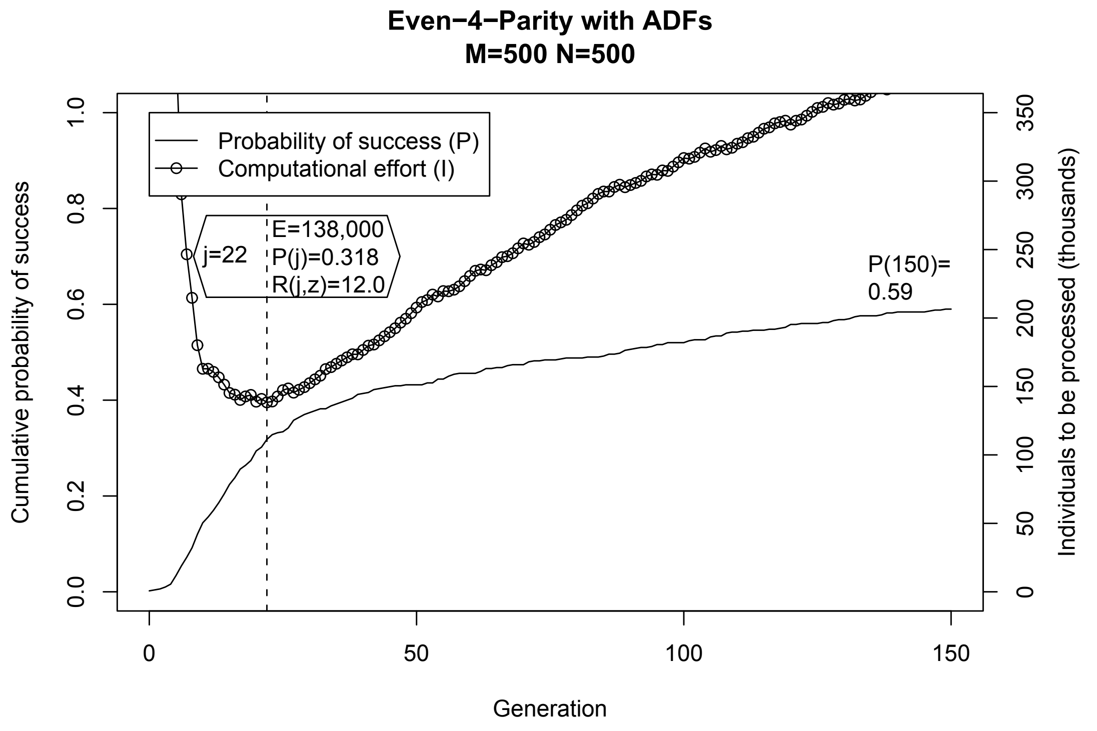


Figure 7.4: Computational effort and cumulative probability of success for even-4-parity with ADFs.

Even-4, and 5 completed 500 runs within their allocated time. Even-6 terminated just minutes before completion as it had used up its allocated seven days. Even-7 completed only 75 runs within seven days and, although it was allocated more time, a total of only 195 runs could be executed.

In comparison to the results without ADFs, all four of these experiments had a minimum generation well before the maximum of 150 generations. As a result the computational effort measures will likely be unaffected were the maximum generation to be lengthened.

7.5 Efficiency Ratios

In *Genetic Programming II* Koza analysed the efficiency gains when ADFs were introduced. His analysis offered an observation of the improvement in performance by calculating R_{E}: the minimum computational effort without ADFs divided by the minimum computational effort with ADFs (see section 3.4.3 for further discussion of this measure).

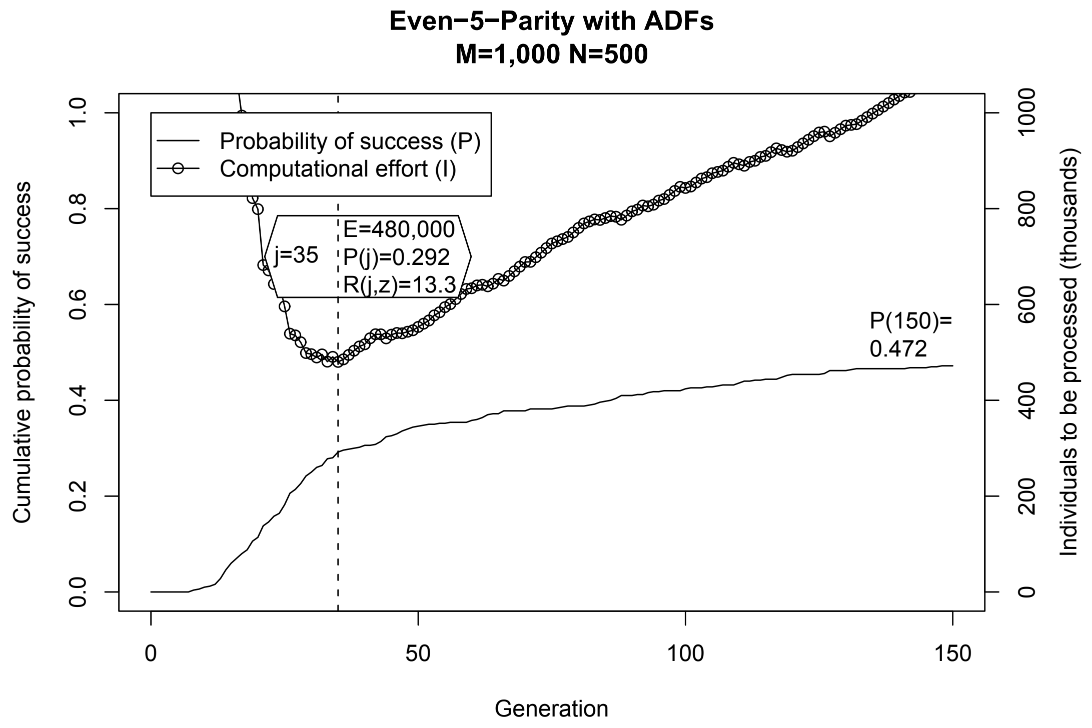


Figure 7.5: Computational effort and cumulative probability of success for even-5-parity with ADFs.

Problem	Min. Comp. Effort	Success Effort	Success Prop. $P(150)$
Even-4	138,000	142	59%
	(119,000–162,000)	(125–164)	(55–63)
Even-5	480,000	211	47%
	(409,000–566,000)	(184–244)	(43–52)
Even-6	800,000	404	30%
	(643,000–999,000)	(341–480)	(26–34)
Even-7	1,490,000	315	36%
	(1,070,000–2,090,000)	(247–405)	(30–43)

Table 7.3: Minimum computational effort, success effort, and final success proportion for the different problem sizes used in the experiments with ADFs. 95% confidence intervals are shown in parentheses.

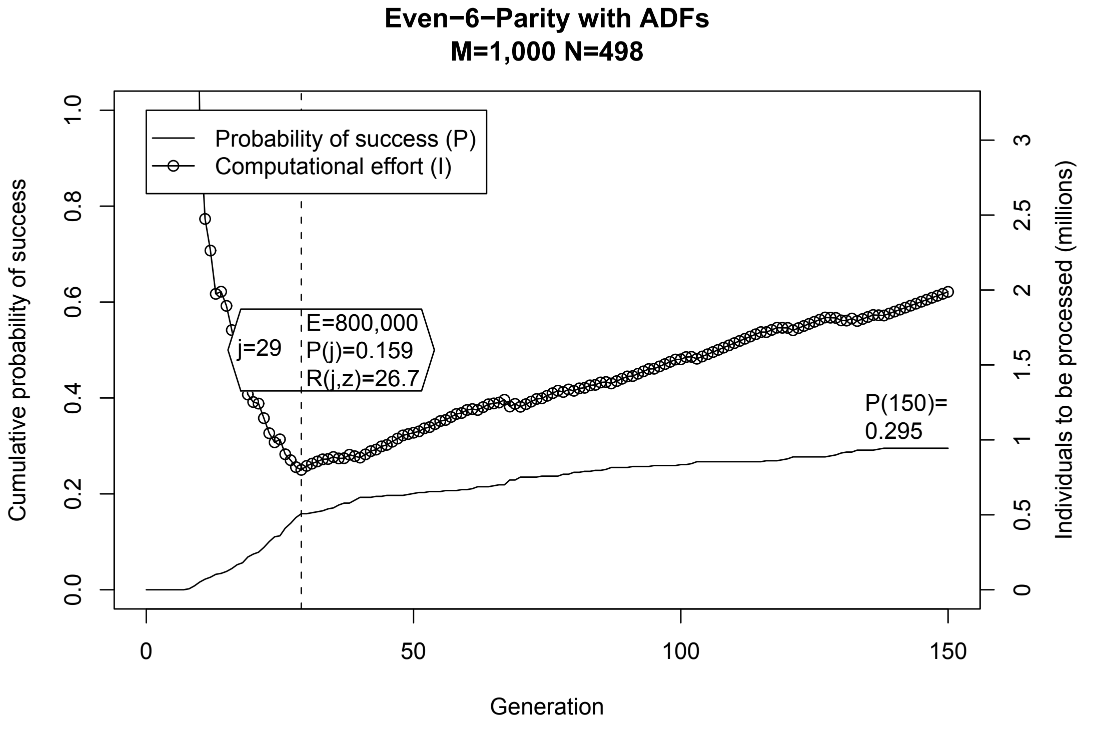


Figure 7.6: Computational effort and cumulative probability of success for even-6-parity with ADFs.

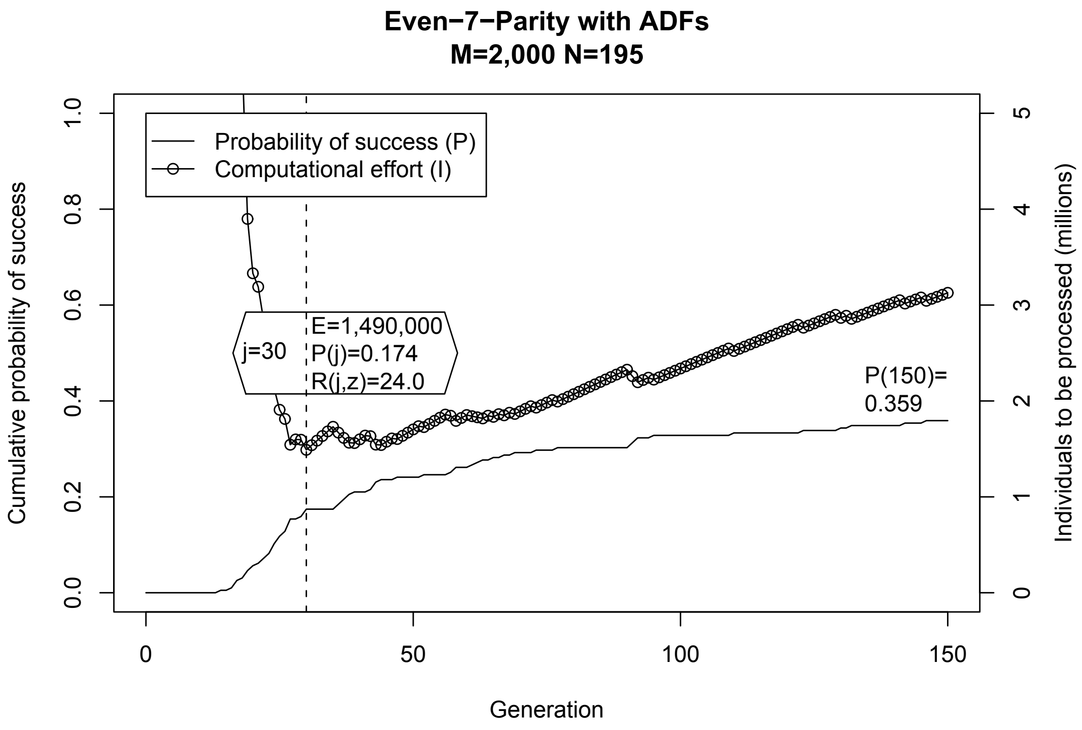


Figure 7.7: Computational effort and cumulative probability of success for even-7-parity with ADFs.

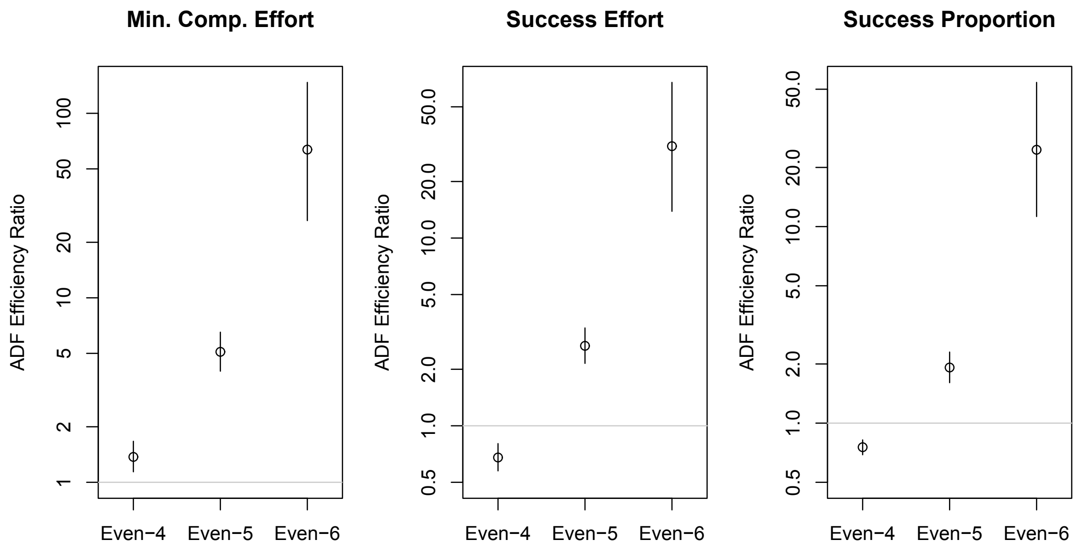


Figure 7.8: Efficiency ratio for the addition of ADFs calculated using minimum computational effort, success effort, and success proportion. 95% confidence intervals are included. The grey line is plotted at the break-even point of 1.0. Note the log scale.

Figure 7.8 replicates his analysis but with the data from this chapter. Unlike Koza we have used not only minimum computational effort, but also success effort and success proportion. Also unlike Koza, we have included confidence intervals for each of the ratios (calculated using the methods in tables 3.19, 4.7, and 2.3). Even-7-Parity was excluded given our lack of minimum computational effort and success effort measurements without ADFs, and that the success proportion ratio would include infinity.

The major feature of figure 7.8 is that, irrespective of measure, the advantage of using ADFs increases at least exponentially. This supports Koza's observation [72, page 192]. An interesting highlight however, is the unexpectedly poor "improvement" of even-4 with ADFs when success effort and success proportion are used in the ratio—an effect principally influenced by the higher final success proportion of even-4 without ADFs than with ADFs.

Problem	Min. Comp. Effort	Success Effort	Success Prop. $P(50)$
Even-4	138,000	83	43%
	(119,000–162,000)	(72–96)	(39–48)
Even-5	480,000	120	35%
	(409,000–566,000)	(104–139)	(31–39)
Even-6	800,000	223	20%
	(643,000–999,000)	(184–271)	(17–24)
Even-7	1,490,000	185	24%
	(1,070,000–2,090,000)	(141–245)	(19–31)

Table 7.4: Minimum computational effort, success effort, and final success proportion for the different problem sizes used in the experiments with ADFs. The maximum of 50 generations per run were executed. 95% confidence intervals are shown in parentheses.

7.6 Results to 50 Generations

Although the run lengths to 150 generations will be used in chapter 9, the traditional setting has been to run to just 50 generations. To allow for comparison with previous work, tables 7.4 and 7.5 give the results up to the more traditional limit. Chapter 8 will also make use of this data.

Note that successes were observed for neither even-6 nor even-7 when ADFs were not used. Without any successes values could be calculated for neither minimum computational effort nor success effort.

When ADFs were used, the observations of minimum computational effort do not differ between 50 generations (table 7.4) and 150 generations (table 7.3). This was because the minimum generation occurred before generation 50 for all four versions of even-n. On the other hand, success effort values were reduced when the experiments were considered at 50 generations primarily thanks to the reduction (by 100 generations) of the cost of failure.

Problem	Min. Comp. Effort	Success Effort	Success Prop. $P(50)$
Even-4	204,000 (179,000–234,000)	96 (85–109)	44% (40–48)
Even-5	7,280,000 (3.7–14.4 million)	3120 (1590–6040)	1.6% (0.8–3.1)
Even-6	no successes	no successes	0% (0–0.8%)
Even-7	no successes	no successes	0% (0–0.8%)

Table 7.5: Minimum computational effort, success effort, and final success proportion for the different problem sizes used in the experiments without ADFs. The maximum of 50 generations per run were executed. 95% confidence intervals are shown in parentheses.

Chapter 8

Manual Fitness-Based Incremental Evolution

This chapter considers a form of incremental evolution introduced by Jackson: fitness-based incremental evolution [67]. Its premise is that performance may improve if the evolutionary process is initially forced to focus on a small portion of the specified fitness cases. We consider this technique both with and without ADFs and compare the performance against direct evolution. A modification to the weighting of each generation is suggested.

8.1 Related Work

In fitness-based incremental evolution the stages are formed from subsets of the fitness cases of the goal stage. So, for example, if there are 100 fitness cases to be solved, in order to find a 100%-compliant solution, then we might consider evolving the population on the first 50 of these cases. Once a solution has been found (or an allocated number of generations has been spent) the individuals are then evolved with evaluation occurring over the full set of fitness cases.

Barlow used a very similar approach when evolving unmanned aerial vehicles (UAVs) [14]. The first 200 of 600 generations focused solely on only one of the four fitness functions. Barlow reported that the technique "works well and makes a great deal of sense" [14, page 58].

The concept is not limited to a two-stage process. It is conceivable that there could be performance benefits to be gained by dividing the total number of generations into three (or more) stages. For example, given a goal of 100 cases, evolution could be focused initially on the first 25 fitness cases, then, once solved, it could move on to 50 and then finally the full 100 cases.

Neither is the concept limited to an entirely linear approach: Jackson tried a three stage approach with the first stage starting with a random initial population and focused on the first eight of the sixteen cases of even-4-parity. The second stage also started with an initial random population but was focused on the second half of the sixteen cases. The third stage combined the final populations of the first two stages and then evolved the population on all sixteen cases.

Jackson's work showed a very slight improvement in performance on even-4- and even-5-parity when compared to genetic programming without ADFs. The performance of his experiments did not compare well against GP with ADFs. However, my analysis (see figure 8.1; using the method offered in table 2.3 of this thesis) showed that most of his results were not statistically significant.

It seems an apparent flaw that Jackson failed to study incremental evolution with ADFs. It appears he considered the two ideas to be in competition. Gustafson however discussed the combination of ADFs and incremental evolution in his thesis [56]. Hsu and Gustafson later studied the two further but decided the addition of ADFs appeared not to be beneficial [61, page 7]. In this chapter we study the addition of ADFs to incremental evolution.

However, Jackson's focus—for his first paper on fitness-based incremental evolution [67]—was on a technique he termed saturation. He later considered a technique based on parameterless functions [65], while his third article on the subject [66] returned to the topic of incremental evolution based on simplified problems. We will summarise each in the following three sections.

8.1.1 Saturation

Jackson's initial study focused on the effect of varying *saturation* levels. Having noted that "programs that pass all four test cases [of the first stage] were often found in the initial population", he concluded that, as a consequence, the second stage's starting population was "often very similar to that which would have been obtained [had incremental evolution not been used]", and that, as a result, "performance hardly differed". His solution was to let the evolutionary process continue until a specified percentage of the population was made up of 100%-correct individuals—the saturation level.

Figure 8.1 plots the efficiency ratio of fitness-based incremental evolution from the final success proportion[1] data given in Jackson's work [67].

[1] Although minimum computational effort measures were given in the paper, insufficient information was provided for the production of confidence intervals.

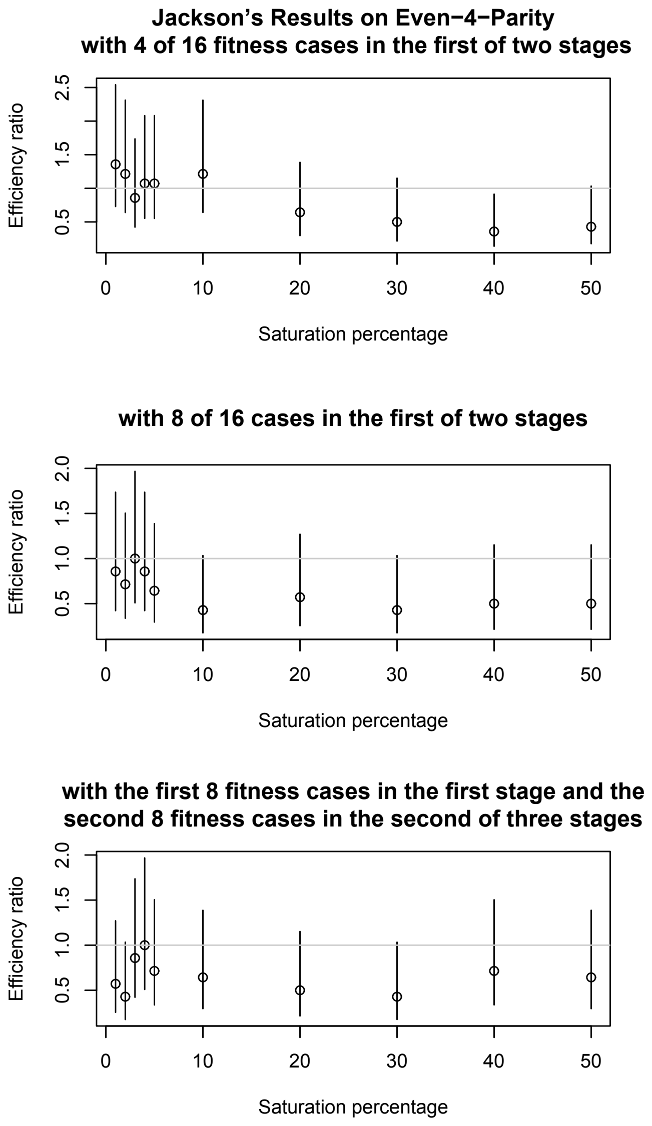


Figure 8.1: Jackson's results [67] comparing the efficiency of using fitness-based incremental evolution to direct evolution on the even-4-parity problem given varying saturation levels in the initial stages. Neither technique used ADFs. Ratios are based on quoted final success proportions. 95% confidence intervals are included; calculated using the method in table 2.3) from run sizes and observed success data provided by Jackson. The grey line is plotted at the break-even point of 1.0.

Although his initial fitness-based incremental evolution study was only of even-4, we can draw some conclusions about the usefulness of saturation. Firstly, there are two general trends.

The first general trend is that the vast majority of experiments produced results that were below the easier of the two bars worth achieving. Canonical (standard) GP without ADFs had superior performance in 24 of the 30 experiments. The hope that one might hold for this technique would most likely be pinned on an increase in relative performance for more difficult problems. Jackson however does not consider this in his paper. We address this topic (among others) in the following sections.

The second general trend is that reducing the saturation level increases performance. Although this fact is demonstrated by the trends in figure 8.1, it is more powerful to consider the raw measures (without comparison to the base-line). Unfortunately, even then the majority of results are statistically indistinguishable: at 95% confidence, the top result is unable to be separated from, at the very least, saturation rates of 1 to 10%. From this evidence it would seem that only very small saturation rates should be considered.

A concerning issue with Jackson's methodology was whether the target saturation rates were ever actually attained. It was not made clear, given a fixed limit of 15 generations in stage one, how saturation levels of up to 50% could ever have been achieved. However the concern is actually of no practical significance given the fact that high levels of saturation were shown to have a negative impact on performance.

The original motivation for the idea does not hold water when the difficulty of the initial stage is increased. My experiments showed that, with eight of the sixteen fitness cases, the initial stage took a number of generations before finding solutions to all its allocated fitness cases (compare the performance in figure 8.2 with the result in figure 8.3 that 35% of the initial population represented a solution of the first four fitness cases).

One might claim that the motivation for saturation should really be to drive the entire population into a genetic space that allows for even more success in the next stage. This however is the motivation of incremental evolution. So perhaps just as important would be to consider other, potentially less costly, techniques such as duplicating the best individuals so that they are over represented in the initial population of the next stage, or how the intermediate stages are chosen. The latter of these two ideas are considered in this chapter.

So with both a reduction in performance and a motivation that doesn't stand

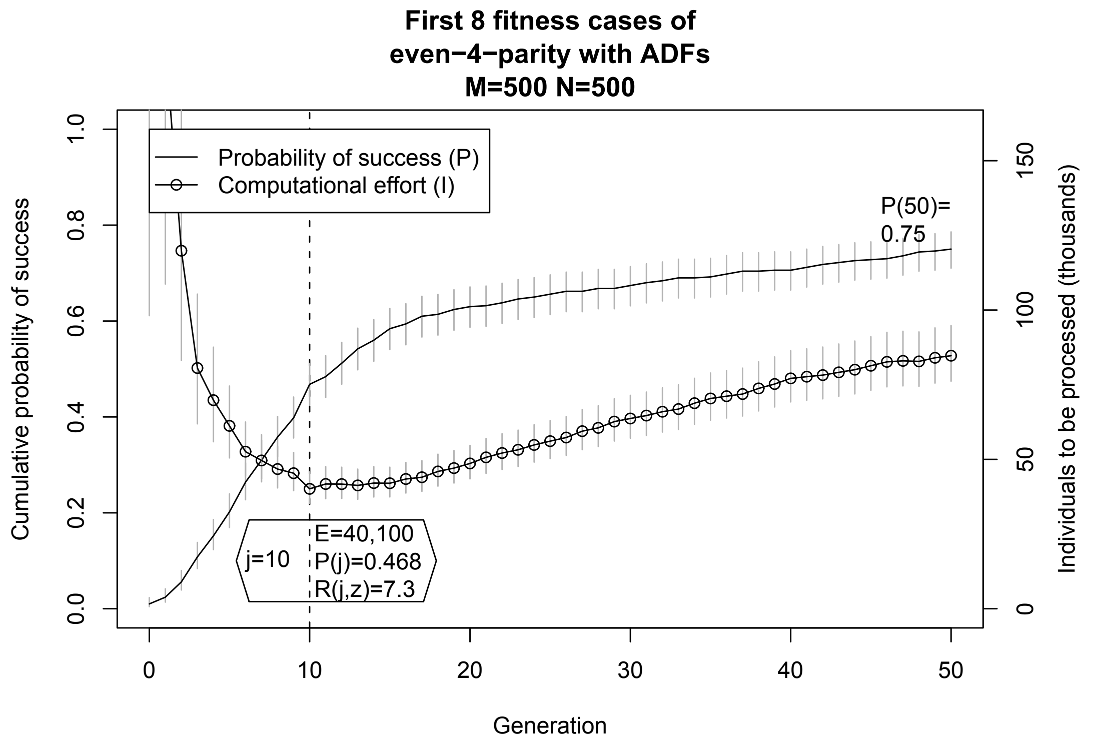


Figure 8.2: Computational effort and success proportion curves for the first eight of the 16 fitness cases of even-4-parity. A solution in this instance is a score of eight out of the eight fitness cases.

up, one might consider the idea of saturation to be dead. However, possibly the most telling of evidence is Jackson's lack of use of saturation levels in either of his second or third studies on the topic [65, 66]—he even labelled his own technique "disappointing" [65].

8.1.2 Parameterless Functions

Jackson developed his fitness-based incremental evolution ideas in a second paper where he considered the concept of *parameterless functions* (PFs) [65].

In this study, a user would specify a number of subsets. The total number of fitness cases would be divided into the given number of subsets and the evolutionary process would be fed one subset at a time. Evolution would move to the next subset (and stop work on the current) only after it had found a solution for all the fitness cases in the current subset; each of the evolved subset solutions was termed a parameterless function. Once solutions had been found for every subset, the evolutionary process would then be given the job of producing code to classify the inputs and call the appropriate parameterless function.

Using this technique on the even-4-parity problem, Jackson was able to improve on not only canonical GP, but also on GP with ADFs. Again, by analysing his final success proportion measures, we can obtain the following statistics:

- Dividing the fitness cases into two subsets of eight cases each (thus producing two parameterless functions) gave an observed efficiency ratio of 3.9 over canonical GP and 1.3 over GP with ADFs. Using the method in table 2.3 we can say, with 95% confidence, that the true efficiency ratios are at least 2.6 and 1.0 respectively.

- Dividing the fitness cases into four subsets of four cases each (thus producing four parameterless functions) gave an observed efficiency ratio of 4.2 over canonical GP and 1.4 over GP with ADFs. With 95% confidence we can say that the true efficiency ratios are at least 2.8 and 1.1 respectively.

