TOWARDS SCALABLE GENETIC PROGRAMMING

by

Steffen Christensen, B.C.S

A thesis submitted to

the Faculty of Graduate Studies and Research

in partial fulfillment of

the requirements of the degree of

Doctor of Philosophy

Ottawa-Carleton Institute for Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario

November 14, 2006

Your file Votre référence
ISBN: 978-0-494-23290-3
Our file Notre référence
ISBN: 978-0-494-23290-3

Abstract

Genetic programming (GP) is a technique for automatically solving optimization problems where candidate solutions are expressible as trees with no human intervention. We propose an extension of GP, termed scalable genetic programming, which solves problems parameterized by a scalable difficulty parameter. We first define a taxonomy of evolutionary computation (EC) systems that identifies variability dimensions and levels for EC systems. We define an algorithm, the scientist algorithm, which uses genetic programming as a subroutine to reliably make progress on scalable problems. The scientist algorithm uses a toolkit of provided routines to progress, by carrying out experiments to determine the value of different methods. We define several of the tools for this toolkit. We define and implement an algorithm for systematically considering all small trees for a problem. We then use these small trees in an iterative algorithm to define subroutines that improve performance on a problem under study. Using this algorithm, we beat the best known performance on the artificial ant on the Santa Fe trail problem by a factor of 7.

As science depends on accurate hypothesis testing to make progress, we perform a comparison and evaluation of statistical techniques used to evaluate evolutionary computation systems. Finding many of these wanting, with the exception of computational effort, we introduce two additional techniques, effective mean best fitness and the *y*-test. We also perform an extensive analysis of the computational effort, and identify some statistical cautions around the use of this key statistic. We provide an algorithm that carefully uses computational effort to determine the best values of population size and generation number for an EC treatment.

Finally, we identify several components that are of use with the scientist algorithm. We treat the use of multiobjective algorithms in GP, principal components analysis, and their combination. We demonstrate this by providing and testing an algorithm that makes evolved trees parsimonious. We introduce the notion of incremental evolution, and use it to make useful subroutines automatically from successful solutions to easy problems. We then use this to demonstrate scalable genetic programming on an integer sorting problem.

Acknowledgements

Firstly, I would like to thank my advisor, Franz Oppacher, for his considerable support and frequent advice in developing this thesis. He is one of the most intelligent people I know, and it shows in his advocacy and advice. Discussions with the members of the Evolutionary Computation and Artificial Life group at Carleton University have also been invaluable, including Franz, Jackie, Mark Wineberg, Hassan Masum, Rob Cattrall, George Carmody, and Lee Graham. They have proved an invaluable resource and I have learned much from the many technical discussions we have had there.

There have been other professors and researchers who have helped me in discussions critical for this thesis. I would like to thank John Oommen for his numerical and mathematical approach to automatic computing problems. The statistical part of this thesis has benefited greatly from discussions with Mark Wineberg, Sean Luke, Pat Farrell, and Ann Woodside. Additional thanks go to Sean for implementing his GP system, ECJ, and for making it available to the broader community. I would like to thank Bill Langdon for his pioneering work on the Santa Fe trail problem, for his experimental approach to EC, and for his excellent tutorial at GECCO on the topic of GP theory.

I would like to thank those who helped this thesis through their social and financial support. The National Sciences and Engineering Research Council of Canada contributed to this work. I would like to thank Franz Oppacher, the School of Computer Science at Carleton, Sean McGuire, and my parents for their support. I would like to offer a special thanks to the Bridgehead coffee shops in Ottawa for providing workspace, free internet access, excellent coffee, and fantastic staff. Special thanks go out to Sahana, Devina, Aimee, Emma, Laine, Liz, Katie, Tyler, Rebecca, and all the rest. I would like to thank Lisa Jayne, Katie Desormeaux, and Franz for proofreading the copy and making many comments and suggestions. Finally, many thanks go to Lisa Jayne for all her help, encouragement and support.

Table of Contents

List of Figures

List of Algorithms

Chapter 1: Introduction

This dissertation consists of two major branches. The first branch consists of methods and statistics for evaluating and comparing stochastic search algorithms fairly, making up Chapters 3 and 4. The second branch introduces the idea of scalable genetic programming. It presents a number of computational tools which will be useful in achieving this purpose, and validates them through experiment. An introduction to scalable genetic programming is given in Chapter 2, while the major contributions are presented in Chapter 5, on small trees; and Chapter 6, on modelling and progressive algorithms. A set of tools are then presented and tested independently and together in Chapter 7, which also demonstrates scalable genetic programming for a simple problem. Finally, Chapter 8 synthesizes the results of this work.

There are 8 chapters in this document, plus one appendix. Chapter 1 offers a short introduction to the components of this thesis. A conventional introduction to genetic programming (GP), a discussion of some of the strengths of GP, and a brief summary of a common GP test problem round out the chapter.

Chapter 2 begins with a discussion of some of the limits on genetic programming performance. We then situate genetic programming in the field of evolutionary computation techniques, which are characterized by a few variability dimensions such as data representation, input complexity, outcome variables, environmental complexity and representation abstractness. This gives rise to a taxonomy of evolutionary computation problems, in which scalable genetic programming presents a new level of complexity for EC to solve. We note the differences between scalability in the context of EC and scability as used in computer science generally. We then discuss some properties that an effective automatic programming algorithm should have if it is to make headway without human intervention. This argument ties in to the remainder of the thesis, where we implement and test several of these elements. We introduce the scientist algorithm here, and contrast it with the robot scientist approach to automating science [King 2001]. We then close the chapter with a discussion of three useful techniques for the scientist algorithm: using intermediate task progress, fitness records, and adaptive choice of test cases.

Chapter 3 is the first of the statistical chapters. In EC research, a reliable way of determining whether a treatment is beneficial is required. Hypothesis testing and statistics are the conventional tools used for this purpose in the scientific literature. A problem immediately presents itself: the field does not have a single uniform measure of determining what technique is better. Instead, a mix of five measures is in common use. Accordingly, we evaluate each of these statistics against one another on a common problem: the artificial ant on the Santa Fe trail. Chapter 3 shows that the best statistic to use when the problem at hand has a finite and achievable probability of success is the effective success probability. We also show that this is reciprocally related to the computational effort statistic introduced by Koza [Koza 2001]. Chapter 3 argues that for problems without natural success criteria, the effective mean best fitness is the best test to use. Some discussion is made of discriminating between these two cases and why a single measure for comparing EC methods won't work. It also treats the suitability of the other tests that are presently used in the field, and dismisses them as lacking.

Chapter 4 continues the statistical bent of Chapter 3. It begins with a detailed investigation of the preferred statistic for comparisons between EC methods: the computational effort. This statistic, as normally defined, has a few problems associated with it. We fix the definition so it is more statistically relevant, and then analyze it. This leads to a bit of a quandary. For small run counts, computational effort is badly broken. We introduce another way of looking at the data, which gives rise to a novel statistical test: the *y*-test. We analyze the *y*-test as to its suitability for doing hypothesis testing.

Chapter 5 adapts the No Free Lunch argument, a largely theoretical attack on universal search algorithms, to the issues of genetic programming. We define an enumeration algorithm for trees, and discuss how the No Free Lunch theorem applies to trees. We confirm Langdon's result that the test problem named "artificial ant on the Santa Fe trail" is GP-hard; that is, it is almost easier for random search to succeed on this problem than for genetic programming. We then use the tree enumeration algorithm defined here to automatically define subroutines from small trees that look promising. These subroutines dramatically reduce the computational effort required to solve the Santa Fe trail problem. For instance, we beat the best known performance on the Santa Fe trail problem by a factor of seven, and beat the typical performance by a factor of 20. This subroutine-generation algorithm can fit nicely into the scientist algorithm as a subroutine.

Chapter 6 introduces the idea of a model of success probability as a function of population size *M* and generation number *G*. Such a model would have great utility, as we could use it to choose parameter optimally. We demonstrate this notion by showing the results of many experiments on the artificial ant on the Santa Fe trail, and prepare a qualitative model of success performance for it. We then present three algorithms; the first is intended to solve a problem of unknown difficulty when nothing is known. The second is an algorithm specialized to the Santa Fe trail performance model that performs within a factor of four of the optimal parameter settings. The third automatically finds optimal values of M and G for a provided benchmark problem.

Chapter 7 presents some technologies that can be used with the scientist algorithm to improve genetic programming. We begin by considering the significant value of using multiobjective optimization in genetic programming. This leads naturally into a discussion of principal components analysis to mitigate the poor performance of multiobjective methods on problems of high dimensionality. We can also use multiobjective techniques to solve main and secondary tasks. We illustrate this usage by the algorithm EVOLVE-TINY-TREES that automatically optimizes a genetic programming tree for parsimony. We then discuss scalable difficulty and incremental evolution. We give an algorithm for making subroutines automatically from successful solutions to simpler problems. This algorithm, and incremental evolution specifically, are developed and tested for utility on a simple integer sorting problem. We conclude this chapter with an example of solving the title problem of this thesis: scalable genetic programming.

Chapter 8 is the conclusion of the thesis. In this chapter, we provide a synthesis of the ideas and theories presented herein. We outline how the advances of this dissertation contribute to the field. We draw together the recommendations and advice strewn throughout this work into one convenient location as a resource for GP practitioners. We conclude with a discussion of future work.

Finally, we would like to introduce the Appendix. In Appendix 1 of this thesis, we take the reader through an example of the sequential trial-and-error task that a genetic programming system encounters. This parallels Searle's "Chinese Room" argument

[Searle 1980, Searle 1984] on artificial intelligence. The Appendix uses an allegory to demonstrate how a genetic programming system would face a typical GP challenge. We end the Appendix with a brief discussion of the techniques used during problem-solving and how they might be implemented in genetic programming. While this allegory may seem philosophical in nature, we strongly recommend that readers not familiar with the workings of genetic programming read the Appendix before continuing. It is intended to be self-contained, and can be read out-of-order. The reader may then return to the more conventional description of the genetic programming algorithm of the next section.

Strengths of Genetic Programming

Genetic Programming (GP) is a subdiscipline of artificial intelligence that tackles the problem of program induction. It is essentially an application of Holland's genetic algorithm [Holland 1975] to the problem of searching for a solution in the set of all programs, defined by a given grammar, up to some maximum size. The variant of Genetic Programming in most common use in the literature remains that initially proposed by John Koza [Koza 1992], although there is much diversity. The GP algorithm, explained in detail in [Koza 1992], has been shown to be successful in inducing solutions to a wide variety of problems for which other program induction methods had held little success.

A common failing of more "classical" AI techniques of the day is that the results obtained were shown to be successful only on the test problems under consideration, or involved adding a great deal of ad-hoc "human smarts" in the programming of the AI system under consideration. Two classical examples are the automated chess and checkers programs, Deep Blue [IBM 2006] and Chinook [Schaeffer 1996]. These are characterized by extensive off-line research on the problems in question with subsequent fine-tuning of minimax board evaluation parameters, vast opening books and endgame tables, and powerful computers doing deep searches of possible move-countermove chains. Very often, the systems explored were not made available for other researchers to test, so reproducibility was suspect. Another bugaboo of early research was that experiments were deemed successful or unsuccessful on the results of a single run.