However, this trend did not entirely continue with even-5-parity:

- Dividing the 32 fitness cases into four subsets of eight cases each (thus producing four parameterless functions) gave an observed difference of 58 percentage points over canonical GP's 0% success rate. With 95% confidence we can say that the true difference is at least 49 percentage points. Against GP with ADFs, an observed efficiency ratio of 1.8 was measured. Again, with 95% confidence, this ratio is at least 1.4.

- Dividing the fitness cases into eight subsets of four cases each (thus producing eight parameterless functions) gave an observed difference of ten percentage points (again against the 0% measure). With 95% confidence, the difference is at least five percentage points. An efficiency ratio of 0.3 was observed against GP with ADFs. With 95% confidence we can say that, with this setup, GP with ADFs is at least 1.9 times more likely to find a solution than Jackson's parameterless-functions technique.

The reason for this poor performance was that, although the code for the parameterless functions would evolve very quickly—sometimes even within the initial generation for subsets with just four cases—the final evolutionary task of classifying the inputs and calling the appropriate parameterless function was very difficult. Its increased difficulty was said to be due to an increase in the size of the set of terminal primitives as each parameterless function was represented as an additional terminal.

Koza studied the impact of extraneous variables in chapter 24 of his first book on genetic programming [71]. Although increasing the terminal set with extraneous variables is not the same as increasing it with valuable terminals (such as the evolved parameterless functions), we can still note that the detrimental effect was large. One would expect, if GP were not having to ignore the extras but instead put them in their correct position, then the performance decrease could be larger still. Jackson's results give evidence to this.

Jackson's technique for avoiding this problem is worth discussion: he hand coded a "select" function such that his final evolutionary step was no longer necessary. GP was still used to evolve solutions to the subsets of the fitness cases, and so still produced parameterless functions, but the "main" program became a basic "switch" statement that routed a given input to the appropriate parameterless function.

The technique that Jackson offered is not generally useful because the hand-coded select function is doing all the interesting work. Most real-world classification problems are indeed focused on how to produce the "select function". Further, Jackson's technique just would not cope with scenarios it had not seen in training, another key element of real-world problems.

8.1.3 Incremental Evolution with Simplified Problems

Another technique, first introduced by Naemura et al. [89], was also studied by Jackson: incremental evolution where the early stages are evaluated on a simplified version of the goal problem. Both the Japanese group and Jackson studied the technique on even-n-parity.

Naemura et al. found that they could beat the performance of GP with ADFs if even-3 was used as an initial stage for an even-4-parity goal. An efficiency ratio (based on their final success proportions) of approximately 1.2 was observed and my analysis of their results show that, with 95% confidence, there was at least a slight improvement in performance. They also used even-3 as an initial stage for a goal of even-6 and demonstrated an efficiency ratio of three (95% confidence of at least 1.6). Finally they experimented with a three-stage setup: the first was evaluated on even-3, the second on even-6 and the final, goal stage was even-9. 40% of their 30 runs found a solution compared to zero when GP with ADFs were used. It is unclear, but it appears that the cost of the non-goal stages were not included in their measures. Although that made for an unfair comparison, Jackson repeated this technique explicitly including the cost of executing the initial stages.

On the even-4, -5, and -6 problems, Jackson showed an efficiency ratio at 95% confidence of *at least* 1.5 [67]. His experiments used either even-2 or even-3 as the initial stage. Similar results were obtained on the majority-on problem domains with five and seven inputs using three inputs as an initial stage [67]. Even greater improvements were demonstrated on the two- and three-bit half- and full-adders in Jackson's third paper on incremental evolution [66].

8.1.4 Summary

Jackson looked at three topics of incremental evolution: fitness-based incremental evolution, parameterless functions, and incremental evolution with simplified problems.

Jackson's hand-coded "select" function and evolved parameterless functions were not considered a reasonable approach for the development of GP; it would be more effective to use a lookup table.

On the other hand his use of simplified problems in the intermediate stages was very successful and holds significant potential benefit for the scaling up of GP to more difficult problems. However, it relies on the user's ability to usefully simplify the problem.

Fitness-based incremental evolution, although unsuccessful in Jackson's study, holds the most general potential benefits. There are a vast number of problem domains that lend themselves to this technique and if we understood how to ensure the technique was beneficial, it could be even more useful than the use of simplified problems.

8.2 Method

In this chapter we will use the methods developed in Part I to take a more thorough look at Jackson's original form of fitness-based incremental evolution. Given the evidence, we avoided his saturation suggestion and instead moved from one stage to the next immediately after a solution was found.

We looked at five decisions that may have an impact on the performance of fitness-based incremental evolution:

- the allocation of generations to each stage,
- the number of fitness cases in stage one,
- the addition of ADFs,

- the difficulty of the problem domain,
- and the weighting placed on the cost of a generation.

For this study we experimented with the even-4, -5, -6, and -7 problems up to 50 generations. The generations were allocated between two stages, always totalling 50 generations. The first stage evaluated the population on a subset of the full set of fitness cases. If a solution to the subset of cases was found in the first stage then the second stage would begin immediately and the unused generations from the first stage's allocation would be available to the second stage. If a solution was not found in the allocated number of generations then the first stage would terminate and then the second stage would begin. The final population from the first stage was used unchanged in the second. The second stage always evaluated the population on the full set of fitness cases for the given problem. If ADFs were used then they were configured in the same way as the direct evolution runs (see table 7.1). The minor parameters were specified in the same way as was done with direct evolution (see section 7.2).

The even-4 problem was considered with a first stage of either the first four or the first eight of the sixteen fitness cases. When four fitness cases were used in the first stage, the probability of solving that subset within a few generations was very high (see figure 8.3), as a result ten experiments were executed with the first stage allocated one to ten generations. When eight fitness cases were used in the first stage, ten experiments were executed, with the first stage allocated from 5 through to 50 generations in steps of 5.

For the even-5, -6, and -7 problems, ten experiments were executed with the first stage allocated from 5 to 50 generations in steps of 5. For even-5, eight and 16 fitness cases (of the 32) were used in the first stage. For even-6, eight, 16 and 32 fitness cases (of the 64) were used in the first stage. For even-7, 16, 32, and 64 fitness cases (of the 128) were used in the first stage.

The fitness cases in the first stage were selected in the same order that Jackson specified; they represented the lower-order even-parity problems. Thus, for even-4-parity, when the first stage consisted of the first four fitness cases, the inputs were `0000`, `0001`, `0010`, `0011`. When the first stage used eight fitness cases, the inputs were `0000`, `0001`, `0010`, `0011`, `0100`, `0101`, `0110`, `0111`.

Even-4, -5, and -6 experiments were replicated with and without ADFs. Even-7 was executed only with ADFs (performance of even-5 and -6 without ADFs was so poor we held no hope for even-7).

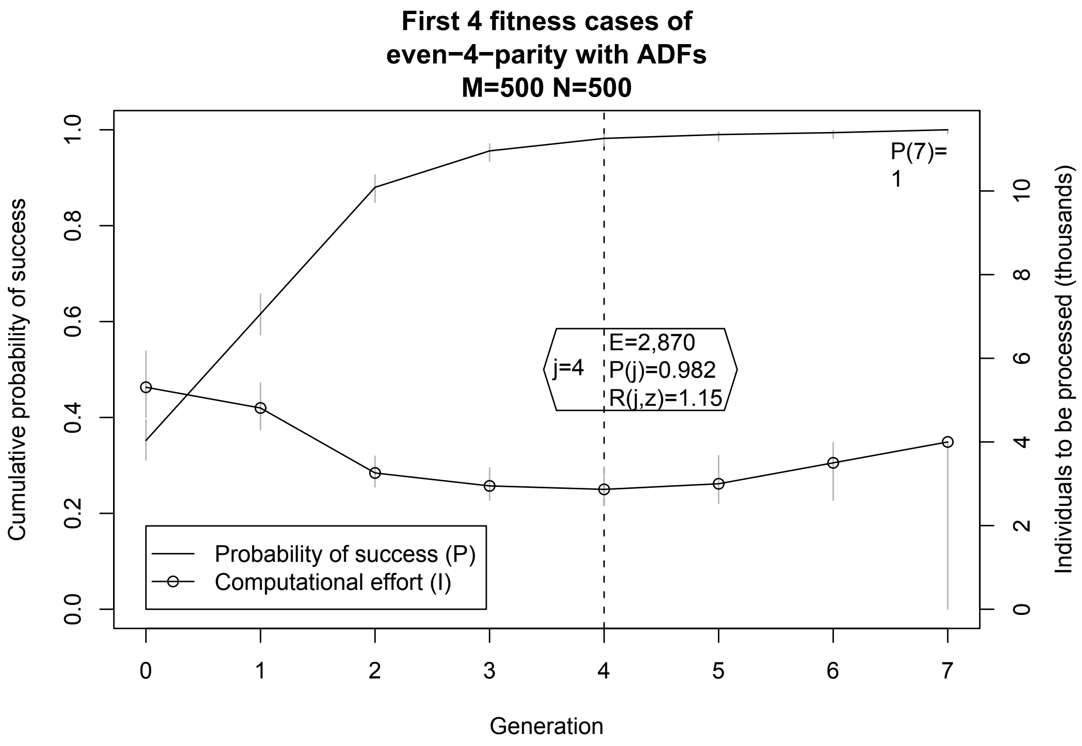


Figure 8.3: Computational effort and success proportion curves for the first four of the 16 fitness cases of even-4-parity. A solution in this instance is a score of four out of the four fitness cases.

A total of 17 experimental configurations were considered. For each configuration ten variations were used in the number of generations allocated to stage one, giving a total of 170 experiments. All experiments were executed to 500 runs.

8.3 Results: Unit-Cost Generations

This section discusses the results of the experiments. We will use an underlying assumption that a generation in the first stage is of equivalent cost to a generation in stage two. When others have made similar comparisons they have either used this technique, or they have failed to consider the cost of the initial stage at all.

Results in this section do not show the technique in a positive light. Section 8.5 however will look at another approach where the costs are proportional to the amount of work done and fitness-based incremental evolution is shown to have more potential.

Appendix D points the interested reader to an electronic form of the more than 500 graphs that were analysed for this section; the raw results of all 85,000 runs are also available.

8.3.1 Addition of ADFs

An issue with the parameter settings suggested by Jackson is that his success rates without ADFs were very low. Using direct evolution on even-4-parity, Jackson reported 14% success and with even-5 a rock-bottom 0% [67]. On the other hand, performance with ADFs was very reasonable. Chapter 7 has just demonstrated similar performance to Jackson's results when the maximum number of generations was limited to 50.

Given poor results without ADFs, what happens when incremental evolution is used *with* ADFs? Jackson offered his technique as an alternative to Koza's Automatically Defined Functions and to Angeline and Pollack's Module Acquisition (MA), but there is no reason why they should not be combined. Indeed it is possible that the ADF (or MA) code encapsulation mechanism may allow for an improved transition from one stage to the next. If so, performance improvements should be seen.

Incremental evolution did indeed benefit from the use of ADFs. In fact, without ADFs the population sizes used would not have been feasible. Direct evolution did not fare well without ADFs and at even-6 none of the 500 runs found a solution. However, as we will discuss now, the use of ADFs may well be a benefit to incremental evolution.

To study the benefits of the addition of ADFs we could compare incremental evolution with ADFs against incremental evolution without ADFs. However, were we to do that the predominant effect that we would measure would be the benefit of ADFs. It is already known that ADFs improve performance on this problem domain [72] so we are not interested in that. Instead we are interested in the influence of incremental evolution given the addition of ADFs. To measure that we need to subtract the standard impact of adding ADFs and that is done by: (a) measuring the relative performance of incremental evolution with ADFs against standard GP with ADFs, then (b) measuring the relative performance of incremental evolution without ADFs against standard GP without ADFs, and then (c) measuring the relative performance between (a) and (b). So, to make this comparison we need an interesting measure: an efficiency ratio of two efficiency ratios.

Theoretical confidence intervals for such a measure are very unlikely to exist. Fortunately however, we can make the most of computer simulations. Table 8.1 gives the algorithm used.

Unfortunately we could only execute this analysis for even-4 and -5 given that, without ADFs, 500 runs of direct evolution failed to find solutions for even-6 or -7 (see table 7.2). However, even this limited view is interesting.

For both even-4 with four fitness cases in the first stage, and even-4 with eight fitness cases, there was very little reason to even suggest a difference in performance. This indicated that fitness-based incremental evolution has no impact on the performance improvements offered by ADFs.

However for even-5, for both 8 fitness cases and 16 fitness cases in the first stage, the results indicated that incremental evolution may benefit from the addition of ADFs.[2] Such a conclusion should be taken with concern for the confidence intervals, all of which include the possibility that there is no impact at all, even for low confidence levels. However all the point estimates are well above one and, further, the confidence levels are heavily impacted by the extremely low levels of success observed under incremental evolution without ADFs.

What can we take from this? That the addition of ADFs was, at the very least, not detrimental, and indeed may even have had a positive impact on fitness-based

[2] The results should really be said to indicate that the addition of incremental evolution to GP with ADFs may *reduce performance less* than the addition of incremental evolution to GP without ADFs.

1. Obtain experimental results for:
   (a) incremental evolution with ADFs
   (b) standard GP with ADFs
   (c) incremental evolution without ADFs
   (d) standard GP without ADFs
2. For each of the four experimental results, produce B random numbers using either the algorithm in table 3.19 (for minimum computational effort), table 4.7 (for success effort) or table 2.3 (for success proportion).
3. Using two of the four sets of B random numbers, obtain B simulated efficiency ratios for the performance of incremental evolution with ADFs against standard GP with ADFs. Label these R_1.
4. Using the remaining two of the four sets of B random numbers, obtain B simulated efficiency ratios for the performance of incremental evolution without ADFs against standard GP without ADFs. Label these R_2.
5. Find the $\frac{\alpha}{2}$ and $1 - \frac{\alpha}{2}$ quantiles of $\frac{R_1}{R_2}$. These provide an upper and lower range for a $1 - \alpha$ confidence interval.

Table 8.1: Algorithm for the efficiency ratio of two efficiency ratios.

incremental evolution. ADFs produced an improvement in performance that was beyond what would normally be expected. It is likely that this is due to the code encapsulation offered by ADFs. Koza demonstrated that the automatically defined functions could break the problem into parts. Incremental evolution must re-write its "main" code to solve the second stage, so the introduction of ADFs may allow it to more easily keep and then manipulate a larger amount of what it learnt from stage one.

The remainder of this discussion will focus on the results where this beneficial addition has been used.

8.3.2 Allocation of Generations

The intention of this section was to show how one might select the optimal number of generations to allocate for stage one. However, the trend that stands out the most, is that this technique isn't any good.

When we examine the minimum computational effort, success effort, and final success proportions for these experiments, the general trend is that performance decreases as the number of generations allocated to stage one increases. This trend occurs irrespective of performance measure, irrespective of the use of ADFs, irrespective of the difficulty of problem, and irrespective of the number of fitness cases in stage one. Figure 8.4 plots a typical example.

For every configuration (excluding two) the general trend looked to show that the best choice was to allocate five generations to stage one. This however, should be considered with the fact that "five" was the lowest allocation we experimented with. It is more than likely that a lower allocation would show still further benefit—to the extreme of zero allocated generations where the method does not differ from direct evolution. Although we have no evidence to defend that hypothesis, at the very least one can say that allocating many generations to stage one decreases performance.

The two exclusions to the previous paragraph should be considered in the realm of statistically insignificant, especially given the contexts of the other results, however:

- Even-5-parity with ADFs and eight fitness cases in the first stage showed a very slight upward peak at 20 allocated generations. The general trend however was still visible.
- Even-4-parity without ADFs and with four fitness cases in the first generation had an interesting step-wise pattern with a period of three generations; the average performance remained constant.

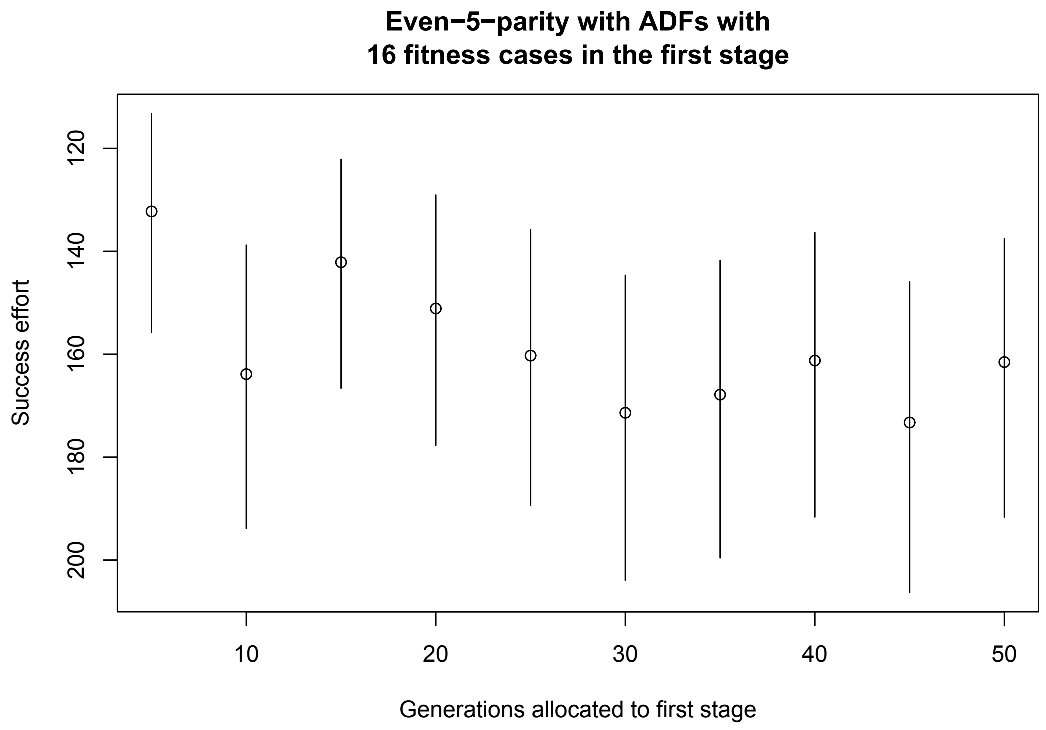


Figure 8.4: An example of a typical performance curve when the generations in stage one are considered of equal cost to those in stage two.

There was no noteworthy difference between the trends shown in the minimum computational effort curves when compared to the success effort curves. In fact the two measures had an average correlation of 0.91 (using Spearman's rank method [24][3]). For two distinct measures that is a very high correlation. Section 8.5.2 discusses this further.

Convergence Concerns

In attempting to understand the reason why performance was so poor, I separately considered the two routes to success. Success in the second stage (as measured by final success proportion) can occur either through success in the first stage or through failure in the first stage.

Plotting the proportion of stage one successes given success in stage two (as compared to failures in stage one given success in stage two), showed that the route "through failure" was only a possible option below about 20 allocated generations (see the left graph of figure 8.5 for an example). After that point practically 100% of successful runs followed the route "through success". (Even-6 had a higher limit of about 30 allocated generations and it depended on the number of fitness cases in stage one, but the general trend was the same.)

This trend is intuitive. The intuition is that the route "through failure" closes down because insufficient generations remain to allow success in stage two. One assumes that the failure in stage one means genetic programming must start from scratch and that it has insufficient time to succeed given the time used by stage one.

However, although it is intuitive that this trend should occur, it is interesting to consider this trend with the number of generations used in stage two when success in stage two was observed. Going with the intuition, one would expect that (at least) the average number of generations used in successes in stage two would be greater than the number of generations remaining, when the route "through failure" was no longer an option. This however was mostly not the case.

The graphs in figure 8.5 give an example of this phenomenon. How could this be explained?

One concerning possibility is that of genetic convergence—a reduction in diversity to the point that the second stage cannot be solved [35]. It is possible that,

[3]Spearman's rank method has a range of -1 to 1. Two sets of 100 random numbers uniformly distributed between 0 and 1 have an expected correlation of 0, but with a 95% confidence interval of ± 0.2.

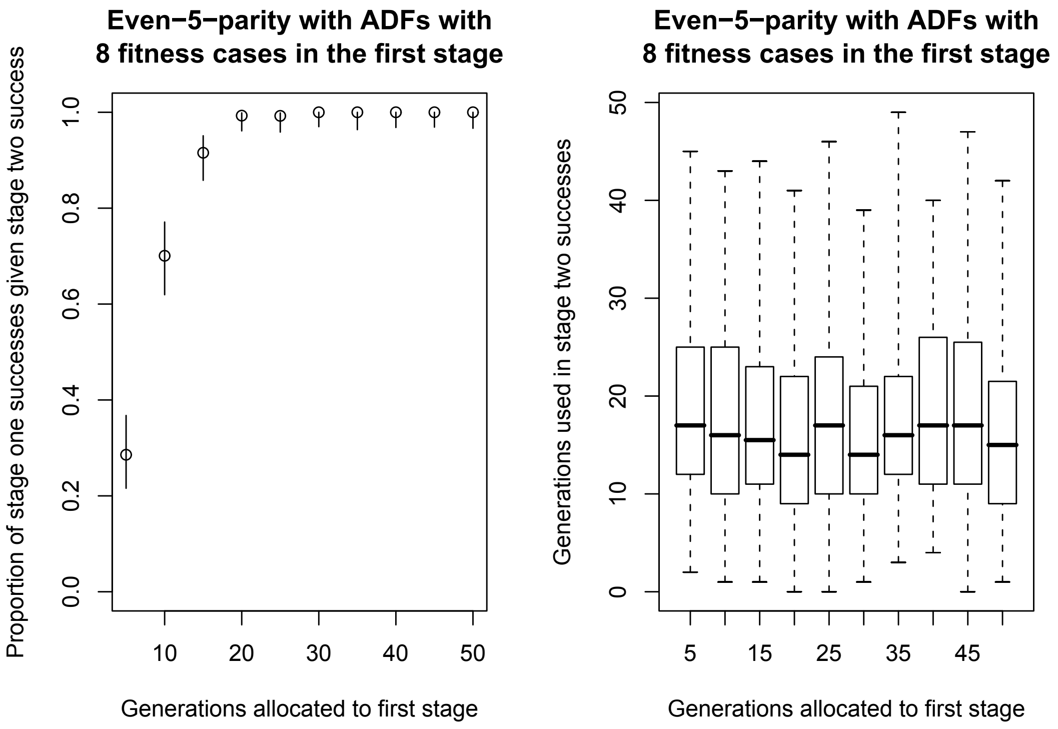


Figure 8.5: An example of the two graphs that point to potential problematic convergence during stage one.

in attempting to solve the partial problem during stage one, the genetic material required for solving stage two had been eliminated from the genetic material in the population. In the case of even-4-parity where 16 fitness cases are used in the first stage, the final input primitive (IN3) is not required for a 100%-correct solution in stage one. A consequence is that selective pressure may eliminate all occurrences from the population.

This issue of convergence is a serious concern for incremental evolution. Were stage one to be executed either (i) over a large number of generations or (ii) with a small population, then the problem becomes an increasingly likely explanation for the closure of the route "through failure".

Fortunately, we can test whether this is a likely problem. By looping through each of the individuals and through all the nodes in every individuals' trees we can count the number of instances of each primitive in the population. Although, in general, we will not know if a primitive provided in the function or terminal sets is required for a solution in stage two, we might pay attention to any primitive with exceedingly low counts relative to the other counts.

For these experiments however we do know what is required in order to find a solution in stage two. The critical difference is in the terminal set and whether all the input primitives (IN0 through to IN$(n-1)$ for even-n-parity) are available.

We re-executed two configurations, counting the number of primitives at the termination of stage one. The configurations were even-4-parity without ADFs with eight fitness cases in the first stage, and even-5-parity with ADFs with eight fitness cases in the first stage. Both configurations used 20 generations in the first stage and both were executed to 500 runs.

For the even-4 configuration, we compared the proportion of IN3 primitives in the runs where stage two failed against the same proportion where stage two succeeded. The proportion was calculated as the number of IN3 primitives divided by the sum of the counts of IN0 through to IN3 (inclusive). A box-plot graphing this comparison can be seen at the top of figure 8.6. The plot shows that the two proportions are very similar. It is unlikely that the number of IN3 nodes caused the closure of the route "through failure",

For the even-5-parity configuration, we compared both the proportion of IN3 and IN4 primitives. The proportion was calculated as the number of IN3 primitives (or IN4 primitives) divided by the sum of the counts of IN0 through to IN4 (inclusive). The lower two plots in figure 8.6 show how these two proportions differed depending on whether stage one succeeded or failed. There is little difference between the two, so once again it is unlikely that this is the primary explanation for the lack of stage two successes given failure in stage one.

For all three plots the lower quartile, median, and upper quartile of the proportion of the required primitives is higher for the route "through success" compared to the route "through failure". This does give evidence that convergence may have a slight impact on this phenomenon, but it is important to note that the route was almost completely closed down. The size of the differences shown in figure 8.6 do not point to such an extreme consequence. It is sufficient indication that convergence is not likely to be the primary cause of concern. If it were the cause, it could be tackled with the use of mutation. This approach is considered in the next chapter.

However, if it was not convergence, what then was the cause of the closure of the route "through failure"? Somehow the evolutionary process of stage one guided the population into an area such that if it did not solve stage one then it had a severely reduced chance of solving stage two. How this occurred remains an open question.

8.3.3 Fitness Cases in Stage One

The objective of this section is to study the best number of fitness cases for the first stage. Given that all the experimental configurations showed a trend of increased performance as they neared direct evolution, it isn't meaningful to compare the performance within a specific problem and across the number of fitness cases using the "best" allocation of generations. However, for even-5, -6, and -7 we can make a comparison that obtains a similar result by fixing the comparison against the same number of generations allocated to stage one.

If we do this, then we produce 30 efficiency ratios for each comparison (ten for each of the three measures with one for each number of generations allocated to stage one). For even-5 we may make only one comparison: between whether it was better to have 8 or 16 fitness cases in the first stage. For even-6 and even-7 we can make three comparisons each.

Given the lack of success, these comparisons were not worth considering for the experiments without ADFs. However, for the experiments with ADFs some comments can be made.

None of the comparisons showed any trend across the number of allocated generations. That is to say there is no obvious trend that might indicate relative performance is influenced by the number of allocated generations. As a consequence it is fair to observe the average of the efficiency ratios. We can also observe the average of our confidence that the ratio is greater than one.

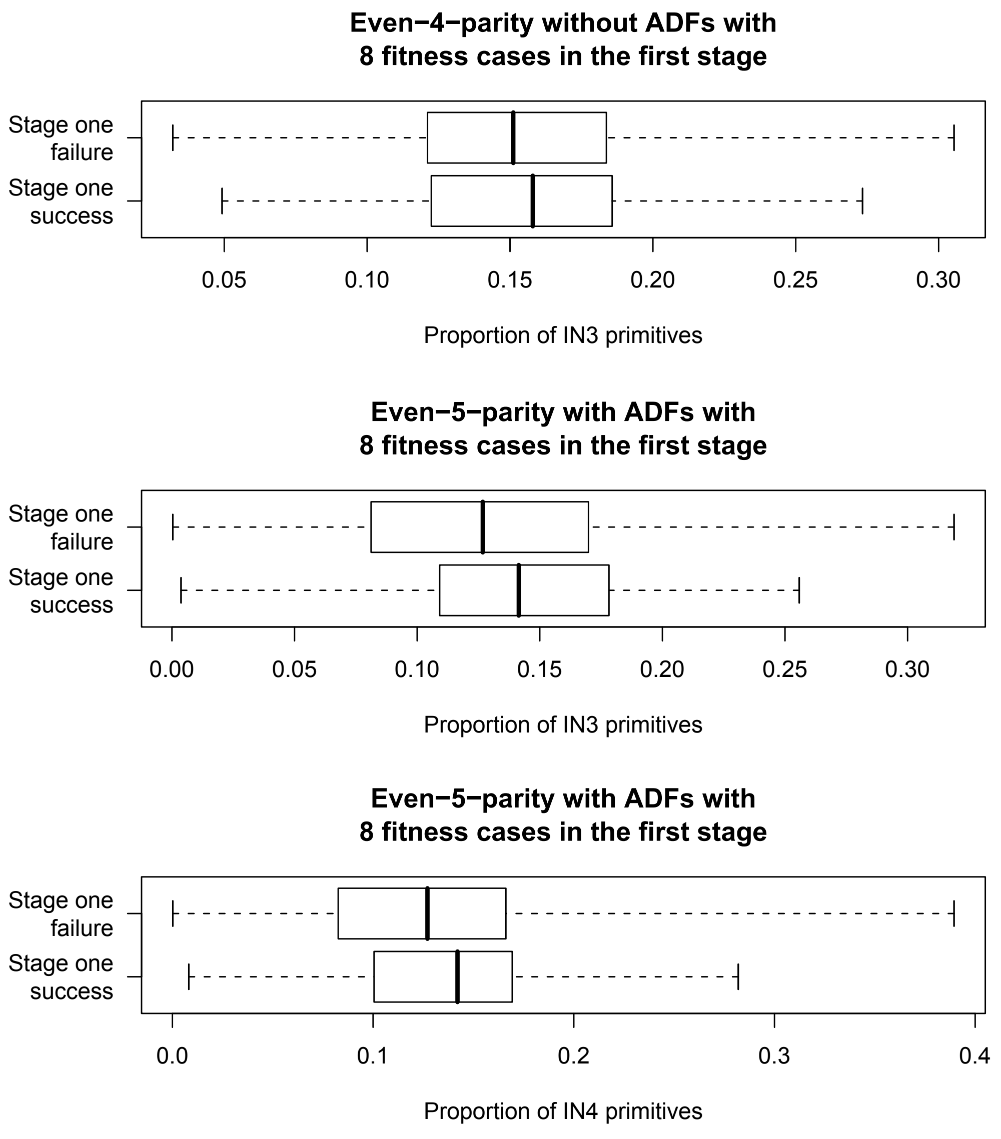


Figure 8.6: Evidence that elimination of required genetic material is likely not a significant factor in the closure of the route “though failure”.