In contrast, Koza introduced a new artificial intelligence technique, genetic programming, which differed from its classical AI counterparts in several important ways. Genetic programming emulates biological evolution, by beginning with very simple operations and a very simple algorithm, the genetic algorithm [Holland 1975]. This approach to problem-solving greatly enhanced the automatizability of results obtained, in that very little programming or heuristics were required to get the desired performance. Furthermore, Koza was one of the early adopters of open source, publishing his source code verbatim in [Koza 1992d] so that other researchers could replicate his results. He also provides full and complete information about minor parameters that were used in his runs, defusing a critical problem that often precluded the replication of "traditional" results in AI. He performed several runs of each presented problem, and conducted some statistics in an attempt to determine and compare the computational effort of solving a given problem.

Since genetic programming requires only a fitness function and some programming primitives to work, it has been hailed by its supporters as an automatic artificial intelligence system. Koza applied his genetic programming system as widely as he was able, describing no less than 14 different application domains in [Koza 1992], and many more in [Koza 1994], [Koza 1999] and [Koza 2004].

Some of the more interesting problems are summarized in [Koza 2004], where Koza discusses how his team currently approaches human competitive intelligence for solving design problems in electric engineering and antenna design. A recent list of human-competitive results is available at [Koza 2006]; this list, current as of this writing is provided in Fig. 1.

Figure 1

#	*Claimed instance*	*Basis for claim of human-competitiveness*	*Reference*
1	Creation of a better-than-classical quantum algorithm for the Deutsch-Jozsa "early promise" problem	of scientific merit, beats achievement in field	Spector, Barnum, and Bernstein 1998
2	Creation of a better-than-classical quantum algorithm for Grover's database search problem	of scientific merit, beats achievement in field	Spector, Barnum, and Bernstein 1999
3	Creation of a quantum algorithm for the depth-two AND/OR query problem that is better than any previously published result	new scientific merit	Spector, Barnum, Bernstein, and Swamy 1999; Barnum, Bernstein, and Spector 2000
4	Creation of a quantum algorithm for the depth-one OR query problem that is better than any previously published result	new scientific merit	Barnum, Bernstein, and Spector 2000
5	Creation of a protocol for communicating information through a quantum gate that was previously thought not to permit such communication	new scientific merit	Spector and Bernstein 2003
6	Creation of a novel variant of quantum dense coding	new scientific merit	Spector and Bernstein 2003
7	Creation of a soccer-playing program that won its first two games in the Robo Cup 1997 competition	wins in competition	Luke 1998
8	Creation of a soccer-playing program that ranked in the middle of the field of 34 human-written programs in the Robo Cup 1998 competition	wins in competition	Andre and Teller 1999
9	Creation of four different algorithms for the transmembrane segment identification problem for proteins	of scientific merit, beats human best	Sections 18.8 and 18.10 of Genetic Programming II and sections 16.5 and 17.2 of Genetic Programming III
10	Creation of a sorting network for seven items using only 16 steps	patentable, new scientific merit	Sections 21.4.4, 23.6, and 57.8.1 of Genetic Programming III
11	Rediscovery of the Campbell ladder topology for lowpass and highpass filters	patentable, beats achievement in field	Section 25.15.1 of Genetic Programming III and section 5.2 of Genetic Programming IV
12	Rediscovery of the Zobel "M-derived half section" and "constant K" filter sections	patentable, beats achievement in field	Section 25.15.2 of Genetic Programming III
13	Rediscovery of the Cauer (elliptic) topology for filters	patentable, beats achievement in field	Section 27.3.7 of Genetic Programming III
14	Automatic decomposition of the problem of synthesizing a crossover filter	patentable, beats achievement in field	Section 32.3 of Genetic Programming III
15	Rediscovery of a recognizable voltage gain stage and a Darlington emitter-follower section of an amplifier and other circuits	patentable, beats achievement in field	Section 42.3 of Genetic Programming III
16	Synthesis of 60 and 96 decibel amplifiers	patentable, beats achievement in field	Section 45.3 of Genetic Programming III
17	Synthesis of analog computational circuits for squaring, cubing, square root, cube root, logarithm, and Gaussian functions	patentable, new scientific merit, solves difficult problem	Section 47.5.3 of Genetic Programming III
18	Synthesis of a real-time analog circuit for time-optimal control of a robot	solves difficult problem	Section 48.3 of Genetic Programming III
19	Synthesis of an electronic thermometer	patentable, solves difficult problem	Section 49.3 of Genetic Programming III
20	Synthesis of a voltage reference circuit	patentable, solves difficult problem	Section 50.3 of Genetic Programming III

21	Creation of a cellular automata rule for the majority classification problem that is better than the Gacs-Kurdyumov-Levin (GKL) rule and all other known rules written by humans	new scientific merit, beats human best	Andre, Bennett, and Koza 1996 and section 58.4 of Genetic Programming III
22	Creation of motifs that detect the D–E–A–D box family of proteins and the manganese superoxide dismutase family	expert database	Section 59.8 of Genetic Programming III
23	Synthesis of topology for a PID-D2 (proportional, integrative, derivative, and second derivative) controller	patentable, beats achievement in field	Section 3.7 of Genetic Programming IV
24	Synthesis of an analog circuit equivalent to Philbrick circuit	patentable, beats achievement in field	Section 4.3 of Genetic Programming IV
25	Synthesis of a NAND circuit	patentable, beats achievement in field	Section 4.4 of Genetic Programming IV
26	Simultaneous synthesis of topology, sizing, placement, and routing of analog electrical circuits	patentable. beats achievement in field, solves difficult problem	Chapter 5 of Genetic Programming IV
27	Synthesis of topology for a PID (proportional, integrative, and derivative) controller	patentable, beats achievement in field	Section 9.2 of Genetic Programming IV
28	Rediscovery of negative feedback	patentable, beats human best, beats achievement in field, solves difficult problem	Chapter 14 of Genetic Programming IV
29	Synthesis of a low-voltage balun circuit	patentable	Section 15.4.1 of Genetic Programming IV
30	Synthesis of a mixed analog-digital variable capacitor circuit	patentable	Section 15.4.2 of Genetic Programming IV
31	Synthesis of a high-current load circuit	patentable	Section 15.4.3 of Genetic Programming IV
32	Synthesis of a voltage-current conversion circuit	patentable	Section 15.4.4 of Genetic Programming IV
33	Synthesis of a cubic function generator	patentable	Section 15.4.5 of Genetic Programming IV
34	Synthesis of a tunable integrated active filter	patentable	Section 15.4.6 of Genetic Programming IV
35	Creation of PID tuning rules that outperform the Ziegler-Nichols and Åström-Hägglund tuning rules	patentable, of scientific merit, new scientific merit, beats human best, beats achievement in field, solves difficult problem	Chapter 12 of Genetic Programming IV
36	Creation of three non-PID controllers that outperform a PID controller using the Ziegler-Nichols or Åström-Hägglund tuning rules	patentable, of scientific merit, new scientific merit, beats human best, beats achievement in field, solves difficult problem	Chapter 13 of Genetic Programming IV

A list of 36 human-competitive results produced by genetic programming. A detailed description of each of these results can be found in [Koza 2006], as well as a precise definition of the bases for claims of human-competitive performance. The naming of the concise terms in this column is our own.

To enumerate a few examples, a circuit that computes a cubic response function has been found automatically that was independently patented as a new invention after 2000 (Problem #33 in Fig. 1). At the time of the publishing of [Koza 1994], the best protein transmembrane domain detection algorithm was forged by genetic programming (Problem #9 in Fig. 1). Genetic programming has successfully induced the intermediates in a biochemical reaction pathway and inferred the relevant rate laws for the pathway [Sakamoto 2001]. GP has demonstrated success at missile avoidance strategies [Moore 1998], quantum circuit design (Problems #1, 3, and 5 in Fig. 1) [Spector 1998, Barnum 2000, and Spector 2003], and many other problems. In recent years, the Human-Competitive Performance Competition that takes place every year at the ACM's Special Interest Group on Genetic and Evolutionary Computation, affectionately known as the Humies, has highlighted several results with genetic programming that meet or exceed best results, such as the work on mesoscale time dynamics of activated states in chemistry by Sastry et al. [Sastry 2005], which won the Silver Humie in 2006.

It should be noted that much of the progress made by genetic programming has been through the application of massive quantities of computing power. Genetic programming is what is humorously referred to as an "embarrassingly parallel" problem. The computation time of genetic programming for attacking serious problems is dominated by the length of time required to evaluate candidate solutions. The success rate of GP depends to a very great degree on the difficulty of the problems, and the number of fitness evaluations that are performed.

Koza [Koza 2004b] is fond of using the analogy of a "brain-second", which he approximates at 10^{15} basic operations. This can be computed in several ways, one of which is to multiply the number of connecting synapses in the brain by the rate at which the neurons fire. There are on the order of a trillion synapses in a typical brain, and the fastest neurons take on the order of a millisecond to fire. He then argues that since it takes humans quite a long time to perform complex design tasks – hours to months, perhaps – it would seem to be logical to only expect similar feats of design after similar amounts of work. Therefore, Koza argues, we might expect that it might take some 10^{20} or 10^{21} operations before we can routinely and automatically induce solutions to hard problems. Koza cites such problems as Einsteinian relativity and Da Vinci's *volo instrumentale* design as examples. In his most recent video, he points out that his history of successes over time represents in part the results of applying exponentially increasing computer power to new problems [Koza 2004b]. While this may indeed prove correct, it is entirely possible that we do not need many millions of brain-seconds of operations to solve hard problems. It is certainly the case that vast numbers of the mathematical operations that our brain naturally performs do not go towards what we think of as problem solving. For instance, we perform arithmetic essentially by stepping through the multiplication algorithm that we all learned in school by rote. It takes hundreds of brain-seconds of work to solve simple problems such as "what is the product of 3 542 729 and 8 742.92 ?" By comparison, a modern microprocessor can solve this same problem in about two clock cycles, performing perhaps a million bit operations along the way. While humans have long held the lead in performing problem-solving operations, we submit that this is largely due to the lack of effective algorithms that can automatically make progress on solving problems. One major step along this path would be to automatically infer an algorithm from a problem description. Problem descriptions and semantic comprehension are not artificial intelligence's strengths as yet. We will settle for the more modest goal of inferring a simple algorithm from a function that measures progress towards a solution instead.

Summary of the Mechanisms of Genetic Programming

In [Koza 1992], Koza proposes using parse trees as the basic data type for Genetic Programming, and he develops key tree-construction, tree-crossover, and tree-mutation operators. One common variant of the genetic programming algorithm, a pure-crossover tournament selection GP, is given in EVOLVE-TREES. Here we have presented the most important parameters as formal parameters to the algorithm, and we have neglected, for the moment, minor parameters and the tree grammar as represented by its functions and terminals. We will reprise the function and terminal decisions in Chapter 6.

Algorithm 1a: EVOLVE-TREES

```
Input:  population size $M$, number of generations $G$, tournament size $T$;
        a fitness operator $fitness : \mathbb{T} \to \mathbb{R}$ for which smaller values represent better
        solutions; an algorithm RANDOM-TREE that generates a random tree; and an
        algorithm CROSSOVER that makes two children using the genetic information
        of the two parent trees provided as parameters
Output: a tree $t \in \mathbb{T}$ which has a small fitness

  **Population Initialization**: generate $M$ random trees
  for $j \leftarrow 1$ to $M$ do
    $trees_j \leftarrow$ RANDOM-TREE
  end for

  **Evolution**: generate $G$ new generations
  for $g \leftarrow 1$ to $G$ do
    **Evaluation**: score members of the population
    for $j \leftarrow 1$ to $M$ do
      $f_j \leftarrow fitness(trees_j)$
    end for
    **Crossover**: generate new individuals from the best individuals in the population
    for $j \leftarrow 1$ to $M$ by 2 do
      $parent_1 \leftarrow trees_{\text{TOURNAMENT } f,T,M}$
      $parent_2 \leftarrow trees_{\text{TOURNAMENT } f,T,M}$
      $children \leftarrow$ CROSSOVER $parent_1, parent_2$
      $newtrees_j \leftarrow children_1$
      $newtrees_{j+1} \leftarrow children_2$
    end for
    **Replacement**: update the tree population
    $trees \leftarrow newtrees$
  end for
  $best \leftarrow \arg\min_{i=1..M} f_i$
  return $trees_{best}$
```

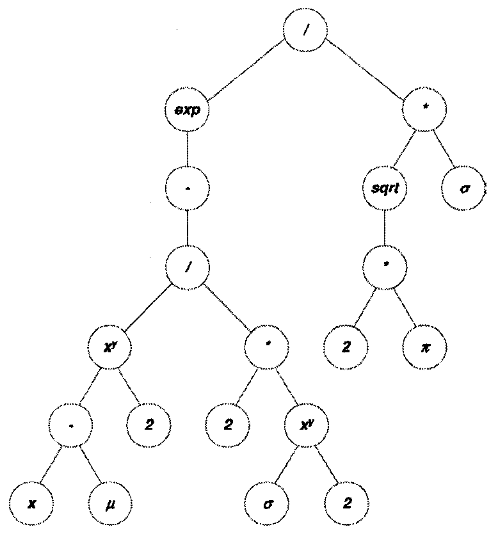
Algorithm 1b: TOURNAMENT

Input: vector of fitnesses *f*, population size *M*, tournament size *T*
Output: the index *i* that has the smallest fitness of those selected

```
  for $j \leftarrow 1$ to $T$ do
    $t_j \leftarrow random\ integer\ on\ [1, M]$
  end for
  $i \leftarrow \arg\min_{i=1..T} f_{t_j}$
  return $i$
```

Perhaps the most straightforward example of genetic programming in action is as applied to the problem of symbolic regression [Koza 1992]. For this problem, the goal is to search for the best equation that models a given data set. This differs from conventional linear and non-linear regression in that there is no predetermined functional form for the regressor. The genetic programming system instead recombines subexpressions to try to fit the relation under test. Suppose that we have observed some highly accurate data corresponding to the probability density function of the Gaussian curve. Our task is to reconstruct the functional form of the Gaussian from the sampled data using symbolic regression. This is accomplished by representing candidate solutions as a parse tree of operators and terms, as in Fig. 2. Here we show a parse tree corresponding to the Gaussian probability density function, $\frac{e^{-\frac{(x-\mu)^2}{2\sigma^2}}}{\sqrt{2\pi}\sigma}$. Genetic Programming (GP) manipulates such parse trees to make progress on and hopefully attain whatever goal is required, be it symbolic regression, shape optimization, feature identification, navigation, resource discovery and exploitation, and so on.

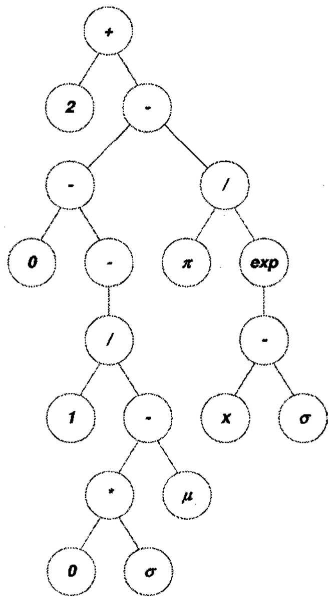
Figure 2



A sample parse tree for the probability density function of the Gaussian distribution, $\frac{e^{-\frac{(x-\mu)^2}{2\sigma^2}}}{\sqrt{2\pi}\sigma}$.

Of course, genetic programming will not typically begin its search at such an elegant functional form. A more likely initial state for such a problem would be a random function tree such as that shown in Fig. 3. Sean Luke in [Luke 2002b] summarizes several random tree generation algorithms. In this thesis, we use either Koza's ramped half-and-half or uniform tree generation. In EVOLVE-TREES, we use the external algorithm RANDOM-TREE to generate an initial population of *M* random parse trees. Each of these parse trees will be a randomly generated tree similar to that of Fig. 3.

Figure 3



A parse tree for a "typical" randomly generated candidate for a symbolic regression problem. This parse tree corresponds to the function $2+\left(0-\left(-\frac{1}{0\sigma-\mu}\right)\right)-\frac{\pi}{e^{-x\sigma}}$. As with virtually all parse trees generated by genetic programming, this function is unsimplified. We would typically report this function in its reduced form as $2-\frac{1}{\mu}-\frac{\pi}{e^{-x\sigma}}$.

A minor terminological note: in genetic programming (GP), we often speak of a collection of candidate *solutions* to a problem. "Solution" is understood to be a generic term for "parse tree which, when appropriately interpreted, provides a partial or complete fulfillment of the desired goal". In the context of symbolic regression, this is normally a particular function of the free parameters of the problem, such as those shown in Figs. 1 and 2. For symbolic regression in particular, the terms "function" and "solution" are often treated as if interchangeable.

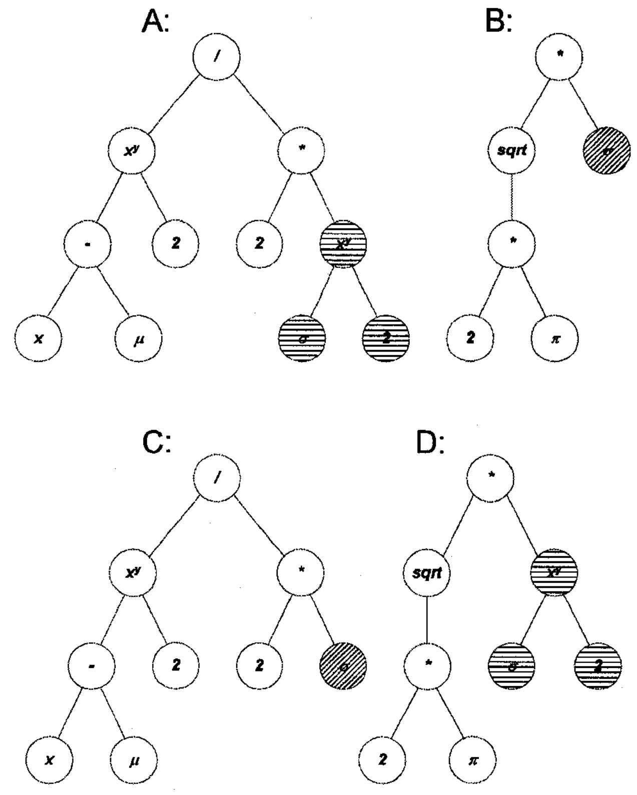
In the typical set-up for genetic programming, we require a *fitness function* that measures progress towards the solution. In symbolic regression problems, the test cases are normally defined over a region of interest in the parameter space corresponding to "interesting" values of the ordinate. The idea is that the behaviour of the target function over the region of interest should be representative of the general nature of the target function. For example, if we wish to approximate the Gaussian function, we should ensure that any evolved function will have the appropriate behaviour at many values of μ, σ, and x. Since the standard deviation σ is positive, we might choose the domain $[0.5, 2.5]$ as a reasonable range of values for σ. The mean of the distribution, μ, can take on both positive and negative values, so we might choose the domain $[-2, 2]$ for it. Finally, we need to choose an appropriate range for x. Since we are interested in capturing both the high amplitude and low amplitude regions of the Gaussian density function, we might choose the range $[-4, 4]$ as an appropriate range for x. For a symbolic regression problem, we normally evaluate a candidate by evaluating the regression error on a finite set of data points. This is useful both for testing the effectiveness of genetic programming and for finding new approximations over the rational or transcendental functions of any data series at hand. For instance, for the Gaussian regression problem listed above, we might pre-compute a set of 50 random values of $\{x, \mu, \sigma\}$ and evaluate the target function on these values. Notice that we do not make any claims of generalizability for the entire domain of the regressor. We simply choose an appropriate, representative subset of possible function values and continue with our regression. This frees us from doing computationally expensive correctness testing on our candidate regressor functions. This is the main "hopeful leap" of reasoning that genetic programming commonly makes. Fortunately, as demonstrated in the successful solution of many human-competitive problems listed in Fig. 1, this is often an appropriate inductive bias to use.

For each test case, we want to be able to algorithmically quantify how "close" each solution is to the known data. For instance, we may compute the mean absolute error of the predicted values, relative to the given data, as given by (1).

$$\varepsilon_{MAE} = \sum_{i=1}^{50} \frac{\left| f\left(x_i, \mu_i, \sigma_i\right) - Gaussian(x_i, \mu_i, \sigma_i) \right|}{50} \tag{1}$$

The fitness function *fitness* in this case would then be the mean absolute error of the candidate function and the precomputed data points. We also need to be able to generate new individuals from existing fit individuals. The standard technique for generating new individuals in genetic programming is crossover [Koza 1992]. Crossover is illustrated in Fig. 4. To perform a crossover operation, we first choose one node at random from each parental tree. We then exchange the subtrees rooted at each selected node. Often only one of the two possible children is created. The algorithm CROSSOVER implements the operation of Fig. 4 programmatically, including the selection of nodes in the trees and the prune and graft operations. It is given in full in [Koza 1992].

Figure 4



An example showing the operation of the crossover operator of genetic programming. In this example, the striped subtree of tree A is crossed-over with the diagonally hatched subtree of tree B to yield the two offspring trees C and D. Notice how this recombines some of the "meaning" of each parent while retaining elements of the primary parents of each tree.

Using crossover exclusively offers us a genealogy of successful solutions, which explains our adoption of the genealogical nomenclature; we say that the two chosen solutions are the parents of the new child or children. The children may be superior or inferior to their parents; we hope that they are better with non-vanishing probability. We term the set of all parents at a given generation the *parental population* at that time, which gives rise to a *child population.* The child population will normally replace the parental population in the next generation, although there are many variations on this theme.