Measure	Avg. ratio	Avg. conf.
Min. comp. effort	1.05	60%
Success effort	1.11	72%
Success Proportion	1.09	73%

Table 8.2:
Efficiency ratios comparing 16 fitness to 8 fitness cases for even-5-parity with ADFs, averaged over the ten values for allocated generations. (Note 16 fitness cases outperformed 8 cases.) Also stated is the average confidence that the ratio is greater than one.

Efficiency of x fitness cases	compared to y fitness cases	Avg. ratio	Avg. conf.
16	8	0.94	40%
32	8	1.22	76%
32	16	1.29	81%

Table 8.3:
Efficiency ratios of even-6-parity with ADFs, averaged over the three measures and ten allocated generations. Also stated is the average confidence that the ratio is greater than one.

For even-5 we can compare whether it was more efficient to use 8 or 16 fitness cases in the first stage. For all three measures the use of 16 fitness cases was, on average, a more efficient choice. Table 8.2 gives the efficiency ratios for the use of 16 versus 8 fitness cases.

For even-6 we can compare whether it was more efficient to use 8, 16, or 32 fitness cases in the first stage. All three measures pointed in the same direction: 16 fitness cases required the most effort; 8 fitness cases was not far ahead; but the most efficient, by up to 30%, was 32 fitness cases. Table 8.3 gives more detail.

For even-7 we can compare which of 16, 32, or 64 fitness cases in the first stage was the most efficient. There was very little difference between 16 and 32 cases, but 64 showed a notable reduction in the effort required. Table 8.4 gives the detail.

From these results, it appears there is a general trend towards increased efficiency as the number of fitness cases in the first stage increases. This is a natural extension of the result in the previous section: if incremental evolution is not a good idea then the further away from it the better. Larger numbers of fitness cases in the first stage is one way of getting closer to the more efficient direct evolution algorithm.

Efficiency of x fitness cases	compared to y fitness cases	Avg. ratio	Avg. conf.
32	16	1.01	54%
64	16	1.18	80%
64	32	1.17	76%

Table 8.4: Comparison of the average efficiency ratio of even-7-parity with ADFs, averaged over the three measures and ten allocated generations. Also stated is the average confidence that the ratio is greater than one.

It is interesting to observe the range in efficiency ratios when only the raw measure (minimum computational effort, success effort, and success proportion) is modified. It is very small. Of all seven comparisons the largest range is just 6%. From this you can draw the conclusion that efficiency ratios might be similar irrespective of the underlying raw measure. However, further comparisons will be required before this conclusion will have any weight.

8.3.4 Problem Difficulty

This section considers the question "What impact does problem difficulty have on fitness-based incremental evolution's performance?".

The issue, when attempting to address this question, is *how* this analysis should be performed. One concern is that the population size varied across the problem difficulty. Another is that the number of fitness cases varied for both stage one and stage two as the problem difficulty varied. Finally, the performance varied as the number of generations allocated to stage one varied.

The issue of varying population sizes can be addressed by comparing performance to the base-line of direct evolution. In this way we will be analysing relative performance.

It is an inherent quality of the even-n-parity problem domain that the number of fitness cases in stage two varies with problem difficulty. Although this is not true of all problem domains—symbolic regression for example could sample the same number of points for both easy and hard problems—there is little that can be done to compensate for this. We will accept it as it stands.

On the other hand, the number of fitness cases in the first stage is a parameter of the technique we are studying. Two comparison techniques are apparent. The first is to keep the number constant. The second is to keep constant the proportion of fitness cases relative to the number in the second stage. Thus for even-5 with 8 cases (equivalently a proportion of $\frac{8}{32} = \frac{1}{4}$) in the first stage, we could either (a) keep the number constant and compare to even-6 with 8 fitness cases or (b) keep the proportion constant and compare to even-6 with $64 \times \frac{1}{4} = 16$ fitness cases. It is not apparent which of the two options makes for the most fair comparison. Option (a) can be viewed as letting incremental evolution do the same amount of work in the first stage irrespective of problem difficulty, while option (b) increases the absolute amount of work in stage one, but keeps the work relative to the goal stage. Option (a) has three settings: 8, 16, and 32 fitness cases. Option (b) also has three settings $\frac{1}{2}$, $\frac{1}{4}$, and $\frac{1}{8}$ of the fitness cases in the second stage.

Finally, we must consider that performance varied according to the number of generations allocated to stage one. Three comparison techniques are available here. The first is to consider if there is any way to select the optimal number of generations to allocate. Section 8.3.2 discussed this and concluded that zero generations would most likely be the best allocation (that is, the use of manual incremental evolution was a bad idea). It is unacceptable for us not to use manual incremental evolution because—despite poor performance—our objective in this section is to establish how the performance of manual incremental evolution varied with problem difficulty. The second comparison technique is to fix the number of generations allocated to the first stage. And the third is to compare the average performance across all ten different allocations.

This leaves two comparison options for the number of fitness cases in the first stage and two options for the allocated generations. The results were analysed for all four combinations using the three measures for each.

When the number of fitness cases in stage one was kept constant and the performance for all ten allocated generations was considered on one graph, all three measures for all three settings showed their efficiency ratio deteriorated as the problem difficulty increased. On the other hand, when the proportion was kept constant, the three settings had differing results: for one-eighth of the fitness cases in stage one, all three measures showed a very slight performance increase as the difficulty increased; one-half showed a mixed, but generally flat, trend; one-quarter consistently showed a decrease in performance as difficulty increased. An example of this comparison can be seen in figure 8.7.

Analysis found no interesting trends as the number of generations allocated to stage one increased.

What can we take from these results? As the problem difficulty increases it is a bad idea to keep constant the amount of work done in the first stage. This conclusion is however in line with the fact that this technique is, in general,

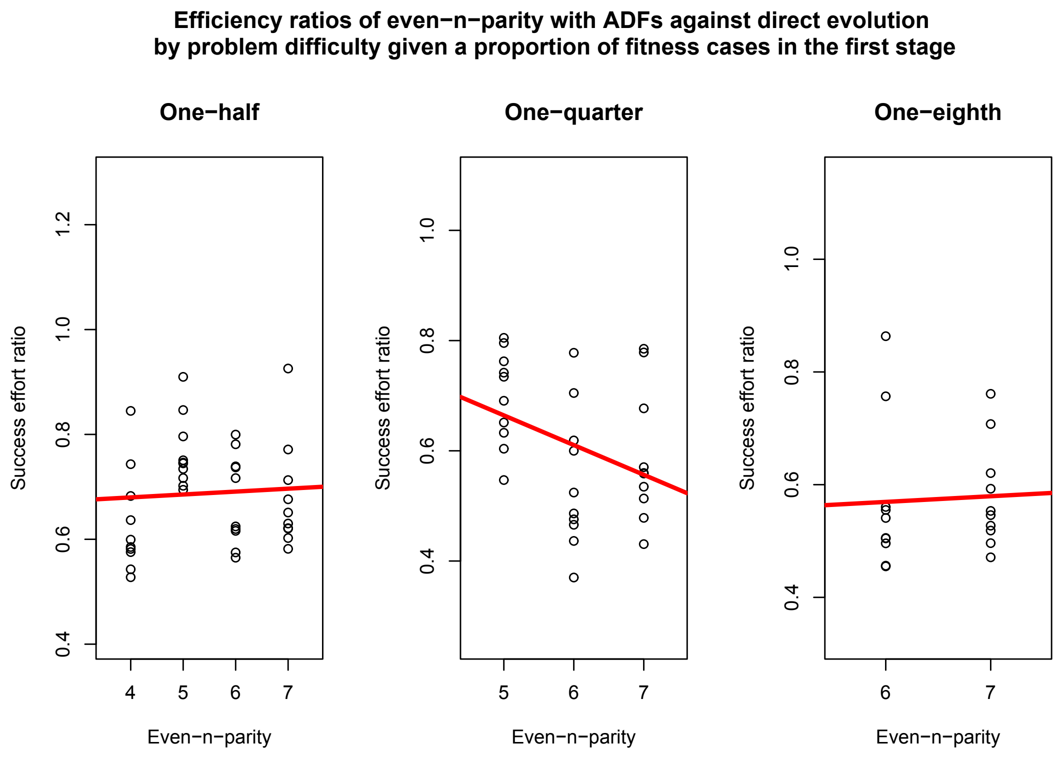


Figure 8.7: Efficiency ratios by problem difficulty.

not a good idea. It is better instead to keep the proportion constant, but these experiments do not give a clear indication as to how one might select *a priori* the optimal proportion.

8.3.5 Comparison to Direct Evolution

When manual incremental evolution with unit-cost generations were compared to direction evolution the results couldn't have been more clear: this is a bad technique.

For each of the ten experimental configurations with ADFs, the efficiency ratio of using this technique was calculated using the direct evolution results (chapter 7) as a comparison. Efficiency ratios were calculated using minimum computational effort, success effort, and final success proportion. Further, the number of generations for successful runs were plotted against the number for direct evolution.

When ADFs were used, none of the 300 ratios showed a performance improvement. And of those 300 measures less than 20% of their 95% confidence intervals even crossed the break-even line. Averaging over the ten measures in each configuration, the confidence that there was an advantage in *not* using this method, never dropped below 95%.

When ADFs were not used, and in contrast to Jackson's results, incremental evolution was outperformed by direct evolution. This occurred for both even-4 and even-5, and was often statistically significant. The difference between this result and Jackson's is not hard to explain. Firstly, only a few of his experiments beat direct evolution, and secondly none of his results were statistically significant.

As well as decreasing the final probability of finding a solution, another problem was that, especially for even-6 and -7, the technique often *increased* the number of generations before a solution was found. This flies in the face of the technique's motivation—it was supposed to make finding a solution easier.

8.3.6 Summary

What can we learn from this analysis? The most significant result was that this technique detrimentally impacted performance: you're better off without it. However, the analyses offered more than that:

- The addition of ADFs offered a performance benefit above the raw benefit offered by the addition of ADFs. In other words, fitness-based incremental evolution most likely benefited from the addition of ADF's code encapsulation mechanism.

- The best choice was to allocate as few generations as possible to the first stage. Intuition says that this gives the greatest chance of success in stage two if you fail in stage one, however the results indicated that, for some reason—not the elimination of genetic material—the route "through failure" was being eliminated by the evolution in stage one.

- The optimal number of fitness cases was "the more the merrier", but this again may have been driven by an effort to move away from incremental evolution.

- As the problem increased in difficulty, the results showed one should not keep the number of fitness cases in stage one constant. Instead, it is better to keep constant the proportion of fitness cases relative to the final stage.

8.4 Defending Adjusted Generations

Although Koza measured minimum computational effort in terms of "individuals to be processed", his expectation was made clear when he wrote "E [minimum computational effort] is the minimum number of fitness evaluations" [72, page 268]. The transition from considering fitness evaluations to considering individuals to be processed came from Koza's expectation that the number of fitness evaluations was directly proportional to the number of individuals processed. This assumption was fair given that throughout all of Koza's books the number of fitness evaluations executed per individual (and indeed per generation) remained constant.

Given that the number of fitness evaluations vary under fitness-based incremental evolution, to maintain Koza's intention one must measure computational effort based on the number of fitness evaluations. Fortunately, the number of fitness evaluations per individual remains constant across a given generation, so another option is to weight the value of each of the generations based on the number of fitness-case evaluations executed within it: a generation with 100 fitness cases should be considered twice as computationally expensive as a generation where only 50 fitness cases were evaluated.

8.4.1 Related Work

For his PhD thesis, Chris Gathercole studied standard GP with a modification he termed *dynamic subset selection* (DSS) [44, chapter 6]. For each generation, DSS uses only a subset of the total fitness cases. Selection of the subset is biased towards fitness cases that have proved difficult over the previous generations. Gathercole struggled to compare his results to standard GP and left it up to his readers to decide which assessment was appropriate: "DSS [matched] GP results using many more generations, but only 20% of the number of tree evaluations". He terminated his runs using neither a maximum number of generations nor a maximum number of evaluations (in fact, his termination criterion is unclear) and yet still compared their final results. This leaves much to be desired. Because of his very limited number of runs, he made no other analysis of his results, but had he executed a number of runs he would have benefited from the use of adjusted generations.

Users of DSS would however not generally be interested in comparing their performance to standard GP. For example, Stephenson et al. used Gathercole's technique to reduce their computational requirements [105]; they ran their test and control experiments (both using DSS) to 50 generations and then measured the speed-up of their approach—their implicit assumption was that the cost of one generation was equivalent to the next. Given DSS uses a fairly consistent number of fitness cases per generation this assumption holds true. It is only when one wishes to compare DSS against a method that used a different number of evaluations that adjusted generations would show a benefit here.

If however the DSS algorithm were modified such that the number of fitness cases was also dynamic, then just as incremental evolution benefits from adjusted generations, so too would modified-DSS. However, Gathercole himself introduces an idea that would have benefited from adjusted-generations. *Limited Error Fitness* (LEF) [44, chapter 7] terminates the evaluation of an individual if the individual fails to correctly solve a threshold of fitness cases. In this way the number of evaluations per individual is no longer constant.

In a similar vein to Gathercole's DSS, Qureshi's PhD thesis considered the comparison of standard GP to GP where a fixed number (but random selection) of fitness cases were evaluated per generation [98, section 4.7]. He too had a difficult time in that his comparison had neither the same number of generations nor the same number of evaluations. Fortunately for him, no matter which comparison he took—even if it disadvantaged his test cases—his technique outperformed his control experiment. Thus he avoided having to make a more rigorous comparison. Nonetheless an interesting point in his discussion was that some of his readers may feel that the comparison should be done on an individuals-evolved basis. Please note that adjusting the weightings of the generations does not alter the number of individuals processed.

Finally, if fitness-based rather than success-based analysis is desired, Steffen Christiansen's y-test was designed specifically for comparisons based on different evaluation counts [21, 23]. Although it appears an excellent method to distinguish between two experimental results, it is unclear how one might compare two differing y-test results. Adjusting the generations does not leave this as an issue.

8.4.2 Calculating Adjusted Generations

To calculate the adjusted generations, for each run we stored the number of fitness cases evaluated in each generation. This list of evaluations was summed, giving the total number of evaluations for the run. The total number of evaluations was then divided by the number of fitness cases in the final stage. This number was used as the cost of the run as measured in "adjusted generations".

One might claim that this measure is only a scaled version of measuring the cost of a run in terms of evaluations rather than generations. This is true. The advantage in scaling is that direct comparison with standard GP becomes possible: one can immediately compare success effort and (to a lesser extent) minimum computational effort measures.

Note that this process meant we could post-analyse a standard run and so we did not re-run the experiments for adjusted generations, but instead just re-considered their results given the new generations-to-failure or generations-to-success. Note also that this process does not change whether a run succeeded or failed; it only adjusts a run's cost.

To calculate the three measures on adjusted generations required modification only to the minimum computational effort method. The runs were binned into groups one generation wide, effectively meaning the ceiling of the adjusted-generations was used. Neither success effort nor final success proportion required modification to their methods. (Note that the use of adjusted generations has no impact at all on the final success proportion measure.)

Other than staying true to Koza's intentions, and giving a much closer approximation to the true cost of using genetic programing, this modification has little impact on the qualities of his measure. Minimum computational effort with adjusted generations still provides "a hardware-independent, software-independent, and algorithm-independent way of comparing the performance of adaptive algorithms" [72, page 293]. However, as will be discussed in the following section, when adjusted generations are used, the measure may become only an upper bound for Koza's minimum computational effort.

This modification does have an impact on the results calculated using success effort, but there is no impact on the final success proportion.

8.5 Results: Adjusted Generations

Section 8.3 discussed the results with the underlying assumption that one generation was equivalent to the next. For canonical genetic programming this is true. For incremental evolution there is good reason why it is not true. This section analyses the results given the assumption that the computational requirements for a generation are proportional to the number of fitness cases evaluated in that generation.

8.5.1 Addition of ADFs

Under adjusted generations, the addition of ADFs can again be seen to represent a benefit to incremental evolution.

As was done in section 8.3.1 the efficiency ratio of two efficiency ratios was studied and very similar results were obtained. Especially for even-5, incremental evolution benefited from the addition of ADFs above what benefits ADFs normally bring. As we discussed, this was most likely caused by the code encapsulation offered by the automatically defined functions and just as was done previously, the following analysis will focus on the results with ADFs.

8.5.2 Allocation of Generations

The objective of this section is to discover the optimal number of generations to allocate to stage one, given this new "adjusted generations" light. In comparison to the equivalent discussion using unit-cost generations (see section 8.3) the results are notably different. The most important difference is that this technique can sometimes be seen to be beneficial.

When the success effort measure was considered, performance tended to improve with the use of this technique for: even-5 with 16 fitness cases in the first

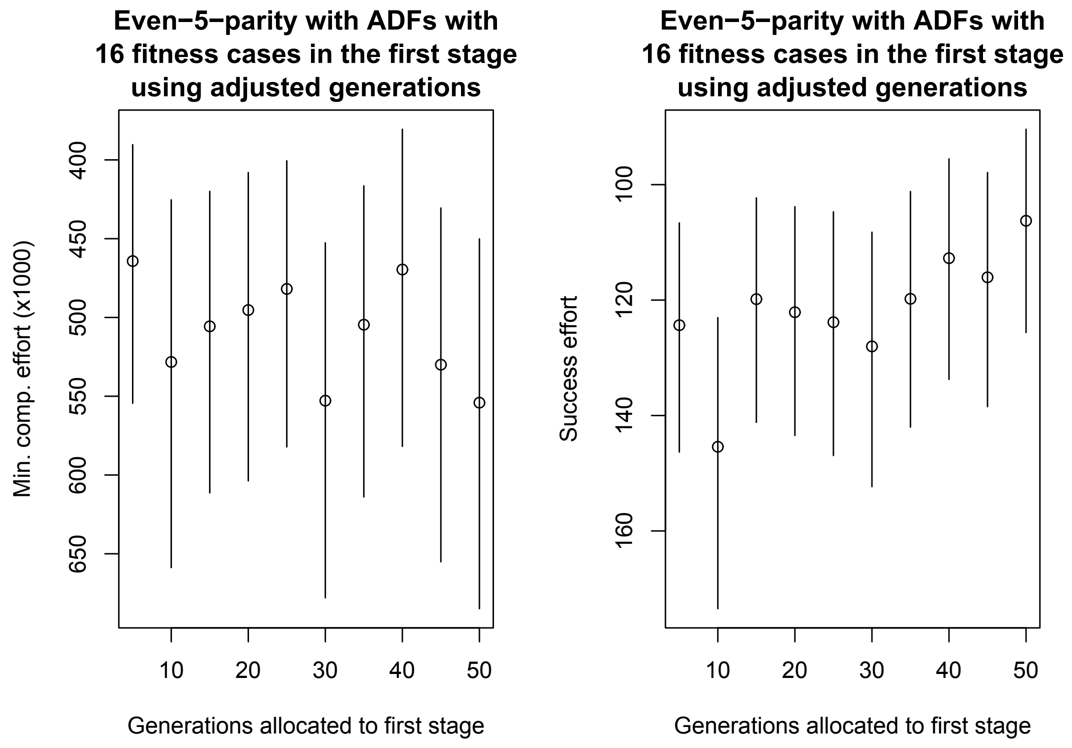


Figure 8.8: Performance as measured by minimum computational effort and success effort given adjusted generations. Note the differing trends.

stage, even-6 with 32 fitness cases, and for all three experimental configurations involving even-7. The graphs that demonstrate these trends can be found in the electronic appendix (see appendix D).

Unfortunately the same cannot be said about the results when viewed with the minimum computational effort measure. At best, one might claim that there was a slight benefit for even-5 with 8 fitness cases and even-6 with 32, but in general the results showed either a downward trend or a flat trend (which indicated incremental evolution did not have any notable impact on performance.)

This difference in opinion between success effort and minimum computational effort shows up in the average correlation coefficient between the two measures: 0.51—still a strong association, but considerably lower than section 8.3's correlation of 0.91. In fact, in one configuration a negative correlation (-0.13) was observed (see figure 8.8).

If we analyse these adjusted-generations results solely on success effort then, in terms of the optimal number of generations to allocate to stage one, we see mixed results across the different configurations. However, "mixed results" should be

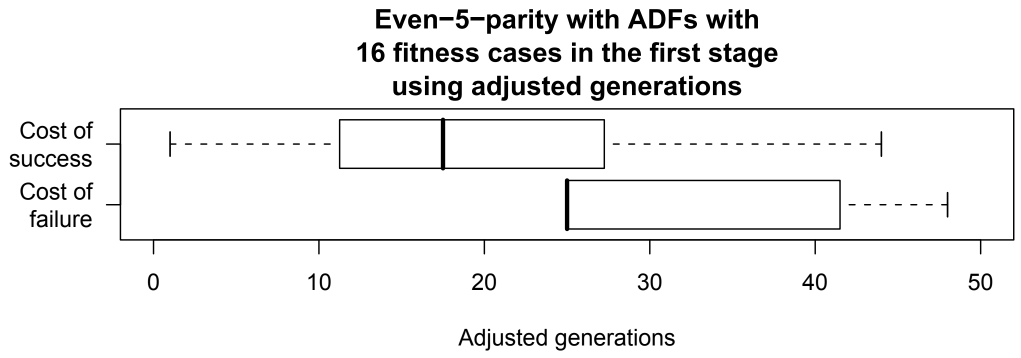


Figure 8.9: The reduced cost of failure using adjusted generations.

considered an improvement—they suggest the optimal number of generations is not clear. That result is an improvement given the obvious deterioration that previously occurred when the number of generations allocated to stage one increased (see section 8.3.2).

How is it that success effort performance can be seen to improve even as the generations allocated to stage one increase to 50 (the total number of generations allocate for a run) while at the same time minimum computational effort can show the opposite trend? This can be explained by considering the cost of failure.

If 50 generations (the maximum for a run) are allocated to stage one and a run fails to find a solution to the first stage, then it will never spend any time in stage two. If stage one considers half of the fitness cases in stage two (as is the case in figure 8.8) then, using adjusted generations, the cost of a generation in stage one is only half that of a generation in stage two. A run that fails to find a solution to stage one will therefore cost only $50 \times 0.5 = 25$ adjusted-generations to execute to termination. In contrast consider a run that finds, within say 10 generations, a solution to all of the fitness cases in stage one, and then spends 30 generations in stage two before finding a solution there. It will cost $10 \times 0.5 + 30 = 35$ adjusted-generations to find that solution. In other words it is possible that the cost of failure could be less than the cost of success. Figure 8.9 plots this effect for even-5 with 16 fitness cases in the first stage.

How is it that this effect can impact the success effort and minimum computational effort measures? Success effort directly includes the cost of failure in the numerator of its formula (see equation 4.1). As a result success effort demonstrates the performance advantage offered by adjusted generations. Minimum computational effort on the other hand does not incorporate this benefit.

Koza's measure finds the minimum adjusted-generation based on the implicit assumption that the runs would continue were they not stopped at the optimal minimum generation. Given that the use of adjusted generations upsets this assumption, what impact does this have? If the minimum cost of failure (in adjusted-generations) is above the minimum generation (in adjusted-generations) then there will be no impact on this measure. As an increase occurs in the number of failures that use fewer adjusted-generations than the minimum generation, so the observed minimum computational effort will be larger than the true value. In these situations the observed minimum computational effort should be considered as an upper bound for the true minimum computational effort. However, as we will see next, even without this modification, Koza's measure is already an upper bound.

Without modification, Koza's minimum computational effort is not capable of indicating a reduction in the effort required when the minimum cost-of-failure drops below the minimum generation. Success effort on the other hand *is* able to give this indication.

Koza's Measure is an Upper Bound

Minimum computational effort, according to Koza's first book on genetic programming, gives the number of individuals that must be processed in order to yield a solution with 99% probability by generation j [71, chapter 8], where j is what we term the minimum generation. Koza's method actually finds the upper bound to this number.

The key assumption in Koza's method is that, for a probability of success $P(i)$, one must execute i generations. This is not true. Instead, one must execute *at most* that many generations. On average, to obtain a probability of success of $P(i)$, one must execute x generations where

$$x = \frac{\int_{k=0}^{i} (Y(k) \cdot k) \cdot dk}{\int_{k=0}^{i} Y(k) \cdot dk}$$

To illustrate this visually, imagine a graph that plots any cumulative probability of success curve. Mark the probability of success at the minimum generation with an 'X'. Koza's measure will not be able to distinguish between this curve and any other that both (a) passes through 'X' and (b) is below the original curve. Yet for the original curve, there is a greater chance of finding a solution within fewer generations, and so it should naturally have a lower computational effort.

When is this issue of concern? As the required number of runs $R(P(j))$ increases, Koza's measure becomes closer and closer to the real minimum computational effort because the cost of success becomes a less and less important component of the total minimum computational effort while the cost of failure—controlled by the specification of minimum generation—becomes the more important component. However, for runs where $R(P(j)) = 1$ Koza's upper bound is furthest from the real value.

And how much of an impact does this actually have? The difference between the "real" value and the value obtained when Koza's method is followed is the difference between the mean and the maximum of the generations-to-success that are lower than the minimum generation.

Measuring what Koza Intended to Measure

It is not difficult to re-define Koza's measure so that it gives the real value, rather than an upper bound, for the expected number of individuals to be processed. The definition need only become

$$I'(i) = (X(i) + 1) \cdot M \cdot R(P(i))$$

where $X(i)$ is the mean of the generations-to-termination—with each generations-to-termination observation truncated such that the number of generations in the observation is at most i.

Making this modification would also solve the issue associated with a variable cost-of-failure given that $X(i)$ accounts for this.

Unfortunately, the impact of this modification on the coverage rates of the methods to produced confidence intervals is unknown. Further, it is inadvisable to compare minimum computational efforts produced using different methods. These two issues are sufficient disadvantage that we would hesitate to recommend the use of this modification.

Nonetheless users of Koza's measure should be aware that, on two counts, it is only an upper bound: (i) when $R(P(j))$ is low and (ii) when the cost-of-failure is below the minimum generation.

8.5.3 Fitness Cases in Stage One

The section looks at the how the number of fitness cases in the first stage impacted performance. Again, just like the unit-cost analysis of section 8.3.3, because there was no obvious manner with which to choose the “best” number of generations to allocate to stage one, we considered the average across each of the ten experiments executed for each configuration. But again because of the difference in configuration between the two experiments with even-4, we limited the comparison to even-5, -6, and -7.

The ordering is similar to that seen in section 8.3.3. It is better to use more fitness cases in the first stage: for even-5, 16 fitness cases in the first stage outperforms 8; for even-6, 32 fitness cases outperforms 16, which does better than 8; and for even-7, the best choice was 64, followed by 32, and then 16. This trend is still fairly clear with an average confidence that a difference truly exists of some 66% (averaged over each of the seven comparisons)

What is not clear is why such a trend exists. Given that there is an advantage to be had in terms of cost-of-failure, one might assume that this trend would peak at some maximum proportion of fitness cases (although the proportion would most likely be problem-dependent). Further experiments (beyond the maximum proportion of 0.5 used in this work) would be required to assess if this hypothesis is true.

On the other hand, it is possible that this trend occurred as an indication that, despite the adjusted-generations light, performance is still generally inferior to direct evolution. A comparison of performance can be found in section 8.5.5.

8.5.4 Problem Difficulty

We now attempt to discover how this technique, under adjusted generations, performed as the problems became more difficult. Analysis similar to that of section 8.3.4 was performed, but this time the results are encouraging.