This begs the question of which solutions we shall choose as parents for newly generated individuals. Many techniques have been proposed; Koza uses roulette-wheel selection in his early work [Koza 1992, Koza 199x], but an effective strategy used nearly universally nowadays is tournament selection. The simplest tournament selection is for a tournament of size two: choose two individuals at random without replacement from the population, and the individual with the better fitness gets to be a parent. A more commonly encountered tournament is one of size 7. Here, seven individuals are chosen at random from the parental population, and the best among them is declared the winner of the tournament. Normally in evolutionary computation, better values correspond to smaller fitness, as for the mean absolute error, (1). The algorithm TOURNAMENT above gives an implementation of tournament selection.

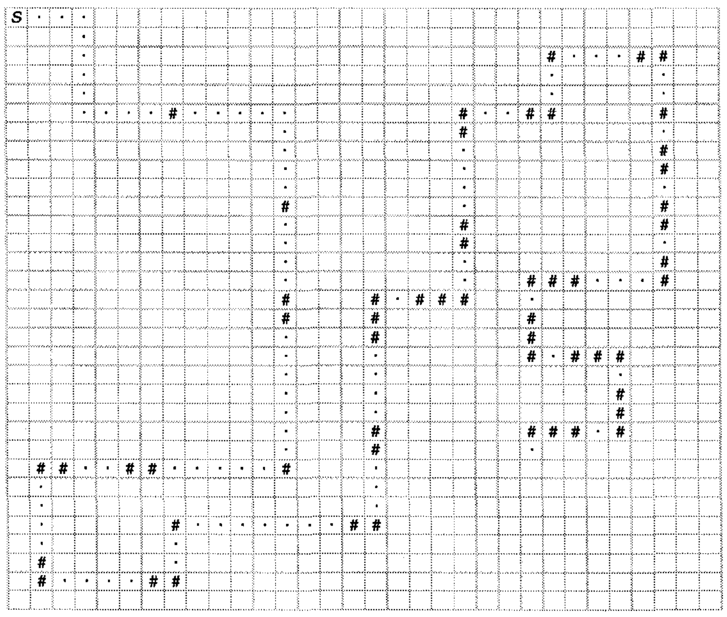
After all the new children are produced, we then replace the parents with the children and say that a *generation* is complete. This process is repeated several times, with a default of typically 50 generations being common. We will have more to say about what generation number to choose in our section on modelling for GP, in Chapter 6. Often, the GP practitioner will have the genetic programming system copy the single best individual from the parental generation as the first child produced, which is termed *elitism.* Elitism ensures that the best fitness as a function of generation number does not worsen, although it may lead to less strong performance in rare cases. This problem, called premature convergence, will be discussed in the section “Limits on Genetic Programming.”

The Artificial Ant on the Santa Fe Trail Problem

We now describe a common benchmark problem for genetic programming, the artificial ant on the Santa Fe trail [Koza 1992b]. This well-studied problem is a common benchmark for testing genetic programming and other automated programming systems against one another. While Langdon and Poli showed [Langdon 1998] that it is a difficult problem for genetic programming, its elegance and its difficulty for GP makes it an excellent benchmark for the methods of this thesis. We have used it extensively to test the statistical methods of Chapters 3 and 4, and it is the basis of Chapters 5 and 6. Therefore, we will describe the problem in detail here. We follow the description of [Langdon 1998].

The problem is to automatically discover a program that can navigate an artificial ant along a twisting trail on a 32x32 toroidal grid. Fig. 5 shows the Santa Fe trail.

Figure 5

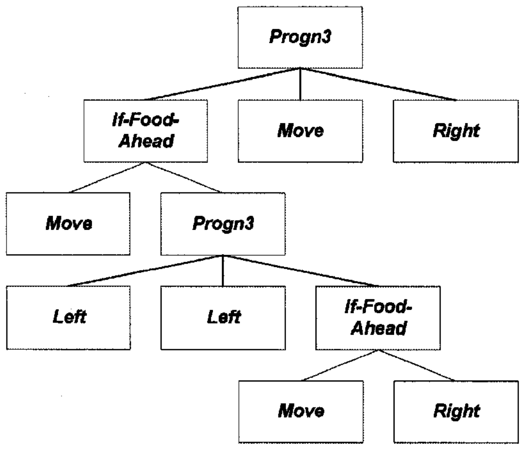


The Santa Fe trail. The artificial ant begins in position marked with an "S", facing to the right. There are 89 food units in the 32x32 toroidal grid, denoted by the "#" symbol. The middle dot "·" represents the shortest path between successive food symbols.

The artificial ant begins facing to the right in the top-right corner of the world. The world is toroidal, so opposite edges communicate with each other – the ant can go without penalty from bottom to top and left edge to right edge of the world. Ant programs use three operations to move about the world: Left, Right, and Move. Left and Right turn the ant in the appropriate directions, and Move moves the ant ahead 1 step and eats whatever food is in the cell ahead. Each of these operations takes one unit of time.

There is a single sensing function, IfFoodAhead, which looks into the cell immediately ahead of the ant. This function takes two subtrees as parameters; if food is ahead, the first subtree is executed. If no food is found ahead, the second subtree is executed. Two other sequencing functions are provided, named Progn2 and Progn3. They consist of two or three parameters, respectively, and simply execute their parameters in order. A successful program of minimal size that solves the artificial ant on the Santa Fe trail is shown in Fig. 6 [Langdon 1998 and this thesis].

Figure 6



A program that successfully solves the artificial ant on the Santa Fe trail problem. This program is among the 12 solutions that exist of length 11; there are no solutions smaller than this one.

The ant is given a time limit, normally 600 time steps, in which to eat all the food. The ant programs are stopped after 600 time steps if they have not eaten all the food, and a fitness score is given based on how much food they have eaten. The fitness function for this problem is given by 89 - # *food eaten*. Since there are 89 units of food along the Santa Fe trail, a perfect individual gets a score of 0.

Next Steps

We continue in the next chapter by introducing and discussing the notion of scalable genetic programming. We will provide a bit of a taxonomy of evolutionary computation systems, and use this to situate scalable genetic programming tasks among other sorts of problems that evolutionary computation is asked to solve. We will then introduce some of the contributions that this thesis makes towards achieving scalable genetic programming, notably the scientist algorithm. In later chapters, we will introduce two different ways of automatically making subroutines from useful bits of programming code, which is central to achieving scalable genetic programming.

This page is intentionally left blank.

Chapter 2: Limits on Genetic Programming, a Problem Taxonomy, and Scalable Genetic Programming

Evolutionary computation has been used to solve a diverse set of problems. In this chapter, we will provide a first attempt at a taxonomy of evolutionary computation problems. This naturally leads into a classification of problems largely in terms of the degree of abstraction involved. This will allow us to precisely define the term "scalable genetic programming" as used in the title of this thesis, as well as flesh out the nature of problems of interest to genetic programming (GP) researchers.

Limits on Genetic Programming

We are interested here largely in practical limitations on genetic programming's ability to induce solutions to posed problems. We hope to avoid being bogged down in a philosophical discussions about what is required for inferring a particular solution. In engineering, we are primarily interested in the performance envelope of a new technique, not so much in its theoretical limits.

However, even though the successes of the previous chapter are impressive, there remain theoretical tactics that we use to solve difficult problems that GP, as currently formulated, does not and cannot use. For instance, standard GP is generally evaluated using a simple one-dimensional fitness function. That is, each individual is given a scalar score that combines the results of many distinct fitness tests in a preset fixed formula. This formula is often just the average or sum of the fitness tests performed. One example that we have already seen is for the problem of symbolic regression. Here, the fitness function is the mean absolute error of several fixed test cases.

One might think that if the full fitness data were made available to the selection and evolution process, GP might be able to make better use of the data. GP normally combines pairs of trees to make candidate solutions in the next generation. If the information were available, GP might be adapted to try breeding two parent functions that each locally fit half of the points of a symbolic regression function to make a new child. By analogy, we don't debug programs that human programmers are writing by evaluating their fitness as a single scalar. The reader is encouraged to read Appendix 1, "The GP Room", to get a sense of how difficult this task is. Through the use of interactive debuggers, we essentially enable the programmer to query every data value at each point throughout a computation. Even in systems where random data inspection is not possible, print statements or the like are used to view intermediate progress and aid in debugging. If we remove this ability from human programmers, we can imagine that their performance would dwindle greatly. If the problem-solving system can make use of it, additional information on intermediate outcomes is very useful for program induction.

We can imagine a further refinement in this direction, where instead of having a pre-selected fixed set of points to consider, the GP is allowed to choose points to evaluate. This more closely approximates how a person might "guess" the form of an unknown function – select a few points of interest in the function's domain, and then evaluates the unknown function at these points to get a sense of how the values vary as a function of the parameters. Indeed, lacking a formal proof system which can verify a function's correctness over an extended interior range of the domain, the only alternative is to have the regression system itself search for challenging points. It might, for instance, propose candidate test cases which are selected for being difficult to predict and then adapt a regression in progress to fit the new, difficult points. An accurate regressor has few such points, resulting in an accurate solution over the entire domain of interest. An analogous process can be readily defined for GP problems that are not symbolic regression, per se.

A second point to notice is that GP must make all its design advances anew with each successive run. A common trick used in the analysis of GP's proficiency at a given task is to perform many random restarts of the stochastic genetic programming algorithm. We then take the best of several runs as the best-of-set solution. However, it seems that if we could somehow remember some of the design innovations from the first run during the second run, we might improve the overall performance of genetic programming on candidate problems.

A third idea is to consider the methods by which people make progress on algorithmic problems. It seems to us that working computer scientists tend to follow certain steps as they progress on a new problem. Appendix 1 identifies several tools of the trade. They include the careful consideration of simple cases, perhaps by enumeration; solution of simple cases; generalization of specific solutions to a more general case; simplification to previously solved programs; breaking problems into subproblems when you aren't making progress; searching for exceptional cases that break the general solution in progress; working a particular instance of a problem class while trying to develop an algorithm for solving a class of problems; and case-based remembering and adaptation of similar, past solutions. Many of these tricks may not be easily algorithmizable, either because of difficult problem-recognition issues or because of access to a wide literature that isn't easily codified.

Chapter 2: Limits on Genetic Programming, a Problem Taxonomy, and Scalable Genetic Programming

A number of other tradeoffs generating current genetic programming research demonstrate how our scientist algorithm can be useful. GP is not immune to the problem of parameter settings - for instance, what population size should we use, how many generations should we run the GP for, what geometry should we use for the individuals and so on. Some researchers have taken on this problem as well [Luke 2002], but it is difficult to get conclusive answers because of two problems. First, while theory has been progressing in genetic algorithms and genetic programming, there is not yet any across-generations estimator of the best fitness of a population. Given that the progress of an evolutionary computation technique depends on the fitness landscape that it is searching, closed-form solutions will likely evade us on anything but toy problems.

Second, experimental approaches are challenged by a lack of good statistical tools in evolutionary computation. We will consider this problem again when we are demonstrating the utility of our statistical framework in Chapters 3 and 4.

It has been argued by Dan Dennett [Dennett 1991] that human cognition is a sort of "toolkit" – that there are a number of very general rules that are pressed into service as needed by the problems currently under examination. It is the author's intuition that some of these tools might be pressed into service algorithmically, to build on the already general-purpose and effective problem-solving technique of genetic programming.