The most interesting result can be seen in figure 8.10—incremental evolution was shown to offer increased benefits as the problem became more difficult. When considered by the number of allocated generations, 25 of the 30 graphs showed an increase or flat trend as the difficulty increased. These are very interesting results; they indicate that, under the adjusted generations light, incremental evolution may become more beneficial as the problem domains become harder.

This result is only true if success effort was considered. We have seen that final success proportion decreases with the use of this technique and that neither success proportion nor minimum computational effort appreciate the reduced cost of runs that terminate before the minimum generation, so limiting the analysis to success effort is quite reasonable.

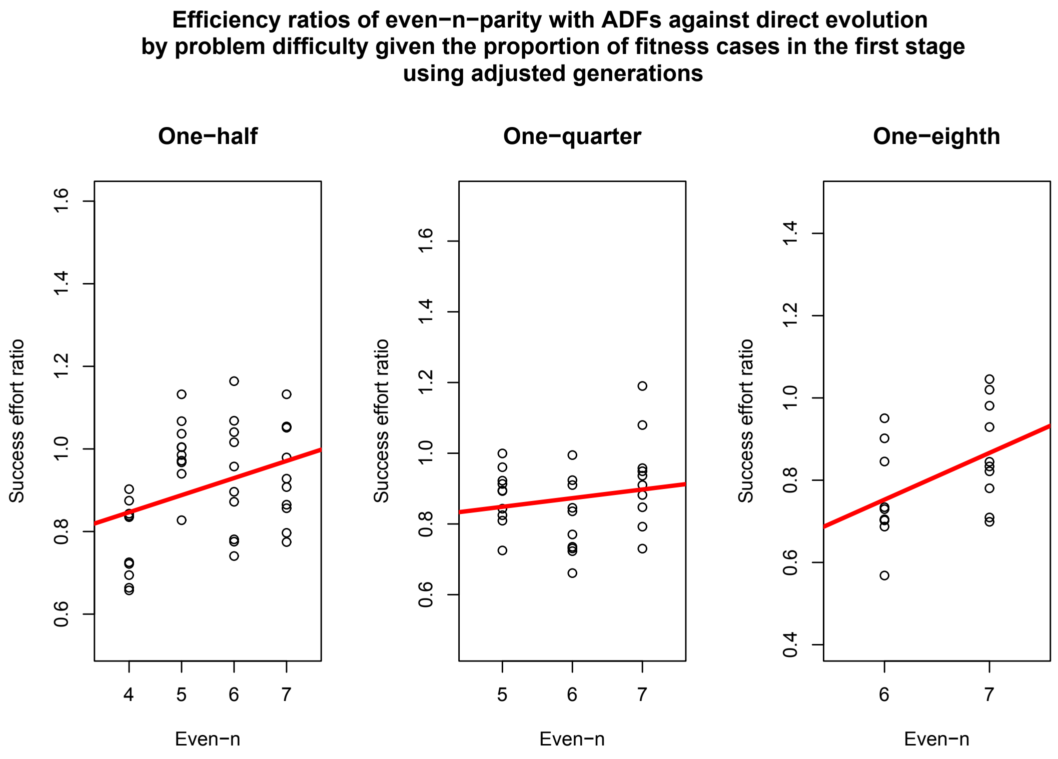


Figure 8.10: Efficiency ratios by problem difficulty.

In line with the analysis in section 8.3.4, if the number of fitness cases in the first stage is kept constant then incremental evolution decreases in performance as the problem increases in difficulty.

8.5.5 Comparison to Direct Evolution

When a comparison is made to direct evolution two things become clear. The first is that we are reminded that the probability of finding a solution consistently decreased as the number of generations allocated to stage one increased (remember that adjusted generations have no impact on final success proportion). But despite this raw reduction in performance, in five of the ten configurations with ADFs, there were occurrences where the required success effort was reduced. So incremental evolution with adjusted generations can offer a benefit. This is an interesting result.

How does this happen? The reason is that both the cost-of-failure and the cost-of-success decrease with this method. The decreases are sufficient to overcome the reduced success proportion.

However, it is not true that this technique is generally beneficial. None of the ten configurations had (over the ten settings for allocated generations) an average success effort ratio that was above one. Given that there was no obvious way to select the optimal (or even near-optimal) number of generations to allocate to stage one, we thus have no rule-of-thumb to predict whether use of this technique would be beneficial.

8.5.6 Summary

What can we take from the result under adjusted generations? The most interesting result was that fitness-based incremental evolution was sometimes beneficial. The other points were:

- The benefits of adding ADFs were again beyond what is normally seen.
- If the technique was beneficial then it was best to allocate as many generations as possible to the first stage. This was shown to decrease the cost of failure, which we then showed that minimum computational effort may not be able to measure. Koza's statistic was also demonstrated to be an upper bound.

- It was shown that performance improved as the number of fitness cases increased. This occurred despite the benefits of reduced cost-of-failure associated with fewer fitness cases in the first stage.

- Incremental evolution was shown to improve in performance as the problem domain became more difficult.

8.6 Future Research

Although this study of fitness-based incremental evolution on even-n-parity was fairly thorough, there are still plenty of unanswered questions.

The greatest limitation of this work is that only one problem domain, even-n-parity, was studied. It is highly likely that the results will vary by problem domain, so it would be interesting to apply this technique to different areas.

It would be beneficial to study how the selection of fitness cases impacts the result. In this work we looked only at selecting the first x fitness cases, but there is the possibility of choosing randomly or possibly pre-ordering the fitness cases with the idea of "coaching" the system. This is very similar to DSS so there would be benefit in a comparative study of incremental evolution and Gathercole's DSS.

When adjusted generations were used it was hypothesised that there was a peak in terms of the optimal number of fitness cases to choose for the first stage (see section 8.5.3). Especially given the trend towards a greater benefit for harder problems, it would be interesting to study this hypothesis under problems even more difficult than even-7-parity.

Qureshi showed that considering a small number of randomly-selected fitness cases per generation (rather than the full set) produced more general solutions for his pursuit game [98, section 4.7]. Our research did not consider this potential advantage and so there is scope for future work to study if fitness-based incremental evolution produces more general solutions.

The addition of ADFs was not able to be well studied as the performance at even-6 and -7 was so low for incremental evolution without ADFs. Experiments with larger population sizes would be required. First it should be confirmed that ADFs do indeed offer an improvement above what can be normally expected. If that is confirmed it would also be interesting to analyse *how* ADFs facilitate the evolutionary process: is the encapsulation hypothesis correct. From other experiments we ran where incremental evolution was not beneficial [112], encapsulation was one of the suggested potential "solutions". It is possible that ADFs were more useful for this problem domain than they would be in general. Also, we have not considered the fact that executing individuals with ADFs typically takes much longer (from a "CPU time" perspective) than execution without ADFs.

Jackson has offered another form of fitness-based incremental evolution: parameterless functions where the main function is also evolved. Although his solution to its problem (of an increasingly-difficult-to-evolve main function) was dismissed, his solution is not the only option. Study of the balance between a decreased difficulty of the parameterless functions versus an increased difficulty of the main function deserves attention. We, however, are going to continue further down our current road.

Finally, the question that goes begging in this chapter is "how does performance change with more than two stages?" Others have studied such systems [12, 14, 16, 17, 34, 42, 50–53, 58, 81, 89, 112] but thorough analysis has not been done. It is likely that the sheer number of permutations limits the feasibility of this, but if it were to be done, others' results would indicate that longer run lengths may be beneficial [44, 83]. In the next chapter we will consider an automated process that produces a dynamic number of stages; we also consider longer run lengths.

Chapter 9

Automatic Fitness-Based Incremental Evolution

In the previous chapter we learnt that manual incremental evolution could sometimes be beneficial on the even-n-parity problem so long as the results were considered in terms of adjusted generations. The primary reason was that, under adjusted generations, the cost of failure could be significantly reduced.

In this chapter we offer two novel methods for automating fitness-based incremental evolution. These methods outperform the results from the last chapter and, under adjusted generations, even regularly outperform direct evolution.

9.1 Motivation

In the study of manual incremental evolution we showed it was possible to outperform direct evolution. However, it was difficult to see how this level of performance might be predicted. We learnt:

- ADFs were beneficial to the incremental evolution process. They offered benefits superior to the benefits they offered to direct evolution.

- To see any advantage, it was clear that one had to accept performance measured in terms of adjusted generations. The use of manual incremental evolution decreased the probability of finding a solution but it also decreased both the cost of failure and the cost of success. If these latter advantages are not to be considered, then manual incremental evolution should be discarded.

- The primary beneficial effect was the decreased cost of failure (a benefit that success effort excels at measuring). This was achieved when the first stage used up the allocated generations with relatively little computational effort. However, the results were confusing as they indicated that increasing the number of fitness cases in the first stage improved performance.

- Finally, performance increased as the problem difficulty increased.

Although those results gave some amount of hope for fitness-based incremental evolution, it was still unclear how one might configure the stages. The primary unanswered question was how three (or more) stages would impact performance. Secondary to that was how many fitness cases each stage should be allocated. Finally, it was unclear whether the use of incremental evolution would be beneficial.

The ideas in this chapter were motivated by the questions of how many stages one should use and how many fitness cases should be specified. What if the number of stages and number of cases were automatically specified depending on a run's performance? We could set GP an initial number of fitness cases to solve. If it succeeded then we could automatically move on to something more difficult. If it failed then we could reduce the number of fitness cases and try again.

Two methods spawned from this idea and are introduced in section 9.4. Conceptually, they differ only in terms of the strategy on success. If GP solves the initial number of fitness cases then one could either aggressively try to solve the goal (or complete set of fitness cases), or one might try a less-aggressive step somewhere between the last success and the goal. However, it's important to remember that generations must be spent at the goal stage; it is not very useful to perpetually aim for half the distance between where you are and the goal—such a technique would guarantee failure.

Finally, we are very pleased to be able to say that, unlike the last chapter, you will see that these two methods produced a number of positive results.

9.2 Related Work

As introduced in the literature review (see section 6.2.7), the concept of "automatic incremental evolution" is not new. We are aware of two problem-specific forms that have been offered within genetic programming, both of which manipulated the environment (as opposed to the fitness function) to make it progressively more difficult.

The first use was originally published by Faustino Gomez in 1999 [52] and then later as part of his PhD thesis [53]. It was a technique to evolve solutions to the two-pole-balancing problem. There were two poles attached to one cart on rails and the controller's job was to apply a force at each time-step such that it balanced the poles for 100,000 time-steps (30 simulated minutes). The problem is known to become more difficult as the difference between the two pole lengths decreases.

The system started with the shorter pole just 10% the length of the longer pole. If the task was solved, the shorter pole was automatically lengthened by a specified length. If evolution was unsuccessful then the length of the shorter pole was reduced to a length half-way between the current length and the last successful length. This process was repeated, producing on average 30 stages to reach the goal—where the shorter pole was 80% the length of the longer pole. Direct evolution, in contrast, failed to find a solution even where the shorter pole had a 50% length.

Although Gomez's implementation of this automatic approach was very successful, he pointed out it was not entirely novel. Others had used a similar technique, but rather than up to 100% increases in the length of the shorter pole, they used only 1% increases [100, 122] and used up to 220 stages [122].

The second use of automatic incremental evolution in GP was by Whiteson et al. [119]. They studied the keep-away soccer domain by initially fixing the speed of the opponent to just 10% of the controlled players' speed. When the average player achieved a specified performance level, then the opponent's speed was automatically incremented five percentage points and evolution continued. However, Whiteson et al. did not study the impact of their approach.

Mouret et al. used incremental evolution to evolve a wing-flapping robot controller [88]. They evolved two wing-beat controllers and then evolved a tail controller. They were surprised by the unintuitive combined performance and concluded that "to raise [the] chances of success, the recourse to some sort of automatic incremental methodology seems mandatory". They felt their experience showed them "one cannot rely on fundamental principles or empirical knowledge" to break up the goal problem into easier sub-problems.

In contrast to the automatic incremental evolution offered by Gomez and Whiteson et al., the form of incremental evolution used in this chapter is one that is potentially generally applicable. Rather than modifying the environment directly, it modifies the way fitness is calculated and thus focuses evolution on a specific area of the problem, automatically gradually enlarging the specific area.

The ideas in this chapter are somewhat similar to the subset-selection schemes[1] Chris Gathercole offered in his PhD thesis [44, chapter 6]. Although not a form of incremental evolution, Gathercole's schemes involved the selection of only a portion of the potential fitness cases used for testing. He found that the techniques produced "results as good as those of standard GP and in much shorter time" (albeit not consistently).

Finally, the use of mutation in this work is similar in motivation to the "burst mutation" scheme suggested by Gomez [53] and the delta-coding strategy by Whitley et al. [121]. The similarities are that mutation was used infrequently—its use triggered only when convergence may have stagnated. Both Gomez and Whitley et al. mutated only the population's best individual, while we have previously discussed benefits of mutating the entire population [113]. In this chapter we took a similar approach in that any individual in the population was a potential mutation candidate.

9.3 The Algorithms

Four different approaches were considered for this work: an aggressive strategy with and without the use of mutation, and a less aggressive strategy, again, with and without mutation. The aggressive strategy is described in table 9.1 and the less-aggressive in table 9.2. The two aggression options only change the fitness-evaluation function—the genetic programming algorithm is otherwise unchanged.

A graphical representation of the two aggression options can be seen in figures 9.1 and 9.2. Both examples attempted to solve the even-7-parity problem (which has 128 fitness cases) using a maximum of 5 generations before an automatic step; both examples failed to find a solution.

The aggressive option (figure 9.1) started by attempting to evolve a solution using the full 128 fitness cases. After five generations a solution had not been found so the number of fitness cases was reduced to half-way between the current (128 cases) and the value for f_{lower} (1): $\frac{128+1}{2} = 64.5$. This was converted to an integer, meaning 64 fitness cases were used in the second stage. After a further five generations a solution had not been found so the number of fitness cases was halved again to $\frac{64+1}{2} \rightarrow 32$. Five further generations were also unsuccessful, so the number of fitness cases was reduced to 16, and then again to 8. After five generations with 8 fitness cases a solution was found (meaning f_{lower} was set to 8)

[1] Dynamic subset selection (DSS), historical subset selection (HSS) and random subset selection (RSS).

1. This method requires:
   - a lower bound for the number of fitness cases to use during evaluation of an individual (f_{lower}),
   - a number of fitness cases for the goal stage (f_{goal}),
   - the maximum number of generations before a step will automatically be taken (g_{max}), and
   - the fitness cases to be used for evaluation.
2. To start, initialise the population, set the number of generations-so-far for this step to zero ($g \longleftarrow 0$), and evaluate every individual using all $f \longleftarrow f_{\text{goal}}$ fitness cases.
3. Loop through the following instructions until, either an individual succeeded in all tested cases and $f = f_{\text{goal}}$, or the number of generations exceeds the specified maximum.
   (a) Increment the number of generations for this step ($g \longleftarrow g + 1$).
   (b) If $g > g_{\text{max}}$ then this step has failed to find a solution within the specified number of generations. Set the number of fitness cases to use for evaluation to $f \longleftarrow \frac{f+f_{\text{lower}}}{2}$ and reset the number of generations for this (next) step ($g \longleftarrow 0$). If a mutation operation is to be used, apply it to the population.
   (c) Evolve the population as usual and evaluate it based on the first f fitness cases.
   (d) If a solution has been found and $f \neq f_{\text{goal}}$ then increase f's lower bound ($f_{\text{lower}} \longleftarrow f$), and reset the number of fitness cases to $f \longleftarrow f_{\text{goal}}$. Also reset the number of generations for this next step ($g \longleftarrow 0$). If mutation is to be used, mutate the population.

Table 9.1: The aggressive strategy for automatic incremental evolution.

1. This method requires:
   - a lower bound for the number of fitness cases to use during evaluation of an individual (f_{lower}),
   - a number of fitness cases for the goal stage (f_{goal}),
   - a stack of values that will contain the values of f where failure occurred,
   - the maximum number of generations before a step will automatically be taken ($g_{\max}$), and
   - the fitness cases to be used for evaluation.
2. To start, initialise the population, clear the stack to empty, set the number of generations-so-far for this step to zero ($g \longleftarrow 0$), and evaluate every individual using $f \longleftarrow \frac{f_{\text{goal}}+f_{\text{lower}}}{2}$ fitness cases.
3. Loop through the following instructions until, either an individual succeeded in all tested cases and $f = f_{\text{goal}}$, or the number of generations exceeds the specified maximum.
   (a) Increment the number of generations for this step ($g \longleftarrow g + 1$).
   (b) If $g > g_{\max}$ then this step has failed to find a solution within the specified number of generations. If $f \neq$ *top of stack* then push f on to the stack. Set the number of fitness cases to use for evaluation to $f \longleftarrow \frac{f+f_{\text{lower}}}{2}$ and reset the number of generations for this (next) step ($g \longleftarrow 0$). If a mutation operation is to be used, apply it to the population.
   (c) Evolve the population as usual and evaluate it based on the first f fitness cases.
   (d) If we have found a solution and $f \neq f_{\text{goal}}$ then: if the stack isn't empty set the number of fitness cases to $f \longleftarrow$ *top of stack* and pop the top off the stack; if the stack is empty then $f \longleftarrow f_{\text{goal}}$. Also reset the number of generations for this next step ($g \longleftarrow 0$). If mutation is to be used, mutate the population.

Table 9.2: The less-aggressive strategy for automatic incremental evolution.

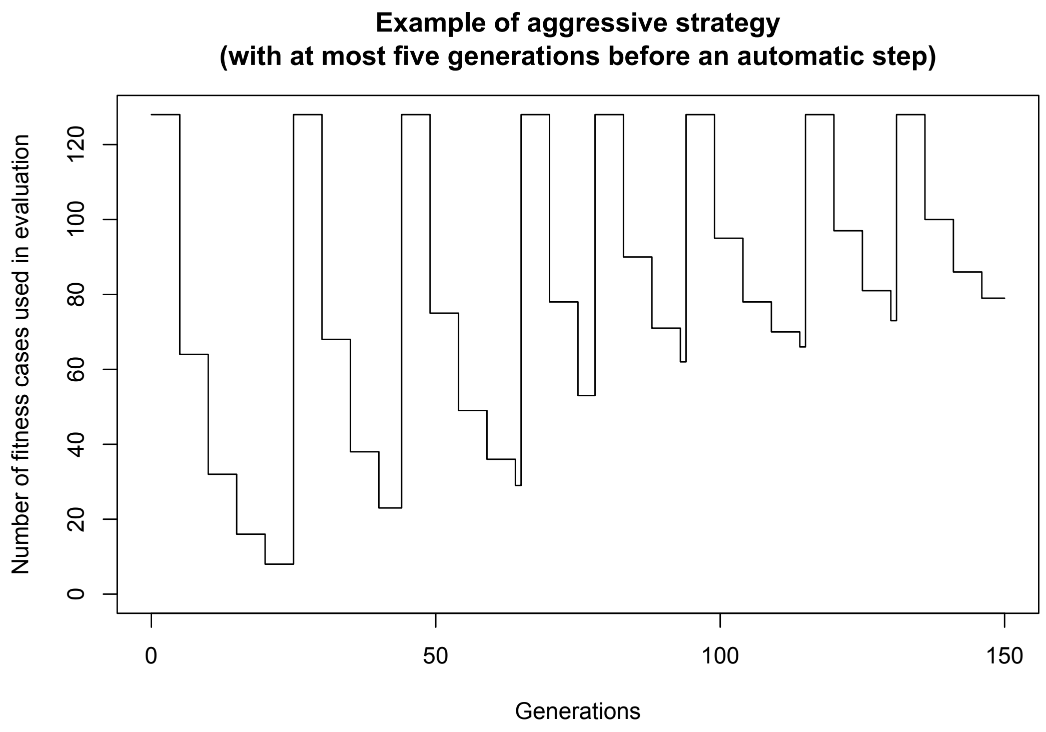


Figure 9.1: Example of an (unsuccessful) run of the aggressive strategy on the even-7-parity problem (which has 128 fitness cases).

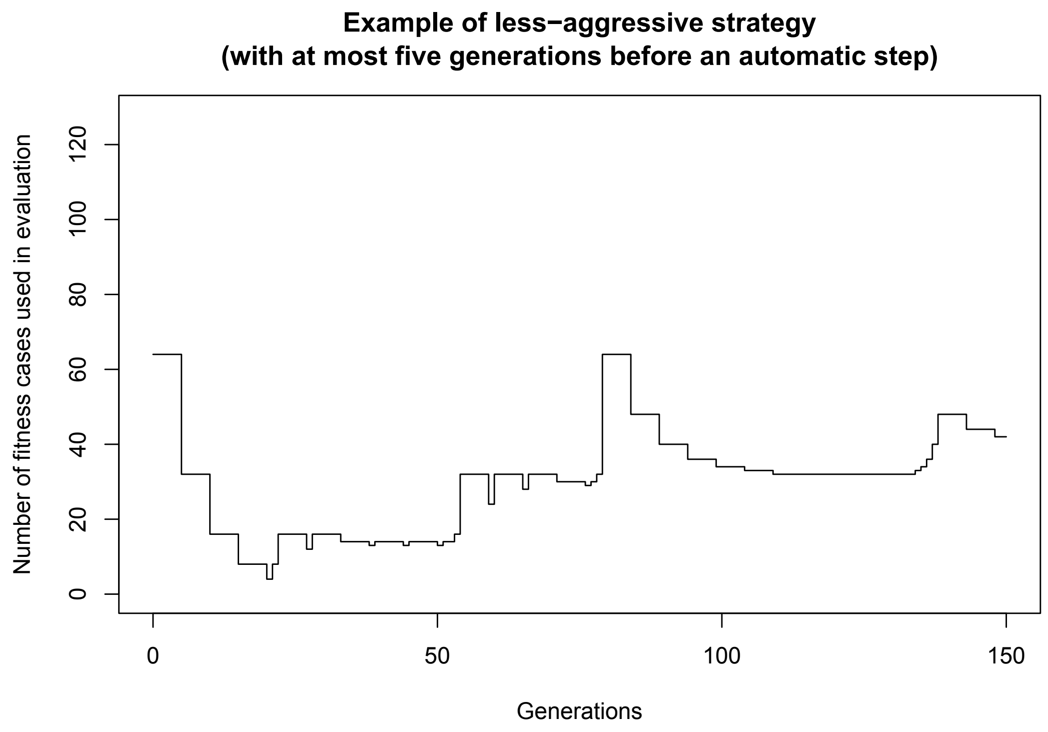


Figure 9.2: Example of an (unsuccessful) run of the less-aggressive strategy on the even-7-parity problem (which has 128 fitness cases).

so the next stage attempted all 128 fitness cases. This failed to find a solution so the number of fitness cases was again reduced, to $\frac{128+8}{2} \rightarrow 68$ cases. This process continued but did not find a solution to all 128 fitness cases within 150 generations.

The less-aggressive option (figure 9.2) started by evolving with half the number of fitness cases between f_{goal} and f_{lower}: 64 fitness cases. After five generations a solution had not been found so the number 64 was pushed onto the stack and $\frac{64+1}{2} \rightarrow 32$ fitness cases were used in the second stage. A solution was not found, so 32 was pushed onto the stack and 16 fitness cases were used. Five generations again elapsed, so 16 was pushed onto the stack and 8 cases were used. 8 cases also proved too difficult, so 8 was pushed onto the stack and the first 4 fitness cases were used. A solution to 4 cases was found within one generation, so the top of the stack was taken (8) and used in the next stage. That was also solved within one generation, so again the top of the stack (16) was used. A solution to 16 fitness cases was not found within five generations so 16 was again pushed onto the stack and the next stage used half the current number (16) and the last number of successful fitness cases (8): $\frac{16+8}{2} \rightarrow 12$. This process continued to 150 generations but failed to find a solution to the full 128 cases.

You may have noticed that it is possible that the final generation (or indeed, stage) may not evaluate the population on the full set of fitness cases, thus making success an impossibility. This topic is discussed further in section 9.9 and is the topic of the next chapter.

Finally, one small note that might save an implementor's time: it is important *not* to mutate after a success at the goal state. Mutating at that point may well mean you destroy the individual that found a solution.

9.4 Method

For each of the four approaches (two aggression options and two mutation options) we experimented with the even-4, -5, -6, and -7 problems up to 150 generations. For each of the four approaches eight different values—5, 10, 15, 20, 25, 30, 40 and 50—were used for the number of generations before an automatic step (g_{max}). Thus a total of 128 experiments were executed, each for 500 runs.

If mutation was required then 50% of the population was randomly chosen to undergo subtree mutation (as originally described by Koza [71, section 6.5.1] and implemented in Beagle by the class GP::MutationStandardOp). A maximum depth of five was used for the creation of mutant subtrees.

Given that ADFs were shown to be of benefit under manual incremental evolution (chapter 8) we elected to use them exclusively. We did not study the impact of removing the use of ADFs from the algorithm, but it is highly likely that only a reduction in performance would have been seen. ADFs were configured in the same way as the direct evolution runs (see table 7.1 on page 116).

The minor parameters were specified in the same way as was done with direct evolution (see section 7.2). Runs were analysed to 150 generations unless otherwise specified.

9.5 Results: Unit-Cost Generations

As was done with manual incremental evolution, we will first analyse the results based on the assumption that every generation has equivalent cost. Section 9.6 considers the results using adjusted generations.

When averaged across all four problem difficulty levels, all eight $g_{\max}$ values, and all four combinations of aggression and mutation, automatic incremental evolution had a success effort efficiency ratio of 0.86 compared to direct evolution. In other words, on average, direct evolution was the better choice—with an average confidence of 70%. Of the 128 experiments, 95 (74%) had a ratio pointing to direct evolution.

The use of minimum computational effort produced very similar results: an average efficiency ratio of 0.83 and a confidence, that direct evolution was the better choice, of 69%. 94 experiments had an efficiency ratio less than one.

Success proportion also produced very similar results with an average efficiency ratio of 0.90 and a confidence—in direct evolution—of 69%. 95 of the experiments pointed to direct evolution.

However not all configurations of automatic incremental evolution fared as poorly as the average. The following sections detail the influence that aggression, mutation, $g_{\max}$, problem difficulty, and run length had on performance.

9.5.1 Aggression

Which of the two strategies was better, the aggressive one or the less-aggressive one? Irrespective of the measure used, there was little doubt. The aggressive strategy was the higher performer.

Under success effort, when compared to direct evolution, automatic incremental evolution with aggression scored an average efficiency ratio of 1.004—very

		Success effort	Min. comp. effort	Success Proportion
Aggressive	ratio	1.004	0.97	1.006
	conf.	46%	45%	47%
Less-aggressive	ratio	0.71	0.70	0.79
	conf.	15%	18%	15%

Table 9.3: Average efficiency ratios and confidence levels for the aggressive and less-aggressive strategies when compared to direct evolution. Ratios greater than one mean that, on average, direct evolution was inferior. Confidence levels all indicate direct evolution was superior—if only marginally.

	Success effort	Min. comp. effort	Success Proportion
ratio	1.48	1.50	1.32
conf.	83%	80%	83%

Table 9.4: Average efficiency ratios and confidence levels comparing the aggressive and less-aggressive strategies. Ratios greater than one mean that, on average, aggression was the higher performer. Also shown is the confidence one should have in aggression being a better choice than the less-aggressive strategy.

slightly better than direct evolution. In contrast, the use of the less-aggressive strategy gave an efficiency ratio of 0.71. Their average confidence levels provide an interestingly, subtly-different picture. Even though the aggressive option had an average ratio above 1.0, the average confidence pointed in the other direction: with just 46% confidence in the aggressive option being better than direct evolution. The less-aggressive option scored a comparable average confidence of 15% (or 85% confidence in direct evolution being the better choice).

Table 9.3 gives a summary of the efficiency ratios—where automatic incremental evolution was compared to direct evolution—for success effort, minimum computational effort, and final success proportion.

This evidence leaves little doubt that the aggressive strategy was the more efficient choice, but a better approach would be to compare the two options directly. By eliminating the interference of the direct evolution results we can more clearly see the expected benefit and our confidence in the results. Table 9.4 gives such a comparison and shows the aggressive strategy is better than the non-aggressive version.

9.5.2 Mutation

The use of mutation was most likely a good thing. Experiments with mutation had an average success effort efficiency ratio of 1.18 when compared to experiments without mutation—with an average confidence of 62% that mutation improved performance.

Minimum computational effort produced a similar result (1.22) with similar confidence levels (58%). Success proportion at the final generation also produced similar results (1.12 and 62% respectively).

The results were consistent with and without aggression, thus the highest performing approach was the aggressive strategy with mutation. In fact, that approach had an average success effort efficiency ratio of 1.03—although the confidence level showed that it was no better than 50:50. This result indicates that fitness-based automatic increment evolution can equal direct evolution.

9.5.3 Generations Before Step

Averaged over all aggression, mutation, and problem difficulty settings, there was little evidence for a general trend. When we considered only the results where the aggressive strategy was used, it appeared that performance decreased as $g_{\max}$ increased—but the variability in the results means that this should be regarded as insignificant. The trend appeared to strengthen when both mutation and aggression were used, but again this should be treated as insignificant.

The number of stages[2] used during a run depended on the number of generations before an automatic step ($gmax$). It was highest for runs with a $g_{\max}$ of 5, where an average of 25 stages were used over the 150 generations. For runs with a $g_{\max}$ value of 50, an average of only 3.4 stages were used.

9.5.4 Problem Difficulty

Unlike the previous chapter, this comparison was fairly straight-forward; one must account only for the changes in population size. A comparison with direct evolution does this, leaving us to compare efficiency ratios.

Averaged over all other settings, a slight trend can be seen. As the problem becomes more challenging, automatic incremental evolution decreases performance less. At even-4 the success effort efficiency ratio was 0.73 and by even-7

[2] A stage is a group of generations where the number of fitness cases remains constant.

the average ratio increased to 0.92. Minimum computational effort and success proportion ratios produced very similar results.

9.5.5 Run Length

These experiments were all executed to 150 generations, or three-times the traditional run length. This additional length favoured incremental evolution, as direct evolution performed even more strongly when the run lengths were artificially shortened (by labelling as failures all runs that executed more than 50 generations, and by clamping the number of generations to 50).

Reducing the number of generations did not change automatic incremental evolution's relative performance—the aggressive strategy with mutation was still the top choice of the four approaches.

9.5.6 Summary

From this analysis, the better choice was to use direct evolution. Although incremental evolution using the aggressive strategy with mutation may have offered a slight performance benefit, the advantage was certainly not sufficient to warrant the additional complexity. However as we will now see, the use of adjusted generations will change that conclusion.

But before we continue, one point that is worth noting is that success effort and minimum computational effort consistently provided very similar conclusions.

9.6 Results: Adjusted Generations

This section re-analyses the results with a modified assumption about the cost of a generation. In this section we assume that the cost of a generation is proportional to the number of individual-evaluations executed. This idea was discussed in detail in the previous chapter (see section 8.5).

As we discussed previously (section 8.5.2), minimum computational effort becomes an upper bound when the cost of failure is below the minimum generation. That makes it an inappropriate measure to rely on for this section as one impact of the use of adjusted generations and fitness-based incremental evolution is to reduce the cost of failure. In fact—and we will discuss this next—automatic incremental evolution does a very good job of reducing that cost.

When averaged over all 128 experiments, automatic incremental evolution with adjusted generations had a success effort efficiency ratio of 1.53 when compared to direct evolution. We found an average confidence of 81% that automatic incremental evolution was a better choice than direct evolution. 108 (84%) of the 128 experiments had an efficiency ratio greater than one.

The use of minimum computational effort on the other hand produced an efficiency ratio of 1.07—but with only a 51% confidence that incremental evolution was the better when compared to direct evolution. We will not discuss minimum computational effort measures any further in this section.

As expected, success proportion produced identical results with and without adjusted generations—it is, after all, only measuring the probability of success and does not consider the cost to obtain those successes. Consequently, success proportion will not be considered any further in this section.

9.6.1 Aggression

The use of the aggressive strategy produced a success effort efficiency ratio of 1.48 when compared to direct evolution. Associated to that was a confidence of 72% that the direct evolution was the inferior performer. The use of the less-aggressive strategy gave a comparable ratio of 1.58 and confidence of 89%. So, in contrast to the results with unit-cost generations, the less aggressive strategy won out; but both beat direct evolution.

A comparison directly comparing the two approaches gave a success effort efficiency ratio of 1.05 and a confidence of 58% that the less-aggressive strategy was better than the aggressive one–certainly not a significant difference between them.

There exists an interesting balance between the aggressive and less-aggressive strategies. The aggressive strategy was shown to improve the likelihood of finding a solution (see section 9.5.1), but the less-aggressive strategy will "hold down" a poor performer more than the aggressive strategy and thus reduce the cost of failure.

9.6.2 Mutation

The results with mutation were consistent with the results under the unit-cost assumption: the use of mutation was most likely a good idea. Using mutation gave a success effort efficiency ratio of 1.16 when compared to not using it. The confidence of 62% however was not particularly high.

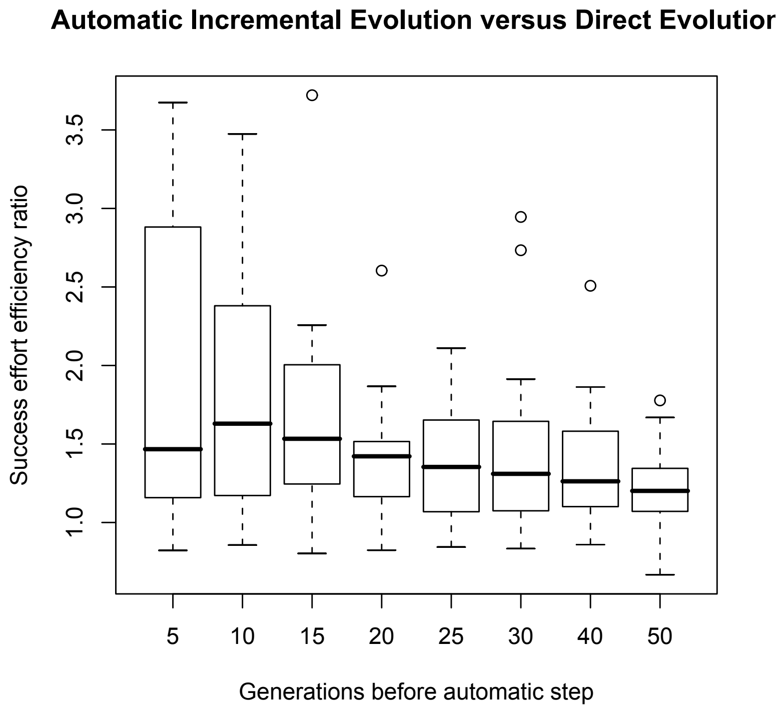


Figure 9.3: The advantage of automatic incremental evolution when adjusted generations were used, by values of g_{max}. The results include all values of aggression, mutation, and problem difficulty.

The results were consistent with and without aggression, thus making the highest performer of the four approaches the less-aggressive strategy with mutation. Compared to direct evolution it had a ratio of 1.70 and a confidence, that automatic incremental evolution was the better choice, of 79%.

9.6.3 Generations Before Step

Performance decreased, in general, as the number of generations before an automatic step (g_{max}) increased. Figure 9.3 plots this effect.

The optimum appeared not to be 5 generations, but 10. This is another balance between increasing the probability of success and decreasing the cost of failure. A higher value of g_{max} improved the probability of success but a lower value allowed the algorithm to more quickly "hold down" poor performers and thus decrease the cost of failure.

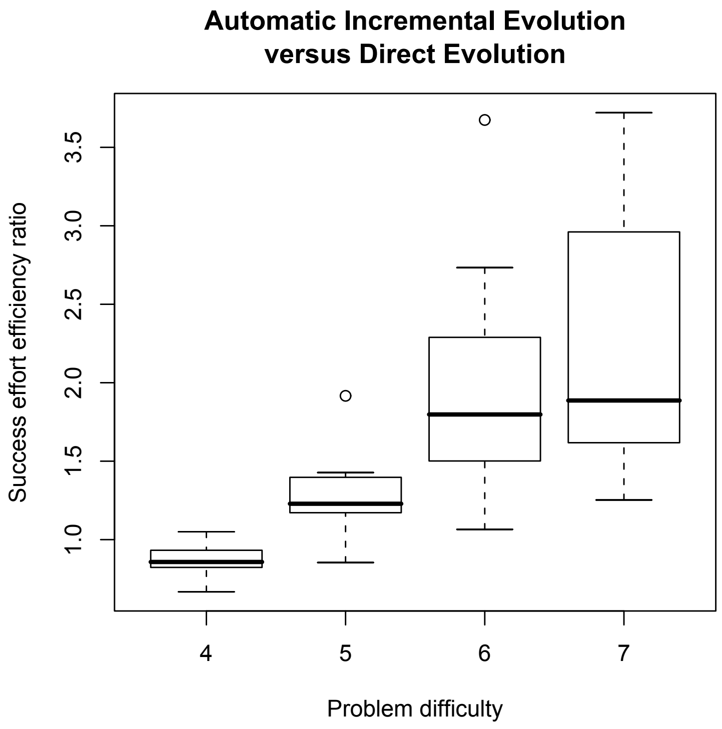


Figure 9.4: The advantage of automatic incremental evolution when adjusted generations were used, by problem difficulty. The results are for the less-aggressive strategy.

9.6.4 Problem Difficulty

Unlike the results when unit-cost generations are assumed, in this case there was an obvious trend as the problem difficulty increases: the improvement over direct evolution increased. Figure 9.4 demonstrates this for the less-aggressive strategy, but the results were similar irrespective of aggression and mutation.

9.6.5 Run Length

Under the unit-cost assumption the longer run lengths were beneficial to incremental evolution. Under adjusted generations, the difference was almost crucial. With 150 generations, automatic incremental evolution had an efficiency ratio of 1.53 over direct evolution (averaging over all settings of aggression, mutation, $g_{\max}$, and problem difficulty). If only 50 generations were used the ratio dropped to 1.03. Further, the advantage that the less-aggressive strategy had with 150 generations completely evaporated when limited to 50 generations.

9.6.6 Summary

When compared to automatic incremental evolution with unit-cost generations, the use of adjusted generations produced an average success effort efficiency ratio of 1.81. This is a significant difference (with a confidence level of 96%), based only on whether generations or evaluations were counted.

The best of the configurations with adjusted generations was the less-aggressive strategy with mutation and about 10 generations before an automatic step. It offered a success effort efficiency ratio of between 1.10 (for even-4) and 2.17 (for even-7). We can be 95% confident that, on average, that specification of automatic incremental evolution was better than direct evolution.

9.7 Analysis of the Three Measures

This study has analysed the use of automatic incremental evolution, but in this section we will shift the focus to our three measures: minimum computational effort, success effort, and final success proportion.

The most apparent point is that mentioned in section 9.6: success proportion cannot measure changes in the cost to execute runs. Given that automatic incremental evolution's benefits are solely to do with that cost, success proportion had very little use. A genetic programming practitioner should consider this limitation before comparing such measurements in others' research.

An excellent example of the benefits of confidence levels involved the use of the minimum computational effort measure. When asked "which was better: automatic incremental evolution or direct evolution?", minimum computational effort gave an average efficiency ratio of 1.07—that's an almost 10% performance improvement. But when asked "how confident are you that a difference truly exists?", an average confidence of just 51% was given. Confidence levels offer a useful tool for us to value a measurement; without them we are forced to accept comparisons at face value—in this case we would have expected an almost 10% improvement and had no expectation that we might well see a reduction in performance.

9.8 Summary

We have shown that fitness-based automatic incremental evolution can equal direct evolution. If you are prepared to accept that one generation does not necessarily have the same cost as the next, and that the cost is proportional to the number of evaluations executed, then automatic incremental evolution, statistically, significantly outperformed direct evolution.

It is an open question whether the technique offers performance improvements on other domains, but if the cost of failure is a considerable portion, then this technique may well prove beneficial.

9.9 Future Research

As with the manual incremental experiments, the greatest issue with this work is that it studied only one problem domain, even-n-parity. These techniques would have to be trialled on other domains before any general conclusions could be reliably produced.

Also, as with the manual incremental experiments, we have considered only one ordering of the fitness cases. It would be interesting to study the impact of alternative orderings.

The mutation of 50% of the population was a high rate compared to traditional uses. The frequency of application however was fairly low and dependent on the number of generations before an automatic step ($g_{\max}$). However because mutation at such a high level proved beneficial, we have yet another parameter to adjust. Researchers applying fitness-based automatic incremental evolution may well wish to study the associated performance impact of varying the quantity of mutation.

Equally, mutation might not be the most appropriate diversity-enhancing operator. Perhaps it is worth considering Winkeler and Manjunath's approach (see section 6.2.5) and move to the next stage when, say, 90% success was attained (rather than the 100% required in this work).

Further, in this study we did not analyse the impact of mutation on direct evolution. Mutation is known to be beneficial for the even-n-parity problems [97], so it would be interesting for future research to see if—like ADFs—the use of mutation offers more to incremental evolution than it does to direct evolution.

When offering a new approach, the desire is that it will be able to extend the useful scope of the field. Will fitness-based automatic incremental evolution be useful to allow genetic programming to tackle hard problems? We expect that the answer is "Yes". By decreasing the cost of failure and potentially improving the likelihood of success, it may well decrease the effort required. Appendix A looks at the issues of success-based problems, but there are plenty of hard problems that do have a known definition of success. Many such examples can be found at the UCI machine learning repository [11]: two-thirds of their problems are classification problems that may be excellently suited to this technique.

We have already brought up the issue that the final generation (or indeed, stage) may not assess the population using the full set of fitness cases. There are three approaches to resolve this. The first is to consider the issue a concern. The second is to consider it an advantage. The third is to make use of that fact that success is impossible.

If continued use of the automatic incremental evolution method is guaranteed to fail then our first approach is to consider it a concern. We could elect to "panic" and, for example, reserve the final five generations for the goal stage. This would at least give GP a chance to succeed. However the likelihood of success has got to be considered remote: what are the chances of successfully solving the full problem when a subset has remained out of reach?

The second approach is to consider that it is an advantage to not necessarily arrive at the goal stage. After all, when the full set of fitness cases isn't used then it takes fewer evaluations to assess a population and given that we know it's not going to succeed then the fewer evaluations the better.

The final approach is to think, if we concluded "the fewer the better", then surely it would be better still not to evaluate the population at all. It is that thinking that motivated the next chapter: if we know there is no hope for the final generation to succeed, then there is no point evaluating it and we should terminate early. However, we can do even better than that.

Chapter 10

Early Termination

In this chapter we introduce a heuristic that focuses on reducing the cost of the runs that fail. It is first applied to canonical (direct evolution) genetic programming and then to automatic incremental evolution. It is the use of the success effort measure that allows us to see the improvements in performance.

10.1 Motivation

My father used to tell me that, in an exam, the first 50% of the marks were easier to obtain than the second, and that the last portion of the marks were the most difficult to obtain. Using such thinking we might suspect that there is a "minimal learning curve" for GP to succeed: if the system spent too much time on the easier initial sets of fitness cases then there would not be enough time for the remaining cases. Such a curve would show the greatest gains early on and, as the generations increase, the size of the gains would decrease. In other words the rate of learning would decrease as the generations increased.

Equally, one might argue that the first fitness cases are the hardest as they are completely foreign. As a result GP may spend the most time working on those in order to form general solutions that will serve it well for the later fitness cases. Such a learning curve would have a rate that increased as the generations increased.

We can probably discard that latter learning style; such thinking might be appropriate for a design-based approach, but evolution relies on an ability to incrementally improve. It is unlikely that evolution would take a path that was slow to perform at the start, but would pay dividends at the end. It is far more likely to take a path with higher performance at the start. Although not entirely greedy, evolution can't look very far ahead.

10.2 Defending the Early Termination Heuristic

Although that theoretical argument is very useful, we can also call on empirical evidence to assist with this discussion. We can ask, which of the two approaches does GP typically take? We will use two different sources: four large datasets and two of Koza's books.

10.2.1 Four Datasets

We can once again recruit the four datasets that were used in chapters 3, 4, and 5: Ant, Parity, Multiplexor, and Symbreg. For these four problems we can look selectively at the successful runs and analyse their learning curves.

But how do you determine the change of rate of learning? The learning curve would consist of the best-fitness obtained for each generation, but the task of determining the rate of learning is made more difficult by the natural bumps and plateaus that one would expect from experimental data: for example, even if an exponential learning curve were observed on average, that curve will never be followed exactly for each experiment. One solution to the problem is to draw a straight line from each curve's start point to its end point and measure the proportion of the curve that is above the straight line: the greater the proportion the more the curve has a decreasing learning rate. The solution has a minor problem in that it does not give credit for "learning" that has occurred in the first generation. To accommodate for that, the start point for the straight line should be lowered to the average performance of the first generation—that point represents the fitness one can expect from a random population. The end point remains the same; it is the fitness obtained by the best individual in the final generation of the run. The left graph in figure 10.1 provides an example.

Armed with that measure, we studied the Multiplexor domain. On average, for each of the 985 successful runs, 94% of each curve's points were above the line from average fitness. However only 30% of the curves were entirely above that line. Although 30% is low, one measure that may give perspective is the minimum proportional increase of the line's end-point generation that was required to ensure the curve was entirely above the line. The right graph in figure 10.1 demonstrates this pictorially. Extending the line from average fitness 8.5% along the x-axis (generations) meant that 95% of the Multiplexor curves were above the extended line. Concretely, that equated to the line's end-point occurring at

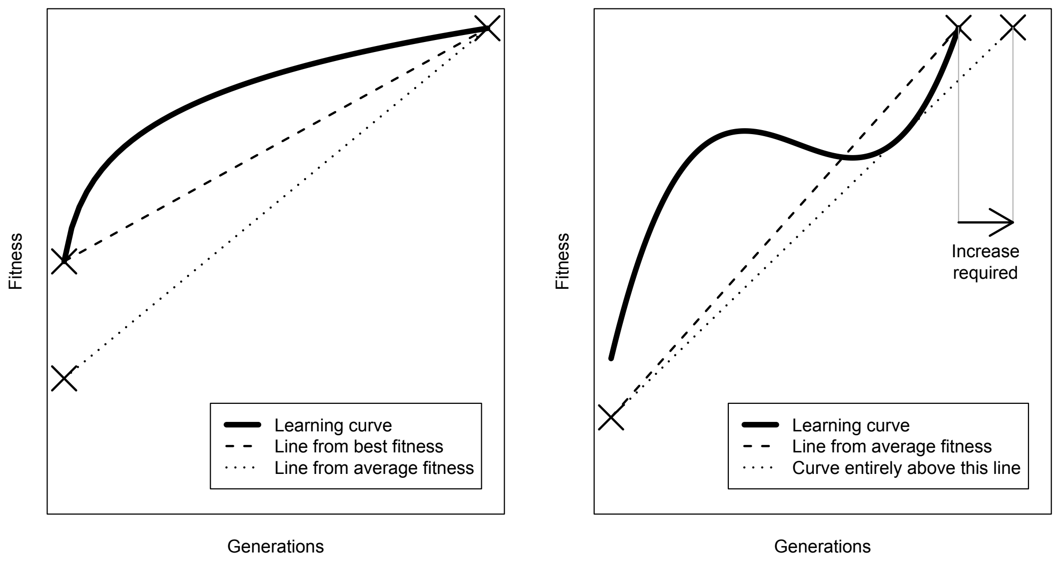


Figure 10.1: Left graph: Lines used to analyse change in rate of learning. Right graph: Lines used to assess proportional increase required for learning curve to be entirely above the straight line.

generation 26; or generation 37 for 99% inclusion. Given that the experiments were run to 50 generations, this was not a problematic increase. Table 10.1 gives the results for the other problem domains.

(Unfortunately, Christensen's dataset did not include average-fitness information. In order to study that domain, we executed 6,000 runs of the same artificial ant problem domain. We refer to this dataset as *Ant2*. 1,183 (20%) of the 6,000 runs found a 100%-correct solution. The difference in performance between Ant2's 20% success rate and Ant's 13.3% must be explained by implementation details between ECJ and Open BEAGLE, given that their configurations were as similar as possible.)

For each of the problem domains we can easily draw straight lines that include 95% of the learning curves. Indeed, for three of the four domains we can even include 99% of the learning curves. (The Parity domain, at 51 generations, just escapes the maximum of 50 generations.) From this study we can conclude that the vast majority of every successful learning curve was above linear.

	Ant2	Parity	Symbreg	Multiplexor
A	95%	88%	97%	94%
B	70%	0%	75%	30%
C (95%)	29%	17%	6.2%	8.5%
D (95%)	41	23	20	26
D (99%)	51	31	39	37
E	1,183	2,392	726	985

Table 10.1: Successful learning curves' performance relative to a straight line plotted from the average of the first generation to the final generation of the curve. A: Average proportion of each curve above the line. B: Proportion of learning curves *entirely* above the line. C: Maximum proportional increase in generation of line's end point required for 95% of curves to be entirely above line. D: Largest generation required for 95% (or 99%) of curves to be above line. E: Number of successful runs.

10.2.2 Koza's Books

Koza's first and second books contain what he called graphs of "standardised fitness" for the problem, where "standardised fitness restates the raw fitness so that a lower numerical value is always a better value" [71, page 96]. These graphs plot best-, average-, and worst-fitness per generation for one run of the given problem. Adjusting for the fact that smaller fitness scores were better, these graphs can be used to answer the question "did the learning rate decrease?".

There are 19 such graphs in Koza's first book [71], four of which we discarded given that they had too few generations, were artificially extended, or used a log scale. That left 15 graphs which included an array of problem domains: cart centering, the artificial ant on the Santa Fe and Los Altos trails, 11-multiplexor, broom balancing, the truck backer-upper problem, lizard foraging, wall-following, recursive sequence induction, intertwined spirals, even-4-parity, and box moving. All of them demonstrated learning rates that decreased as the number of generations increased and, just as with the four large datasets, the vast majority of the curves' points were above the straight line from average fitness.

There are 14 "standardised fitness" graphs in Koza's second book [72]. Again, the range of problem domains was significant: the two boxes problem, even-6-parity, the 64-square lawnmower problem with and without ADFs, the impulse-response problem, the artificial ant on the San Mateo trial, letter recognition, five different versions of the transmembrane problem, and two versions of the omega-loop problem. Again all of them had decreasing learning rates with the vast majority of each curve above the straight line from average fitness.

The single negative example we observed [71, page 580] was on the first half of Koza's "biathlon", The symbolic regression of $x^4 - 5x^2 + 4$ was solved within 10 generations and, when measured in terms of hits, had an *increasing* learning rate as the generations increased.

Koza's two books have provided 29 from 30 examples of decreasing learning rates as the generations increase. The vast majority of all of the curves, bar only one, were above the straight line from average fitness. Although this result is very strong, there are a few considerations: (i) it seems the majority, but not all, the graphs were from successful runs, (ii) a successful run may not have found the optimum solution, but then for some problems the optimum was not known, (iii) at least some of the problems had non-linear standardised fitness. Each of these considerations may impact the curvature of a learning curve, but overall the result lends weight to the argument that a successful learning curve typically has a decreasing rate of improvement.

10.3 The Heuristic

Those two empirical studies have shown that there exists a straight line that the vast majority of successful runs remain above. Unfortunately it is not possible to predict the definition of that line *before* the execution of the run. However, we can construct a more conservative line. A line that starts at the origin is guaranteed to be no higher a start point than the average performance of generation zero so long as negative fitness is impossible. A line that terminates at the definition of success at the maximum generation would also be more conservative so long as the maximum generation is large enough—a decision made by the user as a prerequisite of beginning a GP run.

If we construct such a line then we would know two things: (i) from the empirical evidence, the vast majority of successful runs would remain above it and (ii) because the line ends at the definition of success, all runs that failed would at some point cross the line. Thus, we can use such a line as a termination criterion for an early termination heuristic: if a run "crosses the line" we should hold little hope for its eventual success and so terminate it immediately.

10.3.1 Related Work

This suggestion is not the only early termination heuristic that has been offered. We will now discuss the heuristics we have encountered in published work with a focus on how studies of such heuristics have analysed the impact on performance.

In Koza's third book on GP he confesses to *manually* monitoring runs and terminating them when a human feels it's a good idea: "our uniform practice for problems of circuit synthesis in this book is to set the maximum number of generations, G, to some arbitrary large number (e.g. 501) and to manually monitor and manually terminate the run." [73, page 455]. Koza detailed that "manual termination" resulted from "manual observation that the values of fitness for the best-of-generation individual and other individuals in the population appear to have reached a plateau" [73, page 189]. Koza appears not to have attempted a comparison of the cost of applying GP with and without manual termination. It is however noteworthy that Koza found the concept of early termination sufficiently important to have a human spend their time watching the computer's progress.

In *Genetic Programming: An Introduction* [13, section 10.1.1], Banzhaf et al. mention five ways to improve the speed of GP, the first of which is early termination of a run. They suggested that the explosion of introns—which "later in the run ... tend to grow exponentially and to comprise almost all of the code in an entire population" [13, page 182]—could be used as a termination criterion. They cited Francone, Nordin and Banzhaf's work, which used as a termination heuristic the occurrence of destructive crossover (crossover that decreases the child's fitness when compared to its parent's). Francone et al. noted that the use of this criterion resulted in 52% of their runs being terminated before reaching even half of the allocated maximum number of generations, producing a reduction (by more than half) of the number of generations executed [40]. Unfortunately, it appears it would have been too expensive for them to analyse the impact their heuristic had on the quality of their best solutions.

Kramer and Zhang developed a genetic programming system they called GAPS. In it they specified a termination criterion based on performance of the best individual in a generation. "During each generation in which no improvement in the top score occurs, a counter is incremented. When this counter reaches a user-defined threshold, the run is signalled to shut down" [75]. Although GAPS development continued (under the name GP-Lab [47, 48]), we found no performance comparisons that analysed its termination heuristic.

Gianluigi Folino and Giandomenico Spezzano implemented a peer-to-peer distributed GP system called P-CAGE [39]. They discussed three termination criteria: a maximum number of generations, a maximum amount of time, and a criteria based on the computational effort carried out by the system. Their studies, however, did not analyse the performance offered by the different methods.

Other work has used early termination in an attempt to avoid over-fitting [78]—a concept often considered in other machine learning techniques such as artificial neural networks [45]. But again—at least for the work in GP—we have not encountered studies of performance comparisons.

Finally, it is almost a certainty that a number of other researchers will have used early termination heuristics without significant consideration. For example, Shichel et al. evolved a tank-game playing "Robocode" player [104] and they "simply stopped the run manually when the fitness value stopped improving for several generations". They offered no discussion of their decision, and no comment on its effect on performance.

The following studies look at the impact of our straight-line early termination policy. The focus is on how it affects performance but the methodology offered could potentially be applied to analyse the heuristics that others have used.

10.4 Direct Evolution Experiments

The early termination heuristic is intended to decrease the cost of failure but that benefit comes with the potential cost of also decreasing the probability of success. To measure performance in such situations, we have shown that the success effort statistic is the best of our three measures. We will now use that measure to empirically study the effect on performance of our straight-line early-termination heuristic.

Our five large datasets were again brought to use. Given the best-fitness score of each generation for each of the runs, we post-processed the results to calculate what the success effort would have been had the early-termination heuristic been used. 50 generations was used as the maximum (although Gagné's Symbreg and Multiplexor experiments originally had a termination criterion that only approximated this, the difference is negligible).

Table 10.2 gives the results. We can compare the two success effort measures for each of the problem domains and, by the simple variation (described in section 4.2) of the algorithm in table 4.7, analyse the confidence that we should have that the true success efforts are indeed different. The Symbreg results lead the comparisons: early termination more that halved the required effort—with a success effort efficiency ratio of 2.20—and produced a 100% confidence that it was an improvement.[1] Ant and Ant2 also produced "100% confidence" with

[1] It should be remembered that this is a simulation algorithm—although 100% may be slightly overstating the level, the result gives *very* strong indication that there truly is a difference.

Problem	Termination	Success effort	Success proportion	Cost of success	Cost of failure
Ant	Standard	344	13.3%	17.4	50.0
	Early	276	13.0%	16.8	38.7
Ant2	Standard	220	19.7%	15.9	50.0
	Early	182	19.5%	15.9	40.0
Parity	Standard	15.93	99.7%	15.8	50.0
	Early	15.92	99.7%	15.8	48.0
Symbreg	Standard	31.8	72.6%	9.5	59.2
	Early	14.4	72.2%	9.3	13.4
Multiplexor	Standard	17.9	98.5%	17.1	54.7
	Early	17.8	98.5%	17.1	48.0

Table 10.2: Success efforts for experiments using the early-termination heuristic and for experiments using only the standard termination criteria. Also included are the final-generation success proportions, average cost of success (in generations) and the average cost of failure (in generations).

efficiency ratios of 1.25 and 1.21 respectively. Multiplexor and Parity had much lower success effort efficiency ratios (only just above one) and thus lower confidence levels (62% and 52% respectively), but it should be noted that the heuristic had very little room to perform given both problems' very high success rates.

There are a number of points worth noting: (i) the use of the heuristic decreased the success effort for every one of the five problems, (ii) larger decreases in success effort were associated with a lower probability of success, (iii) the reduction in the probability of success was very small in all cases, (iv) as well as decreasing the cost of failure, the heuristic also slightly decreased the cost of success.