Finally, it is rare to find the search space for a problem in genetic programming well-characterized. The pioneering work in this regard was "Why Ants are Hard," by Langdon and Poli [Langdon 1998]. For tree size-limited GP, we can characterize the search "space" as the set of all trees with given node types of a certain maximum size or lower. For depth-limited GP, the search "space" is the set of all trees with given node types of a certain maximum height or lower. Building on the work of Langdon and Poli, we will consider, develop and discuss the search space that GP manipulates by providing a technique for indexing all trees in Chapter 5. We will also provide an algorithm for choosing a particular tree based on its lexicographical ordering. This algorithm enables a breadth-first search over the space of trees, which allows an exact solution of the Kolmogorov complexity [Li 1997] of a problem in tree space. This is in fact possible for some problems in GP, such as for the Artificial Ant on the Santa Fe trail [Langdon 1998].

Problem Taxonomy

Towards scalable genetic programming… an interesting title. What do we mean by "scalable" genetic programming, exactly? We will begin by explaining what it is not. The word "scalable" in this context is to be discriminated from its typical use in computer science, where it has the meaning of "an algorithm that completes in polynomial time as a function of the complexity of the input." This usual definition of "scalable" is not even well-defined in the context of genetic programming, as the difficulty of finding a program that solves a posed problem $\mathcal{P}$ cannot be expressed as a scalable function of its input length. This difficulty becomes infinite for any problem grammar that enables the expression of the Turing halting problem [Turing 1936], so a general expression would be useless. Another difficulty is in measuring the amount of work required to find a program that solves a given problem; this would seem to be impossible in practice without access to a technique such as genetic programming that can routinely solve such problems.

Scalability in the context of this thesis refers instead to the idea of finding *solutions* to automatic programming problems that are scalable. This is equivalent to requiring a human programmer to find an algorithm that solves the posed problem for any value of an input complexity parameter. To shed light on this process, it will be useful to provide a taxonomy of problem classes for which evolutionary computing methods are marshalled.

There are a range of problem classes that evolutionary computation solves or attempts to solve. These can be roughly distinguished by three broadly independent parameters, which we call genetic data, environment and outcome. The genetic data can essentially be any type of data structure, from simple Boolean vectors through reals, real vectors, matrices, permutations, trees and graphs to combinations of these fundamental types. Perhaps it is not obvious, but the outcome can be an arbitrary data structure as well. Typically, we use a scalar real value as the primary feedback to evolution, often called the "fitness function", but there is no fundamental need for this. Multiobjective optimization, for instance, uses a vector as the outcome of the fitness evaluation. Often, the natural outcome of a fitness evaluation is a multidimensional result, which is then coerced into a scalar value for optimization purposes. Indeed, much of the hard work of using evolutionary computation (EC) successfully is in determining how to convert the response of evaluation into a scalar so that the adaptive landscape of the fitness function is smooth. This normally takes place by assigning arbitrary penalty and reward functions to reduce poor performance of an evolutionary computation system and to encourage positive performance. Typically, a period of iterative fine-tuning follows in which initial poor choices are edited in light of poor performance of the EC system. We can view the outcome of a fitness evaluation as a detailed record of fitness metrics. These fitness metrics can, at present, be simplified to a scalar to serve as feedback to the evolutionary system.

Finally, we come to "the environment". The trouble in evolutionary computation is that the environment is variously defined, depending on the author. In a strictly logical model of evolutionary computing, completely removed from biology, we can define the environment as that set of data, when augmented with the genetic data, causes the outcome to be a pure function of the environment and the genetic data. That is, the outcome of a fitness evaluation depends on the union of the environment and the genetic data of the individual. Two further complications assert themselves: stochastic EC systems and coevolutionary systems. Stochastic evolutionary computation systems, in addition, may give different results each time when run - even though the environment and the genetic data remain the same. Only deterministic EC systems will give deterministic results that depend only on the environment and the genetic information of the individual. Coevolutionary systems depend on more than one individual to produce a result. If we view the genetic information as the union of the genomes of all participating individuals, our simple model still holds.

When we define the environment in this way, we can classify EC systems by the kinds of environment which we hope to optimize. A first cut at such a list would include:

- Simple scalar environments. Here the environment is fixed for all genetic individuals. An example would be the Artificial Ant on the Santa Fe trail, which always uses the same trail, of the same size, with the same food in the same positions, with a fixed ant initial position and direction.

- Simple vector environments. The environment is a given set of environments over which fitness evaluation is measured. An example would be symbolic regression, which is normally performed over 50 or 100 particular data points. The outcomes of the error between the evaluations and theoretical curve are typically summed either in quadrature or in absolute value, and thence reduced to a single fitness value.
- Generative environments. Here the environment is typically an infinite or a very large set of cases, often the power set of a problem instance. This is equivalent to requiring that an EC solution give an appropriately correct result over all legal input values of some input set. Simple vector environments are often used as a proxy to represent a generative environment, as in symbolic regression. Alternately, the entire power set of a problem space may be enumerated and used in extension to validate a candidate solution. Even-k-Parity, a benchmark problem defined in [Koza 1992] and detailed in Appendix 1, uses this latter approach.

  We can make a sublist of the variants of generative environments:

  - fixed subset approximation of a generative environment,
  - co-evolved approximation of a generative environment,
  - proof-based models of a generative environment, and
  - enumeration of a generative environment.

Finally, additional levels of sophistication can be added to the above model. For example, we might have a situation in which the complexity of a problem depends on an input parameter. This would result in a scalable generative environment, where the complexity of the problem is selectable by an external process.

- Scalable generative environments. Here the environment is scalable in an input parameter. One simple example would be the product set of a simple data type. For the sake of argument, let us call this data type *D*. The environment would then be some product set of *D*, D^k, which selects the "difficulty" of the problem. Success on problems in such environments approaches the main subject matter of computer science, namely algorithm design. For instance, being able to successfully write a computer program which can sort an arbitrary list of integers would be a problem with a scalable generative environment.

Traditional problem classes

We can also identify different problem classes by the way that problems are traditionally approached in evolutionary computation. Here we have performed a rough ordering of the problem types by the level of abstraction of the task and the generality of the solution provided.

- Boolean hypercube domain optimization: The realm of the standard genetic algorithm. This is the case where the fitness function is of the form $f:\mathbb{B}^k \to \mathbb{R}$, and normally solved by genetic algorithms or their progeny [Holland, 1975].
- Numerical optimization: a fitness function $f:\mathbb{R}^k \to \mathbb{R}$ is presented which represents a function to be optimized – either minimized or maximized. This is properly the domain of *evolutionsstrategien* [Schöneburg 1994], although genetic algorithms have long been adapted for this problem. In our earlier analysis, this would be a simple scalar environment, with genetic information being of the form $\mathbb{R}^k$ and with the outcome of the form $\mathbb{R}$. Of course, there are many non-evolutionary ways of performing numerical optimization as well, treated in depth in books such as Press et al. [Press 1992].
- Multiobjective real-valued optimization: $f:\mathbb{R}^k \to \mathbb{R}^m$, and we are interested in "good points" in $\mathbb{R}^m$, ideally minimizing all dimensions of $\mathbb{R}^m$ at once. Normally there are intrinsic tradeoffs between the *m* different evaluation variables, so no single solution can optimize all functions at once. This results in the creation of a Pareto front [Pareto 1906], where solutions along the Pareto front beat other solutions in

some outcome variables but not others. Multiobjective optimization algorithms such NSGA-2 [Deb 2001] are normally used in this case.

- Tree-based function optimization: We search for a tree $t \in \mathcal{T}$ that optimizes a fitness function $f : \mathcal{T} \to \mathbb{R}$. The tree is derived from some combinatorial space consisting of nodes chosen from a fixed node set, possibly applying rules enforcing syntactic validity. This is the equivalent of numeric optimization where the object to be optimized is most easily represented by a function tree. Genetic programming (GP) is often used to perform this kind of optimization.
- Symbolic regression to known values: We search for a function $f : \mathbb{R}^d \to \mathbb{R}$ such that an auxiliary function $fitness = \sum \|f - f_{true}\|$ is minimized. The candidate function is chosen from some combinatorial space consisting of a function tree made up of elements picked from some basis set. This is the traditional domain of GP.
- Symbolic regression to a known function: We search for a function $f : \mathcal{P}(\mathbb{R}) \to \mathbb{R}$ such that an auxiliary function $fitness = \int \|f - f_{true}\|$ is minimized. The candidate function is chosen from some combinatorial space consisting of a function tree made up of elements picked from some basis set. Genetic programming has also been used for this problem, normally by sampling the codomain at a finite number of points and proceeding as in the previous case. This problem also occurs when generating an interpolating and/or extrapolating function for a given set of data points.
- Program instance induction: Search for a program $p : X \to Y$ that takes an input $x \in X$ and produces an output $y \in Y$ minimizing a fitness function $f : Y \to \mathbb{R}$. An example would be the artificial ant on the Santa Fe trail problem: for one particular

board with 89 units of food on it, evolve an "ant" that eats all 89 units of food in 600 time steps. This kind of problem is normally accomplished by genetic programming, where an abstract syntax tree (AST) is evolved that represents the program *p*. This differs from the previous cases in that the program represented by the AST is interpreted by an interpreter. The program thus executed manipulates the input data to produce the correct output.

- Program induction: Search for a program $p : X^n \to Y$ that takes a set of *n* inputs $x_i \in X$ and produces outputs $y_i \in Y$ each of which minimizes a fitness function $f : \{X, Y\} \to \mathbb{R}$. Often the candidate program should perform correctly on the power set of the input space; in this more general case, we typically sample the power set to "reduce" the exponential input space to a countable number of test cases. An example would be the Even-4-Parity problem [Koza 1992]: evolve a boolean function that returns TRUE if an even number of the 4 provided boolean inputs are TRUE, FALSE otherwise. This is a more abstract version of genetic programming.
- Coevolutionary program induction: As in the program induction case, save that instead of a fixed set of test cases to attempt, an evolutionary system is used to evolve challenging problems for candidate solutions. That is, a separate evolutionary system of any of the types discussed so far generates the problem instances *X* for the problem-generation code to attempt to solve. It is generally easier to evolve a difficult problem instance than it is to evolve a correct program that optimizes that instance. Therefore, the fitness function for the problem-evolving code is typically designed to try to choose easy tasks near the beginning of evolution, moving to more challenging ones as the solving programs improve.

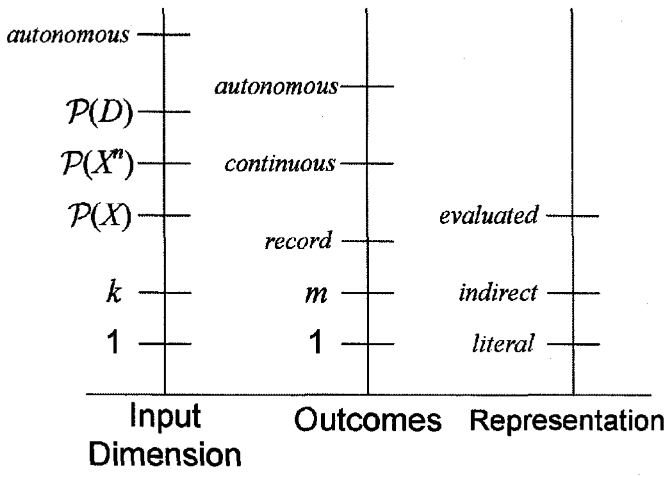
- Validated coevolutionary program induction: As in the coevolutionary program induction case, but a verifying oracle $v: X \to \mathbb{B}$ is provided which authenticates the validity of an instance of the input set *X*. This restricts the evolutionary system that generates the challenge instances to attempt only valid problems.
- Provable program induction: As in the program induction case, save that optimal behaviour is required over the entire power set of the input – that is, for all possible inputs, an optimal outcome should obtain. This can either be achieved by enumerating all feasible inputs and testing on them all, as in Even-4-Parity, or by some proof mechanism that can verify whether a proposed solution is correct for all possible input values.
- Scalable program induction: Search for a program $p: X \to Y$ that minimizes a fitness function $f: \{X, Y\} \to \mathbb{R}$ for each possible input *X* of a scalable input function. The scalability component is achieved using an oracle $O: \mathbb{N} \to X$ that can generate a problem instance of a given challenge level $n \in \mathbb{N}$. An example would be the task of inducing the algorithm sort-integer-array which sorts a provided vector of size *n* of 32-bit integers into ascending order. Another example would be to induce an algorithm for computing the convex hull of a set of *n* points in 2 dimensions. As with program induction, genetic programming is typically used to represent and evolve abstract syntax trees, which are evaluated by interpretation.
- Scalable algorithm induction: Generalize the scalable problem induction task defined previously to replace the concrete data type governing the data with an abstract data type *D*. An example would be to induce a generic sorting algorithm that sorts a provided vector of arbitrary data types possessing a sorting predicate $S: D^2 \to \mathbb{B}$ into

ascending order as defined by *S*. Another example would be to induce an algorithm for computing the convex hull in an arbitrary number of dimensions *d*. To our knowledge, this level of proficiency has not been accomplished in automatic program induction.

- Program definition and induction: Given a fitness description in terms of a set of benefit functions, induce a problem and a solution to the problem in tandem. This would represent the grail of automated programming, automatic progress on problem definition and its solution simultaneously. It remains a conjecture at present.

These two approaches to categorizing can be modelled in terms of the variability dimensions of the evolutionary computation system. Three obvious variables useful in categorizing these systems are shown in Fig. 1: the input dimension, the number of outcomes, and the indirectness of the representation. Some variability levels are shown for each variable and are explained in the figure caption – these are meant to be indicative, rather than exhaustive, and may overlap in practice. We can view any particular EC system as setting three sliders along the variability dimensions of Fig. 1: for instance, the program instance induction problem such as the artificial ant on the Santa Fe trail would have an input dimension of 1, an outcome dimension of 1, and is an evaluated representation. We should probably also have a variability level related to the sophistication of the environment – for example, in the artificial ant on the Santa Fe trail problem, the environment is more sophisticated than for symbolic regression. Another variability level might be the input data type. There is, however, a deeper problem for a

Figure 1



Variability dimensions and some levels of variability for different evolutionary computation tasks. The input dimension may be single-valued, or vector-valued. Higher levels of input dimension include the power sets of a task, from the power set of a concrete variable, to the power set of a scalable variable, to the power set of an abstract data type. Finally, we might define an autonomously determined input dimension as the highest level of abstraction. Outcomes from a fitness evaluation may take the form of a simple scalar, a vector of related items, or a fitness record as described in the main text. We term the ability to measure the interim evaluation state at any point during an evaluation as a "continuous" outcome. We also use "autonomous" as an outcome level, where the EC autonomously determines what outcomes from a state variable are salient. The representation may be a literal coding of a fitness solution; it may be indirectly coded, or it may arise from interpreting the tree as source code and evaluating it.

rigorous categorization of EC systems along these lines. We can define embeddings for each of these problems into one another, subject to the No Free Lunch theorem. For example, take the class of single-objective function optimizations $f:\mathbb{R}^k \to \mathbb{R}$. This problem can be embedded into the boolean-value optimization in many ways, for instance by using the IEEE 80-bit double floating-point representation of each number. This gives the problem representation $f:\mathbb{B}^{80k} \to \mathbb{R}$, which is isomorphic to the boolean-space optimization. In practice, however, running an *evolutionsstrategie* (ES) on the real-valued representation tends to outperform running a GA directly on 80-bit float representations.

Embedding and interconvertability of representation

The implications of this embedding idea are many. Optimizing one form of a function is often observed to be an easier task than optimizing another form, even through they are nominally interchangeable. We should be satisfied with an approximate categorization of problem classes, rather than a rigourous mathematical one.

Suppose that we wish to perform a single objective optimization of an unknown 5-dimensional fitness function *f* as shown in (1).

$$y = f(x_1, x_2, x_3, x_4, x_5) \tag{1}$$

To encode the problem in our computer, we represent each of the input variables and the fitness outcome as IEEE double-precision floating point variables using 80 bits. The codomain of the function is then $5 \cdot 80 = 400$ bits of data, with an output of 80 bits of data. Since the codomain of this function is closed under permutation, a version of the No Free Lunch theorem [Wolpert 1995] will apply over such a function set. We immediately know that no optimization technique can do better, on average, than enumeration. We can compute the size of this function set as $\left(2^{80}\right)^{2^{400}}$, or $10^{10^{121.79\ldots}}$. Since our feasible program space knows nothing *a priori* about the input being offered, all interpretations are equally valid. For instance, one reasonable fitness function would be that represented by LENGTH-OF-INPUT-AS-A-STRING.

Algorithm 1: LENGTH-OF-INPUT-AS-A-STRING

```
Input: 5 80-bit IEEE floating-point numbers $x_1, x_2, x_3, x_4, x_5$
Output: an 80-bit IEEE fitness value $f(x_1, x_2, x_3, x_4, x_5)$.

    $addr \leftarrow$ address of $x_1$
    $a \leftarrow$ interpret $addr$ as the beginning of a null-terminated ASCII string
    $a_{50} \leftarrow null$
    answer $length(a)$ as an IEEE 80-bit floating-point value
```

While it may not seem to be a useful function, this is certainly a legitimate function in the setup over which the No Free Lunch theorem applies. Other possible interpretations of the input data include:

- A binary bit string of 400 bits
- An ASCII string of 80 characters or less
- A modest-sized GP function tree consisting of at most 100 nodes chosen from a set of 16 component nodes
- One 400-bit floating point number
- 5 or fewer 80-bit IEEE doubles
- 10 or fewer 40-bit IEEE floats
- 12 or fewer 32-bit integers

- 25 or fewer 16-bit fixed-point reals
- a C++ record consisting of 400 bits of information, total
- a variable-size bit string of 391 bits or less
- a TSP path among 44 or fewer cities

...and so on. The point of this analysis is that for each of the above data types, we can imagine a problem class which takes the data set as input. We can make a similar argument for any of the variability dimensions that we described earlier – outcome dimension, solution complexity, environment complexity, or representation indirectness. All of these variability dimensions fall prey to the No Free Lunch theorem, suitably expressed. Nonetheless, data types are only valuable as programming constructs inasmuch as problems *themselves* are best modelled by one of these sets of data types. We intend the preceeding categorization of problem classes into abstraction levels in the restricted sense of "typically encountered computable problems". The point of this thesis is to engage the "scalable genetic programming" level of problem. Before we begin, we will briefly mention some of the properties that an automated problem-solving system should have to solve problems of algorithm inference.

Properties that any Successful Automated Programming Algorithm should have

We will now explore some of the features mentioned in the above text. Fig. 2 gives a list of "useful insights" in tabular form that are either necessary or productive in the automated solution of scalable genetic programming problems.

Figure 2

Insight or Property	***Algorithmic Implementation***	***Solution***	***Chapter***
Validation is easier than solution	Define an oracle $F(s)$ to "score" candidate solutions	fitness function	*literature*
Exhaustive coverage of problem space is hard	Define an oracle $O(p)$ to generate problem instances on demand	dynamic problem generation	Ch. 2
...and EC is effective on hard problems	Use an EC system to look for challenging cases for a candidate solution	co-evolution	Ch. 7
Systematically working up from enumerable base cases to a correct general solution often works well	Extend the oracle $O(p)$ to include a scalable difficulty parameter $O(p, n)$	dynamic problem generation	Ch. 2
	Enumerate all simple problem instances of a given size	problem space enumerator	Ch. 5
Reducing a problem into the basic variants of each class and solving each separately is often effective	Clustering or other analysis of fitness values for candidate problems	evaluation and clustering algorithm	Ch. 7
	Automatic generation of subroutines from fit individuals	subroutine-making code	Ch. 5, Ch. 7
Generating all possible solutions and evaluating them can identify valuable building blocks	Enumerate all simple solution instances of a given size	solution space enumerator	Ch. 5
	Evaluate candidate solutions	evaluation	*literature*
	Automatic generation of subroutines from fit individuals	subroutine-making code	Ch. 5
Automatic progress on hard problems will require a generate-and-test methodology	Generate-and-test framework	scientist algorithm	Ch. 5
	Promising pathways should be tracked and used		Ch. 2; Ch. 6
	Useful subroutines should be available to new runs	archiving and tracking code	Ch. 7
Reliable decision-making is required for deciding between methods and evaluating techniques	Efficient, automatic statistics suitable for EC algorithms	good statistics for comparing solutions with different computational effort	Ch. 3; Ch. 4; Ch. 6

A list of insights which would seem to be either required or highly productive for reliable automatic scalable program induction. We have indicated the chapter in this thesis in which the issue is discussed or introduced. In some cases, relevant data are presented in several different locations in the text.

While implementing and characterizing progress on the all of the tricks listed in Fig. 2 might take a lifetime, this thesis does explore and implement some of the low-hanging fruit in this domain. Many theses in artificial intelligence introduce a number of "clever tricks", implement them all and consider performance when using them jointly on a hard problem. While this is a justifiable tack for attacking a difficult engineering problem, it unfortunately confounds the effects of all the implemented tools, making generalization of the research difficult. A more scientific approach would be to define and characterize each innovation separately where possible, and to conjoin techniques only where logically necessary or useful. This thesis adopts the latter approach. We believe that the combination of the techniques of Fig. 2 will ultimately be particularly powerful, but we would prefer that they be characterized as useful individually. This has the perhaps unfortunate consequence that no "great synthesis" of all the techniques presented in this thesis is performed. We feel that the sundry contributions performed by this work will satisfy the title of the thesis, "Towards Scalable Genetic Programming."