From these experiments very few disadvantages but potentially highly-significant advantages were seen. It is reasonable to assume that this approach might be generally applicable, but further research (see section 10.7) would give practitioners greater confidence in its generality.

10.5 Automatic Incremental Evolution Experiments

Chapter 9 demonstrated that the primary advantage of automatic incremental evolution was that it decreased the number of adjusted generations required to find a solution—a reduction in the number of fitness evaluations for failed runs.

A consequence was that the potential fitness scores were held down if the run was not succeeding. Figure 10.2 demonstrates this difference.

This feature of automatic incremental evolution becomes very interesting when viewed from the perspective of the early-termination heuristic. Given that the fitness is held down, how does that impact the performance under early-termination? The remainder of this chapter considers that question.

We compared the performance of automatic fitness-based incremental evolution with and without early termination. The comparison was made both with and without adjusted generations. Comparisons to direct evolution with early termination were also made. ADFs were used in both incremental and direct evolution experiments.

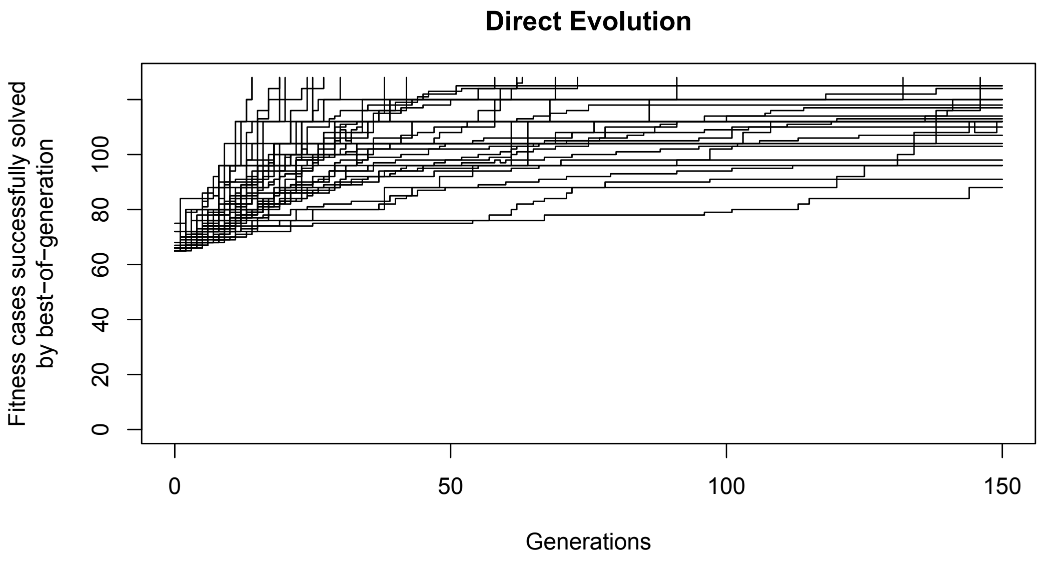
To do this we post-processed the even-n-parity experiments from chapter 9. Runs whose performance fell below a straight line were reclassified as failures and their generations-to-termination was relabelled as the generation at which the performance first fell below the line. The straight line was defined as starting at generation zero with zero fitness cases successfully solved, and ending at 150 generations with 100% of the fitness cases successfully solved. Again, as with the direct evolution experiments in section 10.4, because the benefit of early termination is primarily in the reduction in cost-of-failure, we have elected to make comparisons using only the success effort measure.

10.5.1 Results: Unit-Cost Generations with Early Termination

This section discusses the impact of early termination on automatic fitness-based incremental evolution under the assumption that generations have equivalent costs—as in chapters 8 and 9, the performance was much better under adjusted generations which we will consider in section 10.5.2.

On average, automatic incremental evolution with early termination had a success effort efficiency ratio of 1.52 compared to automatic incremental evolution without early termination. On average, we should have a confidence of 91% that automatic incremental evolution with early termination was the better choice. None of the 128 different configurations had an efficiency ratio less than one.

When compared to direct evolution (with early termination), on average, automatic incremental evolution with early termination had a success effort efficiency ratio of 1.14. Automatic incremental evolution was the better choice with an average confidence level of 62%. 87 (68%) of the 128 configurations had a ratio above one.



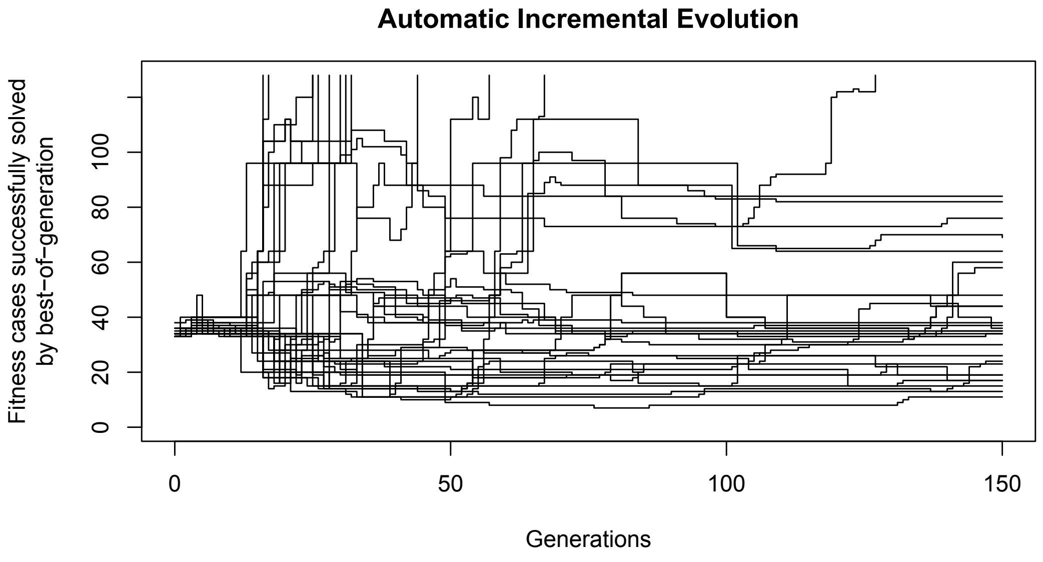


Figure 10.2: Forty runs of the number of fitness cases successfully solved by the best individual in each generation using direct evolution (upper graph) and using automatic incremental evolution (lower graph; with a maximum of five generations before an automatic step) on even-7-parity.

Aggression

Early termination offered a greater relative improvement for the less-aggressive strategy (with an efficiency ratio of 1.69 versus 1.35), but when compared to direct evolution the aggressive strategy was still the better choice (with an efficiency ratio of 1.20 versus less-aggressive's 1.08).

Mutation

The use of mutation offered little average relative performance difference (1.55 versus 1.49) meaning that, when compared to direct evolution, mutation was still the better choice (1.22 versus 1.06). Thus, the most effective technique was to use the aggressive strategy with mutation to give an efficiency ratio of 1.27 over direct evolution with early termination.

Generations Before Step

The average relative performance appeared to decrease fairly linearly as g_{max} increased. With a step size of 5, the relative efficiency ratio was 1.66 while for 50 generations before an automatic step the ratio was 1.45. It is likely that this performance gain is from smaller step sizes lowering the number of fitness cases faster for failing runs.

The mean efficiency ratio was 1.59 for aggressive automatic incremental evolution with mutation and a step size of 5.

Problem Difficulty

The average relative performance gains generally increased as the difficulty increased. For aggressive automatic incremental evolution with mutation and a step size of 5, the greatest improvement was seen in even-6 (with an efficiency ratio of 2.11) and even-7 (with a ratio of 1.84). Even-5 saw a ratio of 1.12.

10.5.2 Results: Adjusted Generations with Early Termination

This section discusses the impact of early termination on automatic fitness-based incremental evolution under the assumption that generations have costs dependent on the number of individual-evaluations required.

The results were *very* similar to the comparison without adjusted generations. On average, automatic incremental evolution with adjusted generations and with early termination had a success effort efficiency ratio of 1.54 compared to automatic incremental evolution without early termination. On average, we should have a confidence of 93% that automatic incremental evolution with early termination was the better choice. All of the 128 different configurations had an efficiency ratio greater than one.

However, when compared to direct evolution (with early termination) there was a stark difference. Without adjusted generations the two options were fairly close. With adjusted generations the success effort efficiency ratio averaged 2.16—a more-than-100% performance improvement, with an average confidence, in automatic incremental evolution being the better choice, of 97%. 126 (98%) of the 128 configurations pointed to automatic incremental evolution.

Aggression

Relative performance was greater with the less-aggressive option (with an efficiency ratio of 1.69 versus 1.38 over automatic incremental evolution without early termination). That helped make the less-aggressive option the better choice when compared to direct evolution with early termination (efficiency ratios of 2.50 versus 1.83).

Mutation

Like the unit-cost experiments, there was little distinction in relative performance between the use of mutation (efficiency ratio of 1.56 versus automatic incremental evolution without early termination) and not using mutation (1.52). But when compared to direct evolution, the use of mutation was, on average, a better decision (2.32 versus 2.01).

The use of the less-aggressive strategy along with the use of mutation gave an efficiency ratio of 2.72 over direct evolution with early termination.

Generations Before Step

Again an almost linear result was seen regarding the number of generations before an automatic step: a step size of five showed the greatest advantage on average (1.75) while a step size of 50 showed the least (1.38).

However the less-aggressive option with mutation had the best performance with a step size of 5 (with an efficiency ratio versus direct evolution of 4.71) while a step size of 5 produced a ratio of 2.86.

Problem Difficulty

The relative performance compared to not using early termination appeared to roughly increase as the problem difficulty increased. The less-aggressive strategy with mutation and a step size of 10 had an efficiency ratio over direct evolution (with early termination) of 3.0 for both even-6 and even-7 and 1.6 for even-5.

10.6 Summary

This chapter introduced an early termination heuristic. We considered its impact on both direct evolution and automatic fitness-based incremental evolution.

For direct evolution the heuristic reduced the cost of using genetic programming (by decreasing the success effort). Further, for automatic incremental evolution we showed use of the heuristic offered an efficiency ratio of 1.5—a 50% improvement over not using it. Finally, we showed that automatic incremental evolution with early termination and adjusted generations had a very-significant efficiency ratio of 2.16 over direct evolution.

From these studies we can have some confidence that the heuristic is a good idea and that it could well be generally applicable, but further studies would increase that confidence.

10.7 Future Research

No effort was made to optimise the line that defined the early-termination heuristic. For direct evolution there is significant scope for improvement. An immediately obvious option could be the use of average fitness in the first generation (rather than zero fitness) for the definition of the start of the line. Initial studies showed a notable direct-evolution performance improvement, but such a modification to the line's definition would have an impact on the heuristic's applicability to automatic fitness-based incremental evolution. Also unless a theoretical defence was offered, any optimisation would require empirical evidence from a significant body of GP runs.

It would be interesting to record the proportion of GP runs that would have benefited from the use of the early termination heuristic. Although the GP practitioner may feel it is currently too risky and unproven an approach, low-cost post-processing of their results would show whether the technique would have been beneficial. If such results were collected and—just as was shown here—the majority pointed to a benefit, it would certainly give practitioners an increased confidence in the use of the heuristic.

Finally, manual incremental evolution would most likely also benefit from this heuristic. No attempt was made to study that, but the technique holds potential that it might improve manual incremental evolution's performance above direct evolution's.

Part III

Finale

Chapter 11

Conclusions, Contributions, Limitations, Future Research

11.1 Conclusions

The work in this thesis can be viewed from two perspectives. The first is that of the statistics offered in Part I; in that case the experiments in Part II can be seen as examples of the statistics' use. The second perspective is that of the incremental evolution experiments; taking their viewpoint makes the statistics a useful tool to assess the experiments. We feel both viewpoints are equally valid.

If we take the second viewpoint, that of the experiments, we can say that without the use of the statistics the analysis of the experiments would have taken the traditional route: no confidence intervals would have been offered and so little idea could have been formed as to what variability one might expect. The use of the statistics allowed us to confidently state when fitness-based automatic incremental evolution reduced the effort required to find a solution when compared to direct evolution. The statistics allowed us to offer both a best estimate of the potential size of an effect and a confidence that one might have that there truly was an effect at all—both very useful quantitative descriptions.

The use of success effort opened up the possibility of benefiting from a reduction in the cost of failure. Traditional use of final success proportion certainly cannot show this benefit, but in many cases neither can Koza's minimum computational effort. If one is prepared to assess cost based on evaluations of individuals, then this measure has shown us potential benefits from manual and automatic incremental evolution. It has also shown us the benefits offered by the early termination heuristic.

If we instead take the viewpoint of the statistics, the chapters on automatic incremental evolution offered concrete evidence that the work on confidence interval production was very valuable, as were the confidence level algorithms that detected if a difference truly existed. The experiments showed up the upper-bound flaws of minimum computational effort and comparable excellence of success effort.

11.2 Contributions

This thesis offers the following contributions:

- A method for producing confidence intervals for Koza's minimum computational effort and empirical evidence that it achieves appropriate coverage and is reliable. We also offered an associated method to produce confidence intervals for the difference and the ratio of two minimum computational effort measures, and an algorithm for how one could find the confidence that one measure truly had a lower minimum computational effort than the other.
- A method for producing confidence intervals for the success effort statistic and empirical evidence that it achieves appropriate coverage. For this measure we also offered an associated method to produce confidence intervals for the difference and the ratio of two success effort measures, and an algorithm for how one could find the confidence that one measure truly had a lower success effort than the other.
- Two algorithms for the automation of fitness-based incremental evolution. We showed that these methods could outperform direct evolution.
- An early-termination algorithm that we showed reduced the cost of failure for both direct and incremental evolution experiments.

11.3 Limitations

Confidence intervals for success proportions will have appropriate coverage rates so long as sufficient (at least five) runs were executed.

The confidence interval methods for Koza's minimum computational effort should be applicable to all genetic programming runs where the generations-to-success follows a normal, log-normal, or similar distribution—we even showed the Wilson-Dependent method was reliable under four very extreme distributions. We hypothesise that these cases cover all genetic programming experiments, but with the limitation that success is sufficiently common to make the confidence intervals useful.

For the success effort measure, the simulation method relies on the distributions of mean-generations-to-success and mean-generations-to-failure and success proportion. The assumptions of these distributions are all underpinned theoretically (thanks to the Central Limit Theorem, and that proportions can be simulated with a Beta distribution), so the method relies only on sufficient runs (say, at least 25) and again, like minimum computational effort, that success is sufficiently frequent that the confidence intervals are useful.

For the experiments on manual and automatic incremental evolution, only the even-n-parity problem domain was studied. The results are almost certainly going to vary by problem domain, so it will be interesting to see how these ideas perform elsewhere—especially in "continuous" domains (given even-n-parity is discrete), and in more difficult real-world problems (such as regression, classification, and predictions problems). Further, we studied only one ordering of the fitness cases. Different orderings are highly likely to impact the results. This too would be an interesting study. Finally, the technique, as described, is only useful for problem domains with multiple fitness cases.

The early termination algorithm was tested on four different domains under direct evolution, but only on even-n-parity for incremental evolution. We provided evidence that it is likely that evolutionary learning curves follow a decreasing improvement in fitness as the generations increase. We believe this could be generally true, but further studies would be required. We hypothesise that the early termination algorithm would be beneficial for the majority of genetic programming experiments. However, it is quite possible that there exist fitness evaluation functions that would not produce a "decreasing learning curve" and would thus not benefit from the algorithm.

11.4 Future Research

Throughout this thesis there are a number of sections titled "future research". If you are interested in extending a specific part of the work in this thesis then those sections are the best place to start.

In general however, we hope that future research in genetic programming utilises the success effort statistic when studies of success-based problems are executed. We have shown that success proportion is not always a good measure and that minimum computational effort has a number of limits. These issues are solved by the success effort statistic, so we hope it achieves greater use.

We appreciate that, for “hard problems”, it may not be appropriate to use a success-based measure. For such work, we hope that the future research options discussed in appendix A are seriously studied. If the research builds on the work we did on success effort then, in my eyes, that would be a fantastic development.

Part IV

Appendices

Appendix A

Success Proportion versus Fitness

Luke and Panait published a paper at GECCO 2002 where they argued that ideal-solution counts should not be used [85]. Their recommendation was to instead use fitness as a measure of performance. Given that this thesis is built entirely on the use of these counts, we will now discuss their paper. This appendix highlights a number of concerns with their argument and concludes that ideal-solution counts do have value.

Their argument can be divided into three parts. The first part is their statistical concerns regarding the use of ideal-solution counts. The second is their empirical evidence showing that mean best-fitness and ideal solution counts are not correlated. The third is their issues with the philosophical motivation for the use of ideal-solution counts. We tackle each in turn.

A.1 Statistical Concerns

Luke and Panait listed three statistical concerns that they had with ideal-solution counts and their associated measures. In the following sections we consider and discard each of them.

A.1.1 A Point Statistic

A point statistic (or point estimate) is a single number that is an estimate of a true value; it is calculated from a sample of observations. The mean of ten observations is an example of a point statistic. Point statistics are a problem as they give no indication of variability: they cannot answer "how accurate is this number?".

Luke and Panait claimed that "ideal solution count measures are statistically suspect. ... there is no accepted procedure to state that two samples differ in a statistically significant way." That is incorrect.

An ideal-solution count is only a part of the story. The full story must include the number of runs that were executed. This becomes clear when you consider if two ideal-solution counts can be directly compared; if the counts came from samples where the numbers of runs differed, then such comparison would not be acceptable. Instead, one would compare the ideal-solution counts as a *proportion* measured as the number of successes (ideal solutions) divided by the number of runs. This is even a measure that Luke and Panait discussed: cumulative success proportion.

Comparison of the raw confidence intervals (see section 2.2.2), or whether the confidence interval for the difference (table 2.2) includes zero or the confidence interval for the ratio (table 2.3) includes one, are three high-quality techniques that we have already considered for how one can statistically assess if two proportions differ. A fourth (but they admit, impractical) option is even given in their paper. Given these options, Luke and Panait's point-statistic concerns cannot be justified. The use of success proportions (and therefore ideal-solution counts) are statistically sound.

A.1.2 Statistical Independence

Luke and Panait next argued that the use of cumulative success proportion plotted *per generation* is statistically concerning because the measurements are typically statistically dependent. I have never read of a study where independent runs were executed to eliminate this issue, however, an option commonly used is to analyse only the final success proportion—an option that eliminates this dependence.

Interestingly, mean best-fitness of run—their recommendation—is also commonly graphed per generation. They make no mention that this too would suffer the same issues of dependence.

The solution is to use neither measure on a per-generation basis. Although Koza's minimum computational effort does depend on this, our studies (chapter 3) have shown this not to be a problem.

A.1.3 Koza's Minimum Computational Effort

Luke and Panait's final statistical concern was that Koza's computational effort is very sensitive to small changes in ideal-solution counts when the counts are low. This issue was raised by Miller and Thomson [87] and Niehaus and Banzhaf [92] (see section 2.3.4).

Fortunately the Wilson-Dependent confidence interval method developed in chapter 3 is sensitive to this issue and allows a user to understand the variability associated with minimum computational effort measurements that have come from low success rates. With the use of these confidence intervals we no longer need to be concerned that we may be mislead by Koza's measure.

A.2 Empirical Evidence

Luke and Panait looked at three problems: symbolic regression, 11-bit boolean multiplexor and the artificial ant. They claimed to have presented "evidence that ideal-solution counts are not necessarily positively related to best-fitness-of-run statistics: in fact they are often inversely correlated." They concluded, "this begs a re-evaluation of much of the GP literature, as published results may be dubious, and in some cases the opposite of their intended meaning."

We re-used the results from our experiments on manual and incremental evolution to provide new evidence for this discussion. We also reanalyse their results, and finally run a set of experiments in an attempt to understand the negative correlation that Luke and Panait saw. We conclude that although negative correlations are possible they are not as common as Luke and Panait claimed.

A.2.1 Re-Using the Incremental Evolution Experiments

The experiments on manual incremental evolution (chapter 8) and automatic incremental evolution (chapter 9) provided an excellent opportunity to reproduce Luke and Panait's analysis on different data.

For each run in each experimental configuration we obtained the maximum fitness score produced by an individual. For each experimental configuration we found the correlation coefficient of the mean of those best-fitness scores and the final success proportion (using Pearson's product-moment method [24]).

The results showed quite the opposite of "often inversely correlated". They were, in general, strongly positively correlated. Tables A.1 and A.2 give the results.

Problem	ADFs	Observations	Correlation	95% C.I.
Even-4 <4>	Yes	10	0.93	(0.74–0.98)
Even-4 <8>	Yes	10	0.78	(0.29–0.94)
Even-5 <8>	Yes	10	0.84	(0.44–0.96)
Even-5 <16>	Yes	10	0.69	(0.11–0.92)
Even-6 <8>	Yes	10	0.90	(0.63–0.98)
Even-6 <16>	Yes	10	0.89	(0.59–0.97)
Even-6 <32>	Yes	10	0.64	(0.02–0.91)
Even-7 <16>	Yes	10	0.79	(0.32–0.95)
Even-7 <32>	Yes	10	0.87	(0.52–0.97)
Even-7 <64>	Yes	10	0.75	(0.22–0.94)
Even-4 <4>	No	10	-0.12 ◇	(-0.70–0.55)
Even-4 <8>	No	10	0.89	(0.59–0.97)
Even-5 <8>	No	10	0.21	(-0.48–0.74)
Even-5 <16>	No	10	0.41	(-0.29–0.83)

Table A.1: Correlation coefficients for manual incremental evolution datasets. The number of fitness cases in the first stage are listed in angle brackets. Approximate 95% confidence intervals, in parentheses, were calculated using the normal approximation method [24, page 509].

Problem	Aggressive	Mutation	Observations	Correlation	95% C.I.
Even-4	Yes	Yes	8	0.80	(0.23–0.96)
Even-4	Yes	No	8	0.84	(0.34–0.97)
Even-4	No	Yes	8	0.54	(-0.26–0.90)
Even-4	No	No	8	0.88	(0.46–0.98)
Even-5	Yes	Yes	8	0.88 ◇	(0.45–0.98)
Even-5	Yes	No	8	0.69	(-0.02–0.94)
Even-5	No	Yes	8	0.60 ◇	(-0.18–0.92)
Even-5	No	No	8	0.95	(0.75–0.99)
Even-6	Yes	Yes	8	0.86	(0.40–0.97)
Even-6	Yes	No	8	0.87	(0.44–0.98)
Even-6	No	Yes	8	0.75	(0.10–0.95)
Even-6	No	No	8	0.72	(0.04–0.95)
Even-7	Yes	Yes	8	0.86	(0.38–0.97)
Even-7	Yes	No	8	0.84	(0.34–0.97)
Even-7	No	Yes	8	0.87	(0.42–0.98)
Even-7	No	No	8	0.65	(-0.10–0.93)

Table A.2: Correlation coefficients for automatic incremental evolution datasets. Approximate 95% confidence intervals, in parentheses, were calculated using the normal approximation method [24, page 509].

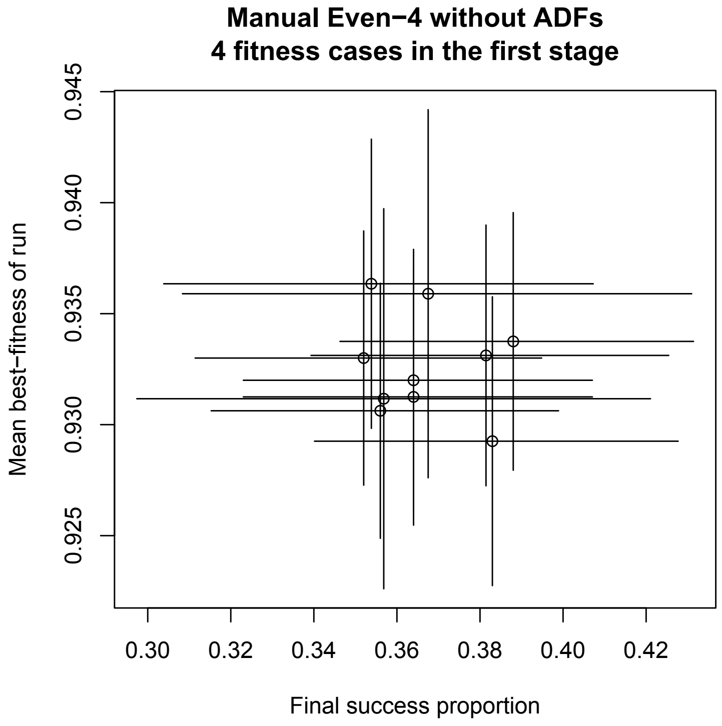


Figure A.1: Plot of success proportion at the final generation against the mean best-fitness. This was the only negatively correlated result with a correlation coefficient of -0.12. 95% confidence intervals are shown.

Only one of the thirty results was negatively correlated—and it can even be discounted. The negative result is plotted in figure A.1. From the plot you can see that the confidence intervals associated with each point are very wide relative to the other data-points. Indeed, all the data-points form one statistical group even with just 80% confidence. Because the data-points are fairly indistinguishable, it would not be unfair to consider them as a single data-point—at which stage no correlation can be obtained.

For a conclusive correlation result it's important that at least some data-points are statistically significantly different. All except three of the results within tables A.1 and A.2 have at least two groups that are statistically significantly different (on both axes) at the 80% level. The three exceptions are marked with diamonds ($\diamond$).

This short study has shown that the prevalence of negative correlations is perhaps not as high as Luke and Panait inferred with their article.

Problem	Observations	Correlation	95% C.I.
Symbolic regression	10	-0.69	(-0.92– -0.11)
Multiplexor	10	0.46	(-0.24–0.84)
Artificial ant	10	0.84	(0.45–0.96)

Table A.3: Correlation coefficients for Luke and Panait's datasets. Approximate 95% confidence intervals, in parentheses, were calculated using the normal approximation method [24, page 509].

A.2.2 Re-Analysing their Results

We will now reconsider Luke and Panait's results using the same method as we used in the previous section. Table A.3 lists the correlation coefficients obtained from their data.[1] Figures A.2, A.3 and A.4 plot ideal-solution counts against mean best-fitness.

The critical difference between these figures and those that Luke and Panait would have plotted, is the existence on these figures of confidence intervals for the ideal-solution counts. With the confidence intervals it is clear that the multiplexor results (figure A.3) have many potential lines that intersect every confidence interval: such straight lines would offer a correlation coefficient of (positive) one.

It is even easier to see the possibility for straight lines through the artificial ant plot (figure A.4)—lines that would intersect each of the confidence intervals and offer the potential for a true correlation coefficient of positive one.

Thus, of Luke and Panait's three datasets, only one—symbolic regression—has a negative correlation. This is quite some distance from their claim of ideal-solution counts being "often inversely correlated" to fitness.

A.2.3 Re-Considering Symbolic Regression

Why is it that Luke and Panait's experiments on symbolic regression produced such peculiar results? In an effort to replicate and then hopefully understand them, we ran a number of experiments on the same domain. Luke and Panait did not provide sufficient information to replicate their method (it was not their intention in the paper), so instead of using their multi-crossover parameter as a variable I elected to use population size.

We ran 46 experimental configurations with the population size ranging from 100 to 1000 individuals using a step size of 20 individuals. 500 runs were executed for each population size for a total of 23,000 runs. The configuration was

[1] My thanks to Sean Luke for providing me with a copy of their data.

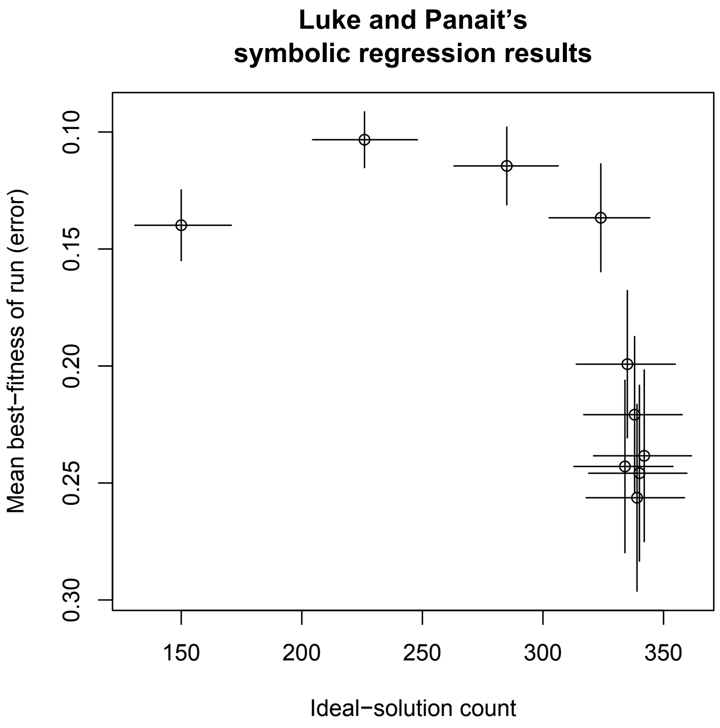


Figure A.2: Plot of success proportion at the final generation against the mean best-fitness for Luke and Panait's symbolic regression dataset. 95% confidence intervals are shown.