Chapter 2: Limits on Genetic Programming, a Problem Taxonomy, and Scalable Genetic Programming

Scalable Genetic Programming and the Scientist Algorithm

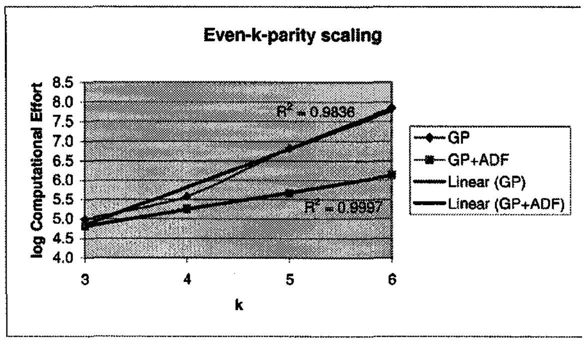
One serious criticism that might be levelled at genetic programming is that it fails to make effective use of subroutines in automatic programming. The ability to parameterize specific algorithms while retaining correctness underlies the fluency of most human-crafted algorithms. Being able to reliably create useful subroutines would go some way towards making automated problem-solving human competitive. The founder of genetic programming, John Koza, has accordingly made several independent attempts at automatically reusing code, including automatically defined functions (ADFs) [Koza 1994], automatically defined macros (ADMs) [Koza 1999], and others. Genetic programming with ADFs has been contrasted explicitly against genetic programming without ADFs. We can use this set of data to illustrate the idea of scalable genetic programming. The effects of using ADFs on problem difficulty for the Even-*k*-Parity problem are shown in tabular form in Fig. 3 and are shown visually in the logarithmic plot of Fig. 4.

Figure 3

	Computational Effort		*log Comp. Eff.*	
k	*GP*	*GP+ADF*	*GP*	*GP+ADF*
3	96 000	64 000	5.0	4.8
4	384 000	176 000	5.6	5.2
5	6 528 000	464 000	6.8	5.7
6	70 176 000	1 344 000	7.8	6.1

Approximate computational effort as a function of the number of Boolean predicates considered, *k*, on the Even-*k*-Parity problem. Computational effort numbers are accurate to roughly a factor of two, given the number of runs performed in these experiments. Trials are performed both with Automatically Defined Functions ($GP + ADF$) and without (*GP*). The data given for $k = 6$ for the GP treatment are extrapolated from the prior data, and are consistent with the absence of any successes in 19 trials. All data are taken from [Koza 1994].

Figure 4



A graph of the logarithm of the approximate computational effort required to solve various Even-*k*-parity problems with 99% confidence. Computational effort is determined by finding the generation that minimizes the computational effort, and then repeating the GP procedure with independent trials to achieve 99% odds of success. For the run counts used in these experiments, computational effort is roughly accurate to within a factor of 2 in value, or ±0.3 in the logarithm. Trials are performed both with Automatically Defined Functions ($GP + ADF$) and without (*GP*), which are seen to reduce the slope but not change the slope or degree of the curve. Linear regressors and their coefficients of variability are given for each series. All data are taken from [Koza 1994].

The performance gains are impressive indeed; an estimated factor of 50 speedup is achieved at $k = 6$ using ADFs. As the trend remains upwards when graphed on a log-linear scale, the computational effort remains exponential in *k*, even after using ADFs. In contrast, most human programmers can solve this problem for any *n* in constant time, by means of a loop and a subroutine that computes EXCLUSIVE-OR. To be fair, the genetic programming system for this problem does not have access to a looping primitive. If one is supplied, however, genetic programming does still not make much progress on the problem unless EXCLUSIVE-OR is provided as well. This, then, is the central issue of scalable genetic programming – to solve algorithm-coding problems as humans do, by performing little experiments to identify, break down, and solve hard problems. We might anticipate that this is a challenging task; accordingly, in this thesis we will merely illustrate how we might go about beginning to implement scalable genetic programming.

Scalable Problem Difficulty

Problems have a natural difficulty level associated with them. Indeed, many problems have a difficulty level that can be easily turned into a free parameter. For instance, consider the artificial ant on the Santa Fe trail problem [Koza 1992b]. This problem has a natural challenge level associated with it, namely the number of time steps required to solve the problem. The reference choice for the problem, 600 steps, represents an intermediate level of difficulty. To give an indication of the scalability of computational effort with challenge level, we can perform an analysis of exact computational effort for an enumerable problem, which we will expand upon in more detail in Chapter 5. Figs. 5 and 6 show how the computational effort changes as a function of problem difficulty and subroutine set used.

Figure 5

Time Steps	*Normal*	*+ IFAM*	*+ IFAM + Var. 1*	*+ IFAM + Var. 2*	*+ IFAM + Var. 3*	*+ IFAM + Var. 4*	*+ IFAM + Var. 4 + IFB3*
400	65 808 138	2 281 978	513 872	457 191	2 322 110	149 632	26 652
600	1 450 956	227 602	18 664	60 952	81 055	54 008	20 573
1 000	716 158	134 744	15 072	18 100	22 526	51 400	17 804
4 000	716 158	134 744	15 072	18 100	22 526	51 400	17 135

Computational effort to 99% success for the enumeration algorithm MEMORIZING-RANDOM-TREE-SEARCH of Chapter 5 at various "challenge levels" on the Artificial Ant on the Santa Fe Trail problem. "IFAM", "Var. 1", and "IFB3" represent different additional functions added to the problem function set. The challenge levels are discriminated by the maximum number of time steps. A detailed description and a derivation of the additional subroutines are presented in Chapter 5.

Figure 6

Time Steps	*Normal*	*+ IFAM*	*+ IFAM + Var. 1*	*+ IFAM + Var. 2*	*+ IFAM + Var. 3*	*+ IFAM + Var. 4*	*+ IFAM + Var. 4 + IFB3*
400	1.00	28.84	128.06	143.94	28.34	439.80	2 469.15
600	1.00	6.37	77.74	23.80	17.90	26.87	70.53
1 000	1.00	5.31	47.52	39.57	31.79	13.93	40.22
4 000	1.00	5.31	47.52	39.57	31.79	13.93	41.79

Computational effort to 99% success for MEMORIZING-RANDOM-TREE-SEARCH on the Santa Fe trail problem for various subroutine sets relative to the effort of the standard subroutine set. Larger numbers represent a savings in effort. "IFAM", "Var. 1", and "IFB3" represent different additional functions added to the problem function set, and are described in Chapter 5. The experimental conditions are sorted by number of time steps available to complete the problem, and so increase in difficulty as the number of time steps goes down. Notice that for the more challenging problems, the relative speedups achieved improve dramatically.

The performance gains indicated in Fig. 6 are impressive indeed. A factor of 70 speedup is achieved for the three-subroutine function set versus the reference function set for the standard 600 time step problem. However, a factor of 2 400 is seen for the hardest problem considered, the 400-time step problem. Adding this dimension of problem difficulty thus expands our understanding of the nature of task difficulty; there clearly is an interaction between problem difficulty and relative improvement achievable. This immediately suggests ways of improving genetic programming, such as changing the difficulties of the tasks encountered; progressively presenting more challenging problems; and restarting runs with subroutines that were successful on simpler versions of the tasks.

We should point out that the subroutines derived by SYSTEMATIC-SUBROUTINE-GENERALIZATION in Chapter 5 were discovered by analysis of the 600-time step problem. If we were to analyse the 1 000 or 400-time step problem, this algorithm may well arrive at a different set of subroutines. This is, however, a modest caveat which should not interfere with the impressive result. The fact that subroutines derived for a specific problem dramaticallly improve the performance of related problems should be all the more impressive when we consider that they were not derived in this context and for this purpose!

Scientist Algorithm

To engage these problems, we posit an adaptive algorithm that we call the "Scientist Algorithm". This algorithm is basically a wrapper around genetic programming. A conventional genetic programming algorithm is used as a subroutine to make sustained progress on a problem. This scientist algorithm differs somewhat from the adaptive algorithm used in the "Robot Scientist" by King et al. [King 2001, Bryant 2001], but borrows from it as well. King et al. used an approximation algorithm to determine which experiments to perform and in which sequence to elucidate the purpose of several opening reading frames in the aromatic amino acid synthesis pathway in *S. cerevisiae* [King 2001]. This differs from the more general problem posed here, in that there reliable estimates can be made of the cost of performing an experiment. Additionally, the nature of the experiments to be performed therein is uniform: all experiments are of the same kind, namely incubation of knockout strains of the yeast to test metabolite utilization. The formulation of a cost metric to guide the search process in the present work along the lines of King's Active Selection of Experiments (ASE) model would be useful in principle, but difficult to achieve in practice. The success or failure of different pathways in the GP scientist algorithm is completely unknown at the time of experimentation: total success is often but a single experiment away. This differs from the situation in [Bryant 2001], where filling out a network of related information in a nearly-optimal way is the goal.

In an attempt to mimic human reasoning and problem-solving, we present SCIENTIST-ALGORITHM, an algorithm that can take different functional elements or "tools" as pieces. These "tools" can then be tried out as to their effectiveness at solving a problem at hand – with the aim of being able to make progess in an adaptive way on a challenging problem. Each tool observes a rigid upper limit on the amount of work that it will perform; thus, we can operate in an efficient manner on the problem at hand. An example of such a tool would be a lightly modified version of the SYSTEMATIC-SUBROUTINE-GENERALIZATION algorithm of Chapter 5, which observes a hard limit in terms of the number of evaluations performed and which returns a set of subroutines that are likely to be useful. More examples are provided in Chapters 6 and 7. SCIENTIST-ALGORITHM can be used as a kind of programmatic scaffold on which the "good tricks" of Fig. 2 can be hung. SCIENTIST-ALGORITHM also uses a few external algorithms that (a) select which tool from the toolkit to use and (b) choose how to allocate resources to tools. The general parameters of an appropriate algorithm for choosing how to allocate resources to tools will be described briefly below, and in more detail in Chapter 6. Ideally, some Bayesian cost-estimation algorithm would be deployed at the highest level, as ASE-Prolog in [Bryant 2001]; however, determining valid cost and benefit estimates for the various hypotheses under test in the scientist algorithm is a difficult problem beyond the remit of this thesis.