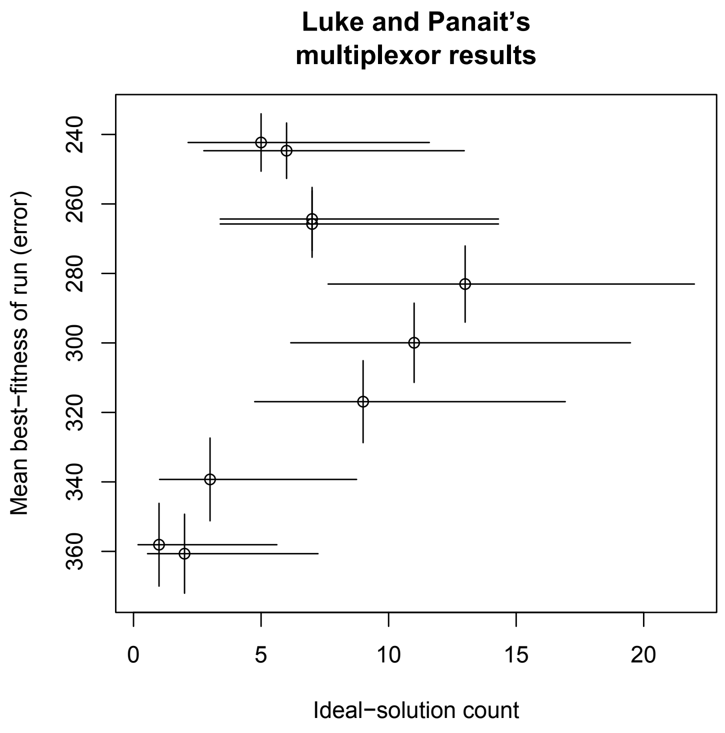


Figure A.3: Plot of success proportion at the final generation against the mean best-fitness for Luke and Panait's 11-bit multiplexor dataset. 95% confidence intervals are shown.

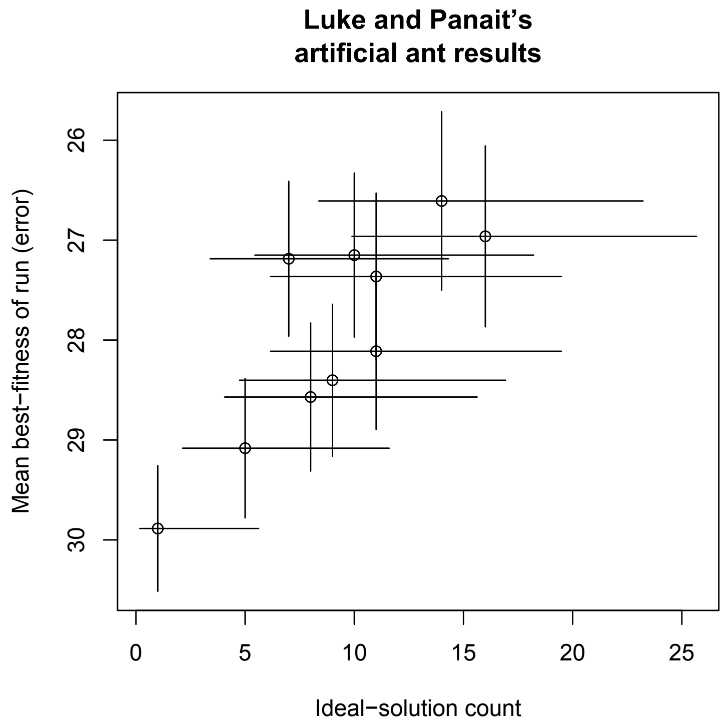


Figure A.4: Plot of success proportion at the final generation against the mean best-fitness for Luke and Panait's artificial ant dataset. 95% confidence intervals are shown.

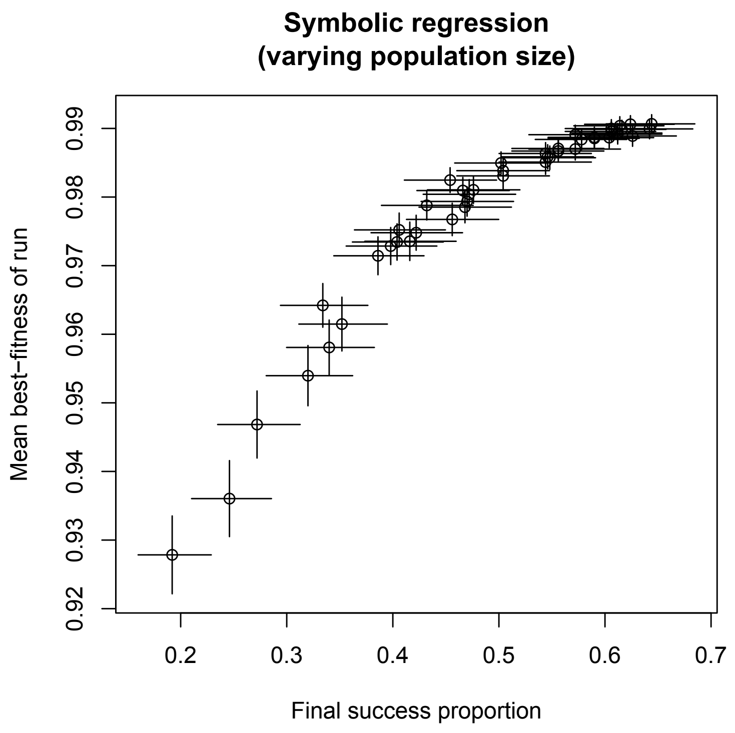


Figure A.5: Plot of success proportion at the final generation against the mean best-fitness for the symbolic regression problem where the population size is varied. 95% confidence intervals are shown.

intended to replicate Koza's $x^4 + x^3 + x^2 + x$ simple symbolic regression experiments [71, section 7.3] as closely as possible. The primary difference was the use of tournament selection with a tournament size of two.

Figure A.5 plots the results. They are nothing like Luke and Panait's.

The fact that my results differ completely from those observed by Luke and Panait can potentially be explained in a number of ways: (i) the experimental configuration may have been sufficiently different, (ii) population size may modify the correlation in a different way to multi-crossover, (iii) one of the results may have been an experimental fluke.

It is my opinion that the most likely of these explanations is the second: that the correlation effect is different for different variables. One lesson that can be learnt then is that the problem domain is not necessarily the sole cause of a negative correlation. The result also provides yet another example of a very strong positive correlation (0.94; 95% CI 0.89–0.97).

A.2.4 Rationalising Negative Correlations

If, as Luke and Panait expected, success is defined as a fitness score above a specified threshold, then it is intuitive that there should be a positive correlation between mean fitness and success proportion. There are, however, situations where the intuition does not prove true.

One such situation is where success proportion is at or near zero. If mean fitness decreases then, because it is not possible to decrease the success proportion below zero, the correlation will decrease to zero. A similar situation exists where mean best-fitness is close to its maximum. An example of this can be seen in figure A.5: although the success proportion increases, mean best-fitness cannot increase proportionally; a consequence is that the correlation asymptotically tends to zero.

As Luke and Panait explained, variance is likely to be the key to understanding negative correlations. If changing the variable (multi-crossover in their study, and population size in my study of symbolic regression) is able to increase the variability of mean best-fitness while decreasing the mean value only slightly, then the proportion of successes could still increase thus producing a negative correlation. It appears this is the effect multi-crossover had on symbolic regression.

Steffen Christensen also considered the correlation between success proportion and mean best-fitness in his PhD thesis [21, chapter 3]. His study showed that the two measures could not be interchanged. Indeed, he concluded that there were two types of problems: those where success proportion was appropriate and those where best-fitness was the better choice.

A.3 Philosophical Motivation

Luke and Panait had a number of concerns with the philosophical motivation around the use of ideal-solution counts. Their issues boil down to this: when applying GP to challenging problems you want to improve fitness, so fitness is what you should be working with. Unfortunately working with fitness is not necessarily easy. We will now discuss their arguments for the use of fitness and my arguments against it. This section ends with the a potential compromise: an important area for future research.

A.3.1 Pro Fitness

Luke and Panait's first argument was that GP's selection pressure is based on fitness and that it is not based on the probability of finding an ideal solution. They pointed out that "many GP problem domains are highly deceptive, leading the evolutionary trajectory away from the ideal rather than toward it"—as a consequence, improvements in fitness might not flow on to improved success proportions.

Their second argument was that GP practitioners suffer "a philosophical conceit that GP operates over problem domains which demand *correct programs* ... a highly fit but suboptimal solution is not valuable." This isn't true. The fallacy is exemplified by Koza's definition of a hit for his simple symbolic regression problem domain: a hit was defined as an individual producing a result within 0.01 of the correct answer [71, section 7.3]. Highly fit but sub-optimal individuals *are* valuable.

Their next argument was far more persuasive: "We are now out of the proof-of-concept period for GP ... we must assume it will typically be used to attack hard problems for which we do not know the optimum, do not expect it to discover the optimum, nor even know if there *is* an optimum." They list example domains such as neural networks, soccer softbot programs, and analog electrical circuits.

Finally, it is worth noting that, if electing to use best-fitness, Christensen recommends the use of median best-fitness rather than mean best-fitness as "fitness functions used in evolutionary computation are often non-linear, and sometimes have arbitrarily large penalty values. This pushes up the mean population fitness and can hurt the mean best fitness as well" [21, page 106].

A.3.2 Anti Fitness

Luke and Panait wrote: "What matters is not if technique A finds more perfect solutions than technique B does to Easy Problem C. What matters is that technique A gets a better answer than B does for Hard Problem D." Although this may be true they make no mention of the cost of each of techniques A and B. Surely they would be interested if A took tens times as much effort as B and yet only offered a 1% improvement. Christensen's y-test offers a method to compare two techniques where the amount of work done (as measured in generations, or evaluations) differs for the two techniques, however the method assumes that the amount of work done is constant—it does not take into consideration that the amount of work may instead follow a distribution [21].

Chapters 8 and 9 on manual and automatic incremental evolution have provided concrete examples of the number of generations-to-failure and generations-to-success being distributions rather than fixed values. Given that possibility, the use of best-fitness gives us two measures by which to evaluate if one evolutionary technique is better than another. How should we tell them apart? It is easy if the number of generations is the same, then we can judge based on fitness. If on the other hand the fitness values are the same then we can judge based on the number of generations. If both measures are better for one of the techniques then again the comparison is easy. But the problem comes when the two measures disagree. This remains an open problem for fitness-based comparison. Minimum computational effort and success effort provide solutions by answering the question based on the number of runs to find a solution. Fitness does not have such a solution.

Luke and Panait's argument against the use of success proportion included: "We submit that if one can 'discover' the optimum enough times to validly measure the performance of a technique against a given problem domain then we are dealing with a toy problem." This may be true, but "toy problems" are very useful. There is some amount of hypocrisy in Luke and Panait's inference that toy problems are not useful given they used three "toy problems" to study multi-crossover—the study that prompted their paper! Others too will use toy problems as test-beds for their new ideas. The fact that the problems are simple does not mean that they are a poor training ground before more difficult problem domains are tackled. The use of success proportion, minimum computational effort, and success effort offer an excellent platform to analyse such "toy problem" studies.

A.3.3 Future Research: A Compromise

Perhaps there is an acceptable compromise between difficult-to-compare fitness and potentially-insensitive success proportion.

One potential solution is the use of *binning*. This technique involves setting an arbitrary threshold at which a run is considered successful. Binning is not recommended by Christensen [21, page 81] but he considered it "common practice". One of Christensen's concerns was that "for improvement-based problems ... we are normally interested in achieving the best possible outcome"—binning removes our ability to detect such excellent performance.

Another potential solution could be to iterate over the range of observed fitness scores, executing say 100 steps. For each step the bin threshold would increase and the confidence that the success effort ratio was greater than one (see table 4.7) could be plotted for that threshold. This would give a graph that indicated the confidence one might have that technique A required less effort than technique B when producing individuals with fitness scores greater than X.

A third potential solution could be based on the probabilities of domination. A simulation algorithm could be constructed to assess the likelihood that technique A is superior, inferior, or indistinguishable to technique B based on domination or non-domination of both fitness and cost.

Yet another potential solution might be formed from a modification of Christensen's y-test [21].

Further research would more specifically define these ideas and study their relative statistical powers. Which, if any, is the best choice is an open question.

A.4 Summary

This appendix has considered the concerns raised by Luke and Panait regarding the use of ideal-solution counts and its related measures: success proportion, minimum computational effort, and success effort. Their statistical concerns were dismissed. Their empirical data was re-analysed and further data based on the incremental evolution experiments was offered. We concluded that they needn't have had such a negative association with ideal-solution counts. Finally their concerns with the philosophical motivation were considered and, although many were fair, we raised issues with fitness-based measures that make neither the perfect choice.

If your problem has a natural definition for success then ideal-solution counts are an excellent choice. If your experiments are not expected to find the optimal and the cost-per-run does not differ then best-fitness measures are the better choice. If the cost-per-run is not constant and ideal-solution counts are not acceptable then you are in a strong position to study some of the ideas suggested in "future research" (section A.3.3).

Appendix B

Proof of Equation 4.2

The following is a proof for equation 4.2 (page 75). We require only a few facts:

- the vector g (generations-to-termination) is comprised only of the two vectors g_{s} (generations-to-success) and g_{f} (generations-to-failure),
- the value p is defined as $\frac{\mathrm{n}(g_{\mathrm{s}})}{\mathrm{n}(g)}$ (where $\mathrm{n}(g)$ represents the number of elements in g), and
- $(1-p) = \frac{\mathrm{n}(g_{\mathrm{f}})}{\mathrm{n}(g)}$.

$$\begin{aligned}
\mathrm{mean}(g) &= \frac{\sum g}{\mathrm{n}(g)} \\
&= \frac{\sum g_{\mathrm{s}} + \sum g_{\mathrm{f}}}{\mathrm{n}(g)} \\
&= \frac{\sum g_{\mathrm{s}}}{\mathrm{n}(g)} + \frac{\sum g_{\mathrm{f}}}{\mathrm{n}(g)} \\
&= \frac{\mathrm{n}(g_{\mathrm{s}})}{\mathrm{n}(g_{\mathrm{s}})} \cdot \frac{\sum g_{\mathrm{s}}}{\mathrm{n}(g)} + \frac{\mathrm{n}(g_{\mathrm{f}})}{\mathrm{n}(g_{\mathrm{f}})} \cdot \frac{\sum g_{\mathrm{f}}}{\mathrm{n}(g)} \\
&= \frac{\mathrm{n}(g_{\mathrm{s}})}{\mathrm{n}(g)} \cdot \frac{\sum g_{\mathrm{s}}}{\mathrm{n}(g_{\mathrm{s}})} + \frac{\mathrm{n}(g_{\mathrm{f}})}{\mathrm{n}(g)} \cdot \frac{\sum g_{\mathrm{f}}}{\mathrm{n}(g_{\mathrm{f}})} \\
&= p \cdot \mathrm{mean}(g_{\mathrm{s}}) + (1-p) \cdot \mathrm{mean}(g_{\mathrm{f}})
\end{aligned}$$

Appendix C

Proof of Equation 4.4

In this appendix we will prove equation 4.4 (page 81). What the step says is that multiplying the curve $Y_{\mathrm{S1}}(g) + Y_{\mathrm{F1}}(g)$ by the straight line g and summing the results will produce a smaller (or equal) sum than if you were to do the same thing with $Y_{\mathrm{S2}}(g) + Y_{\mathrm{F2}}(g)$. This is quite reasonable as you are multiplying more of the first curve's values by smaller (or equal) values of g since the cumulative probability of success of curve P_1 dominates the cumulative probability of success of curve P_2. We will prove this by induction.

Let's replace Y_{F1} in equation 4.4 with a new function:

$$Y_{\mathrm{R1}}(g,k) = \begin{cases} 0 & \text{if} \quad g \neq k \\ 1 - \sum_{i=0}^{k} Y_{\mathrm{S1}}(i) & \text{if} \quad g = k \end{cases} \tag{C.1}$$

Replace Y_{F2} in a similar way. This gives us the following statement to prove:

$$\sum_{g=0}^{F} (Y_{\mathrm{S1}}(g) + Y_{\mathrm{R1}}(g,F)) \cdot g \leq \sum_{g=0}^{F} (Y_{\mathrm{S2}}(g) + Y_{\mathrm{R2}}(g,F)) \cdot g \tag{C.2}$$

To do this we will use two lemmas:

Lemma C.1 *From the definition of* $Y_{\mathrm{R1}}(g,k)$*, this statement can be made:*

$$Y_{\mathrm{R1}}(k,k) = \sum_{i=0}^{k} Y_{\mathrm{R1}}(i,k)$$

since $\sum_{i=0}^{k-1} Y_{\mathrm{R1}}(i,k) = 0$.

Lemma C.2 *For* P_1*:*

$$Y_{\mathrm{R1}}(k,k) \quad = \quad 1 - \sum_{i=0}^{k} Y_{\mathrm{S1}}(i)$$

$$\begin{aligned}
&= 1 - \sum_{i=0}^{k} Y_{\mathrm{S1}}(i) - Y_{\mathrm{S1}}(k+1) + Y_{\mathrm{S1}}(k+1) \\
&= 1 - \sum_{i=0}^{k} Y_{\mathrm{S1}}(i) - \sum_{i=k+1}^{k+1} Y_{\mathrm{S1}}(i) + Y_{\mathrm{S1}}(k+1) \\
&= 1 - \sum_{i=0}^{k+1} Y_{\mathrm{S1}}(i) + Y_{\mathrm{S1}}(k+1) \\
&= Y_{\mathrm{S1}}(k+1) + Y_{\mathrm{R1}}(k+1, k+1)
\end{aligned}$$

A similar statement can be made for P_2.

Let's return our attention back to equation C.2 and consider the case where $F = 0$; the summations are then from zero to zero. In such a case both sides will be zero, and the equation is true:

$$\sum_{g=0}^{0} (Y_{\mathrm{S1}}(g) + Y_{\mathrm{R1}}(g,0)) \cdot g = 0 \leq \sum_{g=0}^{0} (Y_{\mathrm{S2}}(g) + Y_{\mathrm{R2}}(g,0)) \cdot g = 0$$

Now assume it is true that:

$$\sum_{g=0}^{k} (Y_{\mathrm{S1}}(g) + Y_{\mathrm{R1}}(g,k)) \cdot g \leq \sum_{g=0}^{k} (Y_{\mathrm{S2}}(g) + Y_{\mathrm{R2}}(g,k)) \cdot g \tag{C.3}$$

We will now show that:

$$\sum_{g=0}^{k+1} (Y_{\mathrm{S1}}(g) + Y_{\mathrm{R1}}(g,k+1)) \cdot g \leq \sum_{g=0}^{k+1} (Y_{\mathrm{S2}}(g) + Y_{\mathrm{R2}}(g,k+1)) \cdot g \tag{C.4}$$

From equation 4.3 we can say:

$$\begin{aligned}
\sum_{g=0}^{k} Y_{\mathrm{S1}}(g) &\geq \sum_{g=0}^{k} Y_{\mathrm{S2}}(g) \\
1 - \sum_{g=0}^{k} Y_{\mathrm{S1}}(g) &\leq 1 - \sum_{g=0}^{k} Y_{\mathrm{S2}}(g) \\
Y_{\mathrm{R1}}(k,k) &\leq Y_{\mathrm{R2}}(k,k) \\
Y_{\mathrm{R1}}(k,k)((k+1) - k) &\leq Y_{\mathrm{R2}}(k,k)((k+1) - k) \\
Y_{\mathrm{R1}}(k,k) \cdot (k+1) - Y_{\mathrm{R1}}(k,k) \cdot k &\leq Y_{\mathrm{R2}}(k,k) \cdot (k+1) - Y_{\mathrm{R2}}(k,k) \cdot k
\end{aligned}$$

We will now consider the left-hand side (although later we will transform the right-hand side in a similar way). We will consider the operands of the subtraction separately by applying lemma C.2 to the first operand and lemma C.1 to the second to get:

$$\begin{aligned}
& Y_{\mathrm{R1}}(k,k)\cdot(k+1) - Y_{\mathrm{R1}}(k,k)\cdot k \\
&= (Y_{\mathrm{S1}}(k+1) + Y_{\mathrm{R1}}(k+1,k+1))\cdot(k+1) - \sum_{g=0}^{k} Y_{\mathrm{R1}}(g,k)\cdot g \\
&= \sum_{g=k+1}^{k+1} (Y_{\mathrm{S1}}(g) + Y_{\mathrm{R1}}(g,k+1))\cdot g - \sum_{g=0}^{k} Y_{\mathrm{R1}}(g,k)\cdot g
\end{aligned}$$

We next add the left-hand side of the assumption in equation C.3 to get:

$$\begin{aligned}
& \sum_{g=k+1}^{k+1} (Y_{\mathrm{S1}}(g) + Y_{\mathrm{R1}}(g,k+1))\cdot g - \sum_{g=0}^{k} Y_{\mathrm{R1}}(g,k)\cdot g \\
& \qquad + \sum_{g=0}^{k} (Y_{\mathrm{S1}}(g) + Y_{\mathrm{R1}}(g,k))\cdot g \\
&= \sum_{g=k+1}^{k+1} (Y_{\mathrm{S1}}(g) + Y_{\mathrm{R1}}(g,k+1))\cdot g + \sum_{g=0}^{k} Y_{\mathrm{S1}}(g)\cdot g
\end{aligned}$$

The addition of a term equal to zero allows us to simplify the expression:

$$\begin{aligned}
& \sum_{g=k+1}^{k+1} (Y_{\mathrm{S1}}(g) + Y_{\mathrm{R1}}(g,k+1))\cdot g + \sum_{g=0}^{k} Y_{\mathrm{S1}}(g)\cdot g + \sum_{g=0}^{k} Y_{\mathrm{R1}}(g,k+1)\cdot g \\
&= \sum_{g=0}^{k} (Y_{\mathrm{S1}}(g) + Y_{\mathrm{R1}}(g,k+1))\cdot g + \sum_{g=k+1}^{k+1} (Y_{\mathrm{S1}}(g) + Y_{\mathrm{R1}}(g,k+1))\cdot g \\
&= \sum_{g=0}^{k+1} (Y_{\mathrm{S1}}(g) + Y_{\mathrm{R1}}(g,k+1))\cdot g
\end{aligned}$$

Applying a similar process to the right-hand side allows us to say:

$$\sum_{g=0}^{k+1} (Y_{\mathrm{S1}}(g) + Y_{\mathrm{R1}}(g,k+1))\cdot g \leq \sum_{g=0}^{k+1} (Y_{\mathrm{S2}}(g) + Y_{\mathrm{R2}}(g,k+1))\cdot g$$

And so, when $k+1 = F$, we have a proof by induction for equation C.2. When the cost of failure is a fixed value equal to F, the final generation, then

$Y_{\mathrm{R1}}(g, F) = Y_{\mathrm{F1}}(g)\ \forall g$ because:

$$Y_{\mathrm{F1}}(g, F) = \begin{cases} 0 & \text{if} \quad g \neq F \\ 1 - \sum_{i=0}^{k} Y_{\mathrm{S1}}(i) & \text{if} \quad g = F \end{cases}$$

Further, $Y_{\mathrm{R2}}(g) = Y_{\mathrm{F2}}(g)\ \forall g$. Thus:

$$\sum_{g=0}^{F} (Y_{\mathrm{S1}}(g) + Y_{\mathrm{F1}}(g)) \cdot g \leq \sum_{g=0}^{F} (Y_{\mathrm{S2}}(g) + Y_{\mathrm{F2}}(g)) \cdot g$$

And so we have equation 4.4.

Appendix D

Electronic Appendix

The data-files analysed in this thesis are available for download from www.massey.ac.nz/~mgwalker/phd. These include the 15,193 direct evolution runs, the 85,000 manual incremental evolution runs and the 72,939 automatic evolution runs.

The data-files are in plain text with spaces between columns. The columns in all the summary.txt files are: run number, generations-to-termination, and whether the run succeeded (one) or failed (zero). There is one line for each run. Occasionally a run will have a "success" value of two; these lines should be removed as they indicate that the run did not complete (due to the experimental batch exceeding its allocated processing time).

Also available are the complete set of graphs used during the analysis of the data from the manual fitness-based incremental evolution experiments (chapter 8) plus sample computations for success proportion, minimum computational effort, and success effort.

Bibliography

Please note that the page numbers at the end of each reference are back-references into the body of this thesis. CiteSeer references can be viewed at citeseer.ist.psu.edu/ with the document identifier appended.

[1] Ricardo Aler, Daniel Borrajo, and Pedro Isasi. Evolving heuristics for planning. In D. Waagen, V. W. Porto, N. Saravanan, and A. E. Eiben, editors, *Evolutionary Programming VII, Seventh International Conference, EP98*, volume 1447 of *Lecture Notes in Computer Science*, pages 745–754, San Diego, CA, March 1998. Springer-Verlag. Available at scalab.uc3m.es/~dborrajo/articulos.html (accessed 15 Jan 2004). (p. 108)

[2] Ricardo Aler, Daniel Borrajo, and Pedro Isasi. Genetic programming and deductive-inductive learning: A multistrategy approach. In Jude Shavlik, editor, *Proceedings of the Fifteenth International Conference on Machine Learning, ICML'98*, pages 10–18, Madison, Wisconsin, USA, 1998. Morgan Kaufmann. Available at CiteSeer with document identifier '104987' (accessed 14 Jan 2004). (p. 108)

[3] Ricardo Aler, Daniel Borrajo, and Pedro Isasi. Genetic programming of control knowledge for planning. In *Artificial Intelligence Planning Systems*, pages 137–144, 1998. Available at CiteSeer with document identifier 'aler98genetic' (accessed 14 Jan 2004). (p. 108)

[4] David Andre and John R. Koza. Parallel genetic programming: A scalable implementation using the transputer network architecture. In Peter J. Angeline and Kenneth E. Kinnear, Jr, editors, *Advances in Genetic Programming*, volume 2. MIT Press, 1996. (p. 18)

[5] David Andre and John R. Koza. A parallel implementation of genetic programming that achieves super-linear performance. In Hamid R. Arabnia,

editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, volume III, pages 1163–1174, Sunnyvale, 9–11 August 1996. CSREA. (p. 18)

[6] David Andre and Astro Teller. Evolving team darwin united. In Minoru Asada and Hiroaki Kitano, editors, *RoboCup-98: Robot Soccer World Cup II*. Springer Verlag, 1999. Available at CiteSeer with document identifier 'andre99evolving' (accessed 14 Nov 2005). (pp. 103, 108, 113)

[7] P. J. Angeline and J. Pollack. Evolutionary module acquisition. In David B. Fogel and Wirt Atmar, editors, *Proceedings of the Second Annual Conference on Evolutionary Programming, La Jolla, California*, pages 154–163, 1993. Available at CiteSeer with document identifier 'angeline93evolutionary' (accessed 22 Mar 2002). (p. 2)

[8] Peter J. Angeline. An investigation into the sensitivity of genetic programming to the frequency of leaf selection during subtree crossover. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 21–29, Stanford University, CA, USA, 28–31 July 1996. MIT Press. (pp. 3, 10, 20, 21, 22, 27)

[9] M. Asada, S. Noda, S. Tawaratsumida, and K. Hosoda. Vision-based behavior acquisition for a shooting robot by using a reinforcement learning. In *Proceedings of IAPR IEEE Workshop on Visual Behaviors*, pages 112–118, 1994. Available at CiteSeer with document identifier 'asada94visionbased'. (p. 100)

[10] M. Asada, S. Noda, S. Tawaratsumida, and K. Hosoda. Vision-based reinforcement learning for purposive behaviour acquisition. In *IEEE International Conference on Robotics and Automation*, volume 1, pages 146–153, 1995. (p. 100)

[11] A. Asuncion and D. J. Newman. UCI machine learning repository. Available at mlearn.ics.uci.edu/MLRepository.html, 2007. University of California, Irvine, School of Information and Computer Sciences. (p. 185)

[12] Dhiraj Bajaj and Marcelo H. Ang, Jr. An incremental approach in evolving robot behaviour. In *Proceedings of the Sixth International Conference on Control, Automation, Robotics and Vision (ICARCV; Singapore)*, 2000. (p. 166)

[13] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming—An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, 1998. (pp. 2, 21, 25, 192)

[14] Gregory J. Barlow. Design of autonomous navigation controllers for unmanned aerial vehicles using multi-objective genetic programming. Master's thesis, North Carolina State University, March 2004. (pp. 101, 103, 109, 129, 166)

[15] Gregory J. Barlow. Personal communication. Email, October 4–11, 2005. (p. 109)

[16] Gregory J. Barlow, Choong K. Oh, and Edward Grant. Incremental evolution of autonomous controllers for unmanned aerial vehicles using multi-objective genetic programming. In *Proceedings of the IEEE Conference on Cybernetics and Intelligent Systems (CIS)*, pages 688–693, Singapore, December 2004. Peer reviewed version of [17]. (pp. 109, 166, 235)

[17] Gregory J. Barlow, Choong K. Oh, and Edward Grant. Incremental evolution of autonomous controllers for unmanned aerial vehicles using multi-objective genetic programming. In *Late Breaking Papers of the Genetic and Evolutionary Computation Conference (GECCO); Seattle*, 2004. Available on CD only. A precursor to [16]. (pp. 109, 166, 235)

[18] Brad Biggerstaff. Summary: Odds ratio and relative risk. S-News Mailing List Archives available at www.biostat.wustl.edu/archives/html/s-news/2001-05/msg00000.html, 2001. (p. 14)

[19] Rodney A. Brooks. Artificial life and real robots. In *Towards a Practice of Autonomous Systems: European Conference on Artificial Life*, pages 3–10. MIT Press, December 1991. Available at people.csail.mit.edu/brooks/. (p. 100)

[20] Anders L. Christensen and Marco Dorigo. Incremental evolution of robot controllers for a highly integrated task. In *From Animals to Animats 9*, volume 4095 of *Lecture Notes in Computer Science*, pages 473–484, 2006. (p. 104)

[21] Steffen Christensen. *Towards Scalable Genetic Programming.* PhD thesis, Ottawa-Carleton Institute for Computer Science, Ottawa, Canada, 2007. (pp. 14, 21, 22, 23, 25, 43, 45, 50, 105, 156, 220, 221, 222, 223)

[22] Steffen Christensen and Franz Oppacher. An analysis of Koza's computational effort statistic for genetic programming. In James A. Foster, Evelyne Lutton, Julian Miller, Conor Ryan, and Andrea G. B. Tettamanzi, editors, *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *LNCS*, pages 182–191, Kinsale, Ireland, 3–5 April 2002. Springer-Verlag. Their data is available for download from www.scs.carleton.ca/~schriste/ComputationalEffort. (pp. 18, 30, 76)

[23] Steffen Christensen and Franz Oppacher. The Y-test: Fairly comparing experimental setups with unequal effort. In Gary G. Yen, Lipo Wang, Piero Bonissone, and Simon M. Lucas, editors, *Proceedings of the 2006 IEEE Congress on Evolutionary Computation*, pages 1060–1065, Vancouver, 6-21 July 2006. IEEE Press. (p. 156)

[24] G. M. Clarke and D. Cooke. *A Basic Course in Statistics.* Arnold, 4th edition, 1998. (pp. 9, 12, 21, 22, 23, 27, 28, 56, 74, 144, 211, 212, 213, 215)

[25] Janet Clegg, James Alfred Walker, and Julian Francis Miller. A new crossover technique for cartesian genetic programming. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1580–1587, London, 7-11 July 2007. ACM Press. Available at www.cs.bham.ac.uk/~wbl/biblio/gecco2007/docs/p1580.pdf. (pp. 22, 23)

[26] D. Cliff, I. Harvey, and P. Husbands. Incremental evolution of neural network architectures for adaptive behaviour. Technical Report Serial No. CSRP 256, School of Cognitive and Computing Sciences, The University of Sussex, December 1992. (pp. 100, 104)

[27] Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences.* Lawrence Erlbaum Associates, 2nd edition, 1988. (p. 91)

[28] P. J. Darwen and X. Yao. On evolving robust strategies for iterated prisoner's dilemma. In Xin Yao, editor, *Progress in Evolutionary Computation*, volume 956 of *Lecture Notes in Computer Science*, pages 276–292. Springer-Verlag, Heidelberg, Germany, 1995. Available at CiteSeer with document identifier 'darwen95evolving' (accessed 18 Jan 2004). (p. 108)

[29] Edward de Bono. *How to have a Beautiful Mind.* Vermilion, 2004. (p. 107)

[30] Hugo de Garis. Genetic programming: building artificial nervous systems using genetically programmed neural network modules. In B. W. Porter and R. J. Mooney, editors, *Machine Learning: Proceedings of the Seventh International Conference*, pages 132–139. Morgan Kaufmann, 21-23 1990. Available at www.iss.whu.edu.cn/degaris/papers. (p. 100)

[31] Hugo de Garis. Multistrategy learning in neural nets: An incremental approach to genetic programming. In *Multistrategy Learning Workshop*, West Virginia, USA, May 1993. Available at www.iss.whu.edu.cn/degaris/papers. (p. 113)

[32] Kalyanmoy Deb, Amrit Pratab, Sameer Agrawal, and T. Meyarivan. A fast and elitist multi-objective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, April 2002. (p. 103)

[33] Marco Dorigo and Marco Colombetti. *Robot Shaping: An Experiment in Behaviour Engineering.* MIT Press, 1998. (p. 107)

[34] Jeffrey L. Elman. Incremental learning, or the importance of starting small. Technical Report CRL TR 9101, Center for Research in Language, University of California, March 1991. (p. 166)

[35] Roger I. Eriksson. An initial analysis of the ability of learning to maintain diversity during incremental evolution. In *Proceedings of the Workshop on Memetic Algorithms at the Genetic and Evolutionary Computation Conference (GECCO 2000)*, Las Vegas, Nevada, USA, 8–12 July 2000. Available at CiteSeer with document identifier 'eriksson00initial' and at www.cs.nott.ac.uk/~nxk/WOMA/DOCS/node7.html (accessed 22 July 2004). (pp. 103, 144)

[36] RoboCup Federation. Robocup official website. www.robocup.org, 2007. (p. 108)

[37] Gabriel J. Ferrer and Worthy N. Martin. Using genetic programming to evolve board evaluation functions for a boardgame. In *1995 IEEE Conference on Evolutionary Computation*, volume 2, pages 747–752, Perth, Australia, 29 November–1 December 1995. IEEE Press. Available at CiteSeer with document identifier 'ferrer95using' (accessed 9 Jan 2004). (p. 108)

[38] E. C. Fieller. Some problems in interval estimation. *Journal of the Royal Statistical Society. Series B (Methodological)*, 16(2):175–185, 1954. (p. 76)

[39] Gianluigi Folino and Giandomenico Spezzano. P-CAGE: an environment for evolutionary computation in peer-to-peer systems. In Pierre Collet, Marco Tomassini, Marc Ebner, Steven Gustafson, and Anikó Ekárt, editors, *Proceedings of the 9th European Conference on Genetic Programming (EuroGP)*, volume 3905 of *Lecture Notes in Computer Science*, pages 341–350, Budapest, Hungary, April 2006. Springer. (p. 192)

[40] Frank D. Francone, Peter Nordin, and Wolfgang Banzhaf. Benchmarking the generalization capabilities of a compiling genetic programming system using sparse data sets. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 72–80. MIT Press, 1996. Available at www.cs.mun.ca/~banzhaf/papers/benchmarking.pdf. (p. 192)

[41] Alex S. Fukunaga. Application of an incremental evolution technique to spacecraft design optimization. In *Proceedings of the IEEE International Conference on Evolutionary Computation (ICEC97)*, pages 431–435. IEEE Press, 13–16 April 1997. Available through *IEEE Xplore* (accessed 1 Mar 2004). (pp. 104, 113)

[42] Alex S. Fukunaga and Andrew B. Kahng. Improving the performance of evolutionary optimization by dynamically scaling the evolution function. In *1995 IEEE Conference on Evolutionary Computation*, volume 1, pages 182–187, Perth, Australia, 29 November–1 December 1995. IEEE Press. Available at CiteSeer with document identifier 'fukunaga95improving' (accessed 24 Feb 2004). (pp. 102, 103, 104, 110, 166)

[43] J. Hallam G. N. Yannakakis and J. Levine. Evolutionary computation variants for cooperative spatial coordination. In *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, pages 2715–2722. IEEE Press, 2005. (pp. 13, 24)

[44] Chris Gathercole. *An Investigation of Supervised Learning in Genetic Programming.* PhD thesis, The Univeristy of Edinburgh, 1998. (pp. 2, 155, 166, 170)

[45] S. Geman, E. Bienenstock, and R. Doursat. Neural networks and the bias/variance dilemma. *Neural Computation*, 4:1–58, 1991. Available at www.dam.brown.edu/people/geman. (p. 193)

[46] J. T. Gene Hwang. Fieller's problems and resampling techniques. *Statistica Sinica*, 5:191–171, 1995. (pp. 73, 78)

[47] William E. Glaholt and Du Zhang. GP-Lab: The genetic programming laboratory. In *Proceedings of the 16th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 388–395. IEEE Press, 2004. (p. 192)

[48] William Edward Glaholt. GP-Lab: The genetic programming laboratory. Master's thesis, Computer Science, California State University, Sacramento, 2004. (p. 192)

[49] Harvey Goldstein and Michael J. R. Healy. The graphical presentation of a collection of means. *Journal of the Royal Statistical Society. Series A (Statistics in Society)*, 158(1):175–177, 1995. (p. 10)

[50] Faustino Gomez and Risto Miikkulainen. Incremental evolution of complex general behavior. Technical Report AI96-248, The University of Texas at Austin, USA, June 1996. Available at CiteSeer with document identifier 'gomez96incremental' and www.cs.utexas.edu (accessed 15 Sept 2004). (pp. 111, 166)

[51] Faustino Gomez and Risto Miikkulainen. Incremental evolution of complex general behavior. *Adaptive Behaviour*, 5(3/4):317–342, 1997. Available at CiteSeer with document identifier 'gomez97incremental' (accessed 24 July 2004). (pp. 111, 166)

[52] Faustino Gomez and Risto Miikkulainen. Solving non-markovian control tasks with neuroevolution. In *Proceedings of the International Joint Conference on Artificial Intellignce (IJCAI-99, Stockholm, Sweden)*. Morgan Kaufmann, 1999. (pp. 107, 113, 166, 169)

[53] Faustino John Gomez. *Robust Non-linear Control through Neuroevolution*. PhD thesis, Artificial Intelligence Laboratory, The University of Texas at Austin, USA, August 2003. Report AI-TR-03-303. (pp. 99, 104, 107, 110, 111, 113, 166, 169, 170)

[54] Diana F. Gordon and Devika Subramanian. A multistrategy learning scheme for assimilating advice in embedded agents. In *Proceedings of the Second International Workshop on Multistrategy Learning*, pages 218–233, 1993. Available at CiteSeer with document identifier '99783' and at www.aic.nrl.navy.mil/papers/1993/ml.html (assessed 4 Jan 2004). (p. 108)

[55] Axel Großmann. *Continual learning for mobile robots.* PhD thesis, School of Computer Science, The University of Birmingham, Birmingham, UK, February 2001. (p. 100)

[56] Steven Matt Gustafson. Layered learning in genetic programming for a cooperative robot soccer problem. Master's thesis, Department of Computing and Information Science, College of Engineering, Kansas State Univeristy, Kansas, USA, December 2000. Available at CiteSeer with document identifier '450396' and at www.cs.nott.ac.uk/~smg/ (accessed 3 Nov 2005). (pp. 112, 130)

[57] Scott Harmon, Edwin Rodríguez, Christopher Zhong, and William Hsu. A comparison of hybrid incremental reuse strategies for reinforcement learning in genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, 2004. Poster Abstract. (p. 100)

[58] Inman Harvey. *The Artificial Evolution of Adaptive Behaviour.* PhD thesis, University of Sussex, 1995. Available from www.cogs.susx.ac.uk/users/inmanh/. (pp. 101, 108, 109, 166)

[59] Ernest R. Hilgard and Gordon H. Bower. *Theories of Learning.* Prentice-Hall, 1975. (pp. 100, 107)

[60] John H. Holland. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, 1975. (p. 2)

[61] William H. Hsu and Steven M. Gustafson. Genetic programming and multi-agent layered learning by reinforcements. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 764–771, New York, 9–13 July 2002. Morgan Kaufmann. (pp. 100, 101, 107, 112, 130)

[62] William H. Hsu, Scott J. Harmon, Edwin Rodríguez, and Christopher Zhong. Empirical comparison of incremental reuse strategies for genetic programming-based keep-away soccer agents. In *AAAI Fall Symposium on Artificial Multiagent Learning*, 2004. (p. 107)

[63] William H. Hsu, Scott J. Harmon, Edwin Rodríguez, and Christopher Zhong. Empirical comparison of incremental reuse strategies in genetic programming for keep-away soccer. In *Late Breaking Papers of the Genetic and Evolutionary Computation Conference (GECCO); Seattle*, 2004. (pp. 100, 106, 107, 112)

[64] Guoyong Hwang, Jianrong Wu, and G. Rhys Williams. Fieller's interval and the bootstrap-fieller interval for the incremental cost-effectiveness ratio. *Health Services & Outcomes Research Methodology*, 1(3–4):291–303, 2000. (pp. 73, 78)

[65] David Jackson. Hierarchical genetic programming based on test input subsets. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2007)*, 2007. To appear. (pp. 130, 133)

[66] David Jackson. Partitioned incremental evolution of hardware using genetic programming. In *Proceedings of the Congress on Evolutionary Computation (CEC-2007)*, 2007. To appear. (pp. 130, 133, 136)

[67] David Jackson and Adrian P. Gibbons. Layered learning in boolean GP problems. In Ebner et. al, editor, *Genetic Programming. Proceedings of the 10th European Conference, EuroGP 2007*, volume 4445 of *Lecture Notes in Computer Science*, pages 148–159, 2007. (pp. 4, 104, 116, 129, 130, 131, 136, 139)

[68] Tatiana Kalganova. Bidirectional incremental evolution in extrinsic evolvable hardware. In Jason Lohn, Adrian Stoica, and Didier Keymeulen, editors, *Proceedings of the Second NASA/DoD Workshop on Evolvable Hardware (EH2000)*, pages 65–74, Palo Alto, California, USA, 13–15 July 2000. IEEE Press. Available at CiteSeer with document identifier 'kalganova00bidirectional' and at www.brunel.ac.uk/~eestttk/ (accessed 24 Feb 2004). (p. 102)

[69] M. Keijzer, V. Babovic, C. Ryan, M. O'Neill, and M. Cattolico. Adaptive logic programming. In Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 42–49, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann. (pp. 20, 29, 37, 91)

[70] DaeEun Kim. Memory analysis and significance test for agent behaviours. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation (GECCO)*, volume 1, pages 151–158. ACM Press, 8-12 July 2006. Available at www.cs.bham.ac.uk/~wbl/biblio/gecco2006/docs/p151.pdf. (pp. 13, 24)

[71] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992. (pp. 2, 9, 11, 17, 19, 21, 24, 27, 30, 45, 50, 76, 77, 81, 110, 115, 116, 135, 160, 175, 190, 191, 219, 221)

[72] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, 1994. (pp. 2, 3, 11, 17, 30, 76, 115, 116, 125, 140, 154, 157, 190)

[73] John R. Koza, David Andre, Forrest H. Bennett III, and Martin Keane. *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufmann, 1999. (pp. 2, 17, 18, 102, 192)

[74] John R. Koza, Martin A. Keane, Matthew J. Streeter, William Mydlowec, Jessen Yu, and Guido Lanza. *Genetic Programming IV: Routine Human-Competitive Machine Learning*. Kluwer Academic Publishers, 2003. (pp. 1, 17)

[75] Michael D. Kramer and Du Zhang. GAPS: A genetic programming system. In *Proceedings of the 24th International Conference on Computer Software and Applications*, pages 614–619, 2000. (p. 192)

[76] G. Saravana Kumar, P.K. Kalra, and Sanjay G. Dhande. Parameter optimization for B-spline curve fitting using genetic algorithms. In Sarker et al.[101], pages 1871–1878. (p. 108)

[77] W. B. Langdon and J. P. Nordin. Seeding GP populations. In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin, and Terence C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *Lecture Notes in Computer Science*, pages 304–315. Springer-Verlag, 15–16 April 2000. (p. 108)

[78] Geum Yong Lee. Genetic recursive regression for modeling and forecasting real-world chaotic time series. In *Advances in Genetic Programming*, volume 3, chapter 17, pages 401–423. MIT Press, 1999. Available at www.cs.bham.ac.uk/~ewbl/aigp3/ch17.pdf. (p. 193)

[79] Peter M. Lee. *Bayesian Statistics: An Introduction.* Arnold, 2nd edition, 1997. (pp. 13, 74)

[80] Wei-Po Lee. *Evolving Robots: From Simple Behaviours to Complete Systems.* PhD thesis, University of Edinburgh, 1997. (p. 24)

[81] Wei-Po Lee, John Hallam, and Henrik Hautop Lund. Applying genetic programming to evolve behavior primitives and arbitrators for mobile robots. In *Proceedings of IEEE 4th International Conference on Evolutionary Computation*, volume 1. IEEE Press, 1997. (p. 166)

[82] Richard Lowry. VassarStats: Web site for statistical computation. Available at http://faculty.vassar.edu/lowry/VassarStats.html, 2007. Specifically see http://faculty.vassar.edu/lowry/prop2_ind.html for a calculator that implements the Wilson-score methods described in Newcombe's study [90]. (p. 15)

[83] Sean Luke. When short runs beat long runs. In Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 74–80. Morgan Kaufmann, 7–11 July 2001. (pp. 22, 43, 94, 166)

[84] Sean Luke, Charles Hohn, Jonathan Farris, Gary Jackson, and James Hendler. Co-evolving soccer softbot team coordination with genetic programming. In H. Kitano, editor, *RoboCup-97: Robot Soccer World Cup*, pages 398–411. Springer, 1998. (p. 113)

[85] Sean Luke and Liviu Panait. Is the perfect the enemy of the good? In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 820–828, New York, 9-13 July 2002. Morgan Kaufmann Publishers. (pp. 9, 12, 17, 19, 20, 22, 71, 209)

[86] Olli Miettinen and Markku Nurminen. Comparative analysis of two rates. *Statistics in Medicine*, 4:213–226, 1985. (p. 14)

[87] Julian F. Miller and Peter Thomson. Cartesian genetic programming. In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin, and Terence C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 121–132, Edinburgh, 15-16 April 2000. Springer-Verlag. (pp. 9, 19, 24, 211)

[88] Jean-Baptiste Mouret, Stephane Doncieux, and Jean-Arcady Meyer. Evolution of target-following neuro-controllers for flapping-wing animats. In S. Nolfi, G. Baldassare, R. Calabretta, J.C. Hallamand, D. Marocco, J.-A. Meyer, O. Miglino, and D. Parisi, editors, *From Animals to Animats: Proceedings of the 9th International Conference on the Simulation of Adaptive Behavior (SAB)*, pages 606–618, 2006. (p. 169)

[89] Takayoshi Naemura, Tomonori Hashiyama, and Shigeru Okuma. Module generation for genetic programming and its incremental evolution. In *Second Asia-Pacific Conference on Simulated Evolution and Learning*, Australian Defence Force Academy, Canberra, Australia, 1998. (pp. 116, 135, 166)

[90] Robert G. Newcombe. Interval estimation for the difference between independent proportions: comparison of eleven methods. *Statistics in Medicine*, 17:873–890, 1998. (pp. 14, 15, 243)

[91] Robert G. Newcombe. Two-sided confidence intervals for the single proportion: comparison of seven methods. *Statistics in Medicine*, 17:857–872, 1998. (pp. 12, 28, 49, 55)

[92] Jens Niehaus and Wolfgang Banzhaf. More on computational effort statistics for genetic programming. In Conor Ryan, Terence Soule, Maarten Keijzer, Edward Tsang, Riccardo Poli, and Ernesto Costa, editors, *Genetic Programming, Proceedings of EuroGP'2003*, volume 2610 of *LNCS*, pages 164–172, Essex, 14-16 April 2003. Springer-Verlag. (pp. 19, 20, 30, 211)

[93] Stephen Oman and Padraig Cunningham. Using case retrieval to seed genetic algorithms. *International Journal of Computational Intelligence and Applications*, 1(1):71–82, 2001. Available at CiteSeer with document identifier 'oman97using'. (p. 108)

[94] Simon Perkins. *Incremental Acquisition of Complex Visual Behaviour using Genetic Programming and Robot Shaping.* PhD thesis, University of Edinburgh, 1998. (p. 107)

[95] John E. Perry. The effect of population enrichment in genetic programming. In *Proceedings of the First IEEE World Congress on Computational Intelligence*, pages 456–461. IEEE Press, 27–29 June 1994. (p. 108)

[96] Riccardo Poli, William B. Langdon, Nicholas F. McPhee, and John R. Koza. Genetic programming: An introductory tutorial and a survey of techniques and applications. Technical report, Department of Computing and Electronic Systems, University of Essex, UK, October 2007. Available at www.essex.ac.uk/dces/research/publications/technicalreports/ces475.pdf. (p. 2)

[97] Riccardo Poli and Jonathan Page. Solving even-12, -13, -15, -17, -20 and -22 boolean parity problems using sub-machine code gp with smooth uniform crossover, smooth point mutation and demes. Technical Report CSRP-99-2, Centrum voor Wiksunde en Informatica, Amsterdam, January 1999. (pp. 115, 184)

[98] Mohammad Adil Qureshi. *The Evolution of Agents*. PhD thesis, Department of Computer Science, University College London, 2001. (pp. 155, 165)

[99] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2006. ISBN 3-900051-07-0. (p. 44)

[100] N. Saravanan and David B. Fogel. Evolving neural control systems. *IEEE Expert*, pages 23–27, June 1995. (pp. 107, 169)

[101] R. Sarker et al., editor. *Proceedings of the 2003 Congress on Evolutionary Computation*, Canberra, Australia, 8–12 December 2003. IEEE Press. (pp. 242, 246)

[102] Jürgen Schmidhuber. Optimal ordered problem solver. Technical Report TR IDSIA-12-02, version 1.0, Dalle Molle Institute for Artificial Intelligence (IDSIA), Switzerland, July 2002. Available at www.idsia.ch/~juergen/oops.html. (p. 105)

[103] Alan C. Schultz and John J. Grefenstette. Improving tactical plans with genetic algorithms. In *Proceedings of IEEE Conferenecе on Tools for Artificial Intelligence*, pages 328–334, 1990. Available at CiteSeer with document identifier 'schultz90improving' and at cs.gmu.edu/research/gag/pubs.html (accessed 4 Jan 2004). (p. 108)

[104] Y. Shichel, E. Ziserman, and M. Sipper. GP-Robocode: Using genetic programming to evolve robocode players. In *Proceedings of 8th European Conference on Genetic Programming (EuroGP)*, volume 3447 of *Lecture Notes in Computer Science*, pages 143–154. Springer-Verlag, 2005. Available at www.cs.bgu.ac.il/~sipper/papabs/eurogprobo-final.pdf. (p. 193)

[105] Mark Stephenson, Una-May O'Reilly, Martin C. Martin, and Saman Amarasinghe. Genetic programming applied to compiler heuristic optimization. In *Proceedings of the European Conference on Genetic Programming (EuroGP)*, volume 2610 of *Lecture Notes in Computer Science*, pages 245–257, 2003. (p. 155)

[106] Peter Stone. *Layered Learning in Multi-Agent Systems.* PhD thesis, School of Computer Science, Carnegie Mellon University, PA, USA, 1998. (pp. 2, 4, 100, 105, 106, 112, 113)

[107] Peter Stone and Manuela Veloso. A layered approach to learning client behaviours in robocup soccer server. *Applied Artificial Intelligence*, 12:165–188, 1998. Available at CiteSeer with document identifier 'stone98layered' and at www.cs.utexas.edu/~pstone (accessed 15 Nov 2005). (pp. 4, 105)

[108] René Thomsen, Gary B. Fogel, and Thiemo Krink. A clustal alignment improver using evolutionary algorithms. In *Proceedings of the Fourth Congress on Evolutionary Computation (CEC-2002)*, volume 1, pages 121–126, 2002. Available at CiteSeer with document identifier 'thomsen02clustal' and at www.evalife.dk (accessed 3 Jan 2004). (p. 108)

[109] René Thomsen, Gary B. Fogel, and Thiemo Krink. Improvement of clustal-derived sequence alignments with evolutionary algorithms. In Sarker et al.[101], pages 312–319. (p. 108)

[110] J. I. van Hemert. Applying adaptive evolutionary algorithms to hard problems. Master's thesis, Department of Computer Science, Leiden University, The Netherlands, 1998. (p. 23)

[111] Ulrike von Luxburg and Volker H. Franz. Confidence sets for ratios: A purely geometric approach to Fieller's theorem. Technical Report TR-133, Max Planck Institute for Biological Cybernetics, Tübingen, Germany, December 2004. (p. 76)

[112] Matthew Walker. Comparing the performance of incremental evolution to direct evolution. In *Second International Conference on Autonomous Robots and Agents (ICARA)*, pages 119–124, December 2004. (pp. 101, 104, 165, 166)

[113] Matthew Walker. Mutated seeds: A performance enhancer for incremental evolution. In *Proceedings of the Institute of Information and Mathematical Sciences (IIMS) Postgraduate Conference*, pages 95–99, 2004. Available at www.massey.ac.nz/~iimspg/2004conference/proceedings/. (pp. 101, 102, 170)

[114] Matthew Walker, Howard Edwards, and Chris Messom. Confidence intervals for computational effort comparisons. In Ebner et. al, editor, *Genetic Programming. Proceedings of the 10th European Conference, EuroGP 2007*, volume 4445 of *Lecture Notes in Computer Science*, 2007. (p. viii)

[115] Matthew Walker, Howard Edwards, and Chris Messom. The reliability of confidence intervals for computational effort comparisons. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation (GECCO)*, volume 2, pages 1716–1723, London, 7-11 July 2007. (p. viii)

[116] Matthew Walker, Howard Edwards, and Chris Messom. Success effort and other statistics for performance comparisons in genetic programming. In *Congress on Evolutionary Computing (CEC)*. IEEE Press, 2007. Available at www.massey.ac.nz/~chmessom/research.html. (p. viii)

[117] Matthew Walker, Howard Edwards, and Chris Messom. “success effort” for performance comparisons. In *GECCO ’07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1760–1760, London, 7–11 July 2007. Available at www.cs.bham.ac.uk/~wbl/biblio/gecco2007/docs/p1760.pdf. (p. viii)

[118] C. Henrik Westerberg and John Levine. Investigation of different seeding strategies in a genetic planner. In Egbert J. W. Boers, Stefano Cagnoni, Jens Gottlieb, Emma Hart, Pier Luca Lanzi, Günther Raidl, Robert E. Smith, and Harald Tijink, editors, *Applications of Evolutionary Computing; Proceedings of EvoWorkshops2001*, volume 2037, pages 505–514. Springer-Verlag, 18-19 2001. Available at CiteSeer with document identifier ‘westerberg01investigation’. (p. 108)

[119] Shimon Whiteson, Nate Kohl, Risto Miikkulainen, and Peter Stone. Evolving soccer keepaway players through task decomposition. *Machine Learning*, 59:5–30, 2005. (pp. 102, 104, 106, 169)

[120] Shimon Whiteson and Peter Stone. Concurrent layered learning. In *AAMAS 2003: Proceedings of the Second Internation Joint Conference on Autonomous Agents and Multi-Agent Systems*, pages 193–200, 2003. Available at CiteSeer with document identifier 'whiteson03concurrent' (accessed 21 Nov 2005). (pp. 100, 102, 107)

[121] D. Whitley, K. Mathias, and P. Fitzhorn. Delta coding: An iterative search strategy for genetic algorithms,. In Rick Belew and Lashon Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 77–84. Morgan Kaufmann, 1991. Available at CiteSeer with document identifier 'whitley91delta' (accessed 20 Feb 2004). (pp. 100, 170)

[122] Alexis P. Wieland. Evolving neural network controllers for unstable systems. In *Proceedings of the International Joint Conference on Neural Networks*, volume 2, pages 667–673, Seattle, WA, USA, 1991. IEEE Press. (pp. 99, 100, 107, 169)

[123] Edwin B. Wilson. Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association*, 22(158):209–212, 1927. (p. 12)

[124] Jay F. Winkeler and B. S. Manjunath. Incremental evolution in genetic programming. In *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 403–411. Morgan Kaufmann, 1998. Available at CiteSeer with document identifier 'winkeler98incremental' (accessed 29 Feb 2004). (pp. 102, 110)