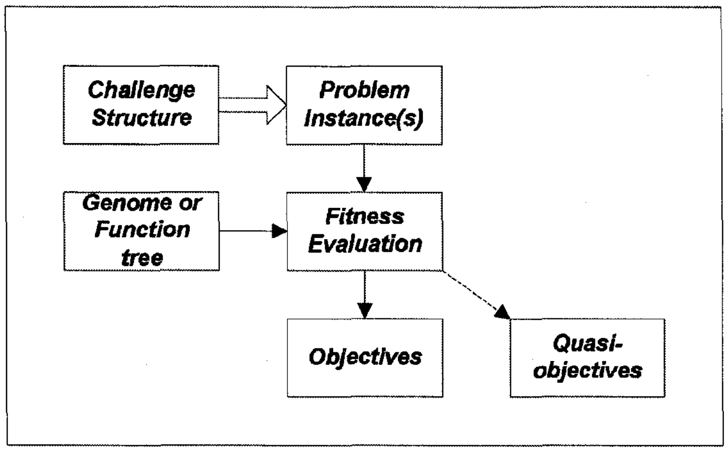
Algorithm 2: SCIENTIST-ALGORITHM

Input: an instance-generating function *I*, which takes an environment structure *E* representing the "environment" of a problem (precisely defined in the main text), and a challenge structure *C* describing the desired difficulty level of a candidate problem instance, and answers an appropriate instance $I(C;E)$ of the target problem;
an oracle *O*, which takes one or more function trees *t* using zero or more subroutines; a subroutine set *S* describing legal subroutines for *t*; an environment structure *E* representing the "environment" of a problem; and a problem instance *I*, which answers a fitness structure $O(t,I;E,S)$ containing at least a simple scalar *fitness*. *fitness* has the property that 0 is optimal, and larger numbers are worse;
a set of modules *K* (the "toolkit") where each element of *K* takes a maximum number of fitness evaluations to perform and that answers the number of fitness evaluations actually performed while being able to modify, *en passant*, any of the subroutine set *S*, a set of valuable trees *T*, and/or the choice of challenge structures *C* by performing experiments using *I* and *O*; and, a set of "built-in" function terminals *F*, which operate in the fitness space scored by *O*

Output: a valid tree *t* that solves the problem and a set of subroutines *S* which have been used to improve performance

```
$S \leftarrow \{\}$
$evals \leftarrow 0$
do
    $n \leftarrow n+1$
    $k \leftarrow$ CHOOSE-TOOL $K, F \cup S, I, O$
    $limit \leftarrow$ CHOOSE-MAX-EVALS $evals, F \cup S, I, O$
    $evals \leftarrow evals + k(limit, T, F, S, I, O)$
    $f_{best} \leftarrow$ BEST-FITNESS $T$
    while $f_{best} > 0$
return $[t, S]$
```

Figure 7



Inputs and outputs to a fitness evaluation for the SCIENTIST-ALGORITHM. Quasi-objectives are auxiliary objectives that, while not impacting correctness, are nonetheless desirable. The production of a problem instance from a challenge structure is indicated. The box labelled "Objectives" takes the place of the traditional fitness function, but in a fine-grained manner.

Intermediate Task Progress

Intermediate rewards provide a powerful mechanism for ascertaining progress on a task. The fitness function central to evolutionary computing and black-box optimization is the normal measure of progress towards a task. However, if satisfaction of a hard problem is the goal, then we will not know of any success until the problem is itself solved - too late to be of any use! We therefore propose that it will be valuable to come up with intermediate rewards - hints that provide an indication that the scientist algorithm is on a promising path. One concrete example of an automatic way to engage in this "hinting" is given in the SYSTEMATIC-SUBROUTINE-GENERALIZATION algorithm of Chapter 5. There, we consider the relative performance with different subroutine sets on the complete set of small trees, until a threshold number of evaluations is reached. Better relative performance on these small trees was used as a proxy for ultimate probability of success.

It would seem that the scientist algorithm will have a better chance of success, in general, when tools appropriate to the task are employed. One source of information using intermediate rewards might be found when using scalable problems. Easy instances might be readily solved by several techniques, with the relative effort used by them serving as a measure of the value in employing tool A over tool B. This implies that problem instances should be controllable, in part, by the scientist algorithm. It is for this reason that we explicitly break out the instance-generation algorithm *I* in SCIENTIST-ALGORITHM. This enables individual tools to choose how difficult the problem instances that they require should be.

Another important tool would be to expand the amount of information available to the scientist algorithm. The problem with the traditional scalar fitness function is that it loses a great deal of information in compiling the fitness value for a problem. The idea here is that more detailed "readouts" from a fitness evaluation *may* be useful in discriminating between potentially productive pathways or tool sets. This is the topic of the next section.

Fitness Values and Fitness Records

Fitness function selection is a bit of a "black magic" art. In some cases, such as symbolic regression, an obvious fitness function can be defined - namely the sum of the mean squared errors between the candidate function and the target function on a fixed random sample of points. We will return to the sample of points in the "Environment and the Choice of Test Cases" section below. In other cases, the choice of an effective fitness function is non-obvious. In practice, the fitness function itself goes through a sort of iterative development process. The typical stages of fitness function development are as follows:

- an obvious fitness function f_0 is manufactured based on problem characteristics and some intuitive notion of what a reasonable algorithmic definition of "better" would be for the task;
- this fitness function is used to evolve some trees using genetic programming;
- performance is generally observed to be poor; typically, evolved genetic programs exhibit a kind of cleverness, but manage to optimize the fitness function without solving the problem as desired;
- the fitness function is modified into a new fitness function, f_i, to improve performance and/or prevent trivial solutions from dominating the population;
- ...and the process repeats from the second step above.

Viewed from afar, this process looks suspiciously like a co-evolutionary one. A human programmer is incrementally adapting a fitness function as a deeper understanding of the problem and how EC makes progress on the problem develops. One approach to ameliorating this problem would be to construct an evolutionary computation system with the ability to identify and resolve its own issues. A second is to throw up one's hands and be aware that problem specification necessarily involves an incremental process of refinement. A successful fitness function for evolutionary computation will shape the fitness landscape so that easy moves in genotype space result in steady and incremental benefits towards problem solution [Langdon 1998]. It is this reshaping of the fitness landscape that we refer to when we talk about "getting the downhill direction right." Problem specification, on the other hand, is beyond the purview of this thesis, since we have defined SCIENTIST-ALGORITHM to work on a fixed oracle O to divine how effective a particular candidate solution is. We are sensitive however, to the iterative approach to problem specification, and suppose that this approach will likely remain the standard procedure for a long time to come.

Most fitness functions throw away information. Consider a symbolic regression problem where the fitness function is given by $F(f) = -\sum_{i=1}^{n}\left(f(x_i) - y_i\right)^2$. The actual fitness information that is generated for this problem are the actual measured errors $f(x_i) - y_i$, which clearly have more information content than the single scalar sum does. Unfortunately for us, working directly on these measured errors is considerably more difficult than working on a simple scalar. Multi-objective problems in general are more difficult to optimize than singly-objective problems. The nature and potential benefits of multiobjectivity will be explored more in Chapter 7.

We can easily come up with additional "quasi-objectives" that might be optimized at the same time as the primary fitness function. For instance, in addition to problem correctness, we might want to prefer programs that are parsimonious; that are correct; that do not perform illegal operations; that do not access memory outside of a legal range; and that are speed-efficient. Quasi-objectives can of course be treated as independent objectives in a multi-objective optimization. However, this problem differs from traditional multi-objective optimization since many or all of the variables can be completely optimized at once. That is, the Pareto front among the quasi-objectives is partially or completely degenerate. Therefore, another productive path is to treat some objectives as auxiliary objectives and to use them to break ties among otherwise equal candidates. We will demonstrate the effectiveness of this technique in Chapter 7 when we look at optimizing for parsimony when solving successively more challenging problems.

The benefit of breaking up fitness-related observables into their component fields is that it becomes possible for an algorithm to make progress when stymied on a particular desirable direction. Consider the case of the artificial ant on the Santa Fe trail. We can see multiple levels of detail into which we might break down the fitness function. The standard Santa Fe trail problem uses the sum of all the food acquired as a fitness measure, with a possible integral score from 0, which states that all the food is eaten by the ant, to 89, where none of the food is eaten. A second level of detail would be to answer a bit-vector which, for each piece of food, tracks whether or not it was eaten. A third level of detail would be to track the entire path traversed by an ant during its journey. This third level offers the possibility of defining some auxiliary objectives, such as:

- the number of non-food containing spots that were traversed during the walk,
- the number of movement operations performed during the walk, and,
- the proportion of the board explored during the walk.

Any of these would count as auxiliary objectives for the Santa Fe ant problem. Would they improve performance? We cannot say – but in some cases, partial solutions act as concrete progress even if not measured directly by the fitness function.

Environment and the Choice of Test Cases

We will develop a practical technique to progress beyond the per-problem tuning of genetic programming to run experiments. This is motivated by our belief that GP, while a valuable source of innovation, lacks the ability to checkpoint good results and make progress in a systematic manner.

We will present a preliminary model of how genetic programming makes progress on a problem that emphasizes the role of selecting successful substitutions of modest complexity. The key ideas of this model are first, that the larger the population used in genetic programming, the more complex are the single substitutions that can be found in a single generation. Second, evolution progresses in a stepwise manner, where the best individuals in each generation represent potentially new high-water marks for the genetic programming system. This naturally limits the number of successive refinements that a GP system is capable of making to G, the number of generations performed. Third, without elitism [Koza 1992c], a GP system might lose productive solutions on successive generations.

In light of this model, there are certain methods of productive problem solving that are not followed by genetic programming. It seems to us productive to perform small experiments using genetic programming, while having an overarching "scientist" program watch the results and direct the GP system about which experiments to run next. In this model, the "scientist" will augment traditional GP by adding the possibility of seeding an initial GP population with key individuals to preferentially explore certain areas of the search space. Essentially, GP is used as a subroutine of the scientist program, whose job it is to make progress on solving a particular problem or family of problems.

This is to be contrasted with published meta-GP techniques that run a GA at a top level that tunes the parameters of a GP technique running repeatedly as a subroutine [Schmidhuber 2004]. The scientist program will choose the parameter settings, this much the two techniques have in common. However, the scientist technique will also actively experiment, by trying different seed populations, function sets and even problem dimensions. The results of these experiments will be systematically recorded and used to inform future experiments. This represents a considerable departure from the state of the art, and is the primary contribution of this thesis. A small example of this algorithm can be seen in the algorithm EVOLVE-TINY-TREES, written up in Chapter 7.

Once we have defined the scientist program, a number of other genetic manipulations become feasible. For instance, it is possible to create, offer and test new subroutines or idioms to GP, in effect synthesizing new terminals and functions for improved performance from successful code. Another improvement that we can offer, beyond traditional GP and related systems, is that we can freeze successful individuals and try directed mutations upon these individuals to refine their abilities, optimize their performance, or create new subroutines from their genetic material to generalize their successes. This can be viewed as an extreme version of elitism, followed by the creation of a new deme with the hero immigrated in the first generation [Gilmour 1939, Wineberg 2000]. We can do such tricks as selective breeding and direct genetic manipulation, to generally reproduce some of the useful tricks that people have used in the biological world. We will also be able to do things like dynamically adapt the selection function in response to a perfectly successful individual, to optimize its size, its generalizability, or other desired traits. We will show that some of the tactics that human computer scientists use to make progress on an algorithmic problem can be used automatically, absent human intervention. For instance, we can evolve a program that sorts a vector of data by first considering a base case and evolving a solution in that environment, inducing the solution into a novel subroutine, and then solve the general sorting problem by using the conditional swap subroutine in a systematic way to discover either insertion sort or bubble sort. Furthermore, we will be able to detect when a particular tactic is not beneficial and respond appropriately to continue to make progress on the problem by a different tactic.

Of course, the simple answer to the advantage of the scientist algorithm is that it enables successful solution of problems much more efficiently than traditional GP does. In practice, we can reasonably expect to see orders-of-magnitude speedups for some sorts of problems over what the best GP systems can currently do.

Before we return to the "meat" of this thesis, however, we should discuss how to compare incremental stochastic algorithms against one another fairly. Hypothesis testing is central to the scientific method, and the techniques used to evaluate evolutionary computation methods have some strengths and some weaknesses. In the next two chapters, we will discuss and evaluate the measures which are currently used to compare evolutionary computation systems. We will then propose a new test, the *y*-test, which will prove useful in performing such comparisons.

Chapter 2: Limits on Genetic Programming, a Problem Taxonomy, and Scalable Genetic Programming

This page is intentionally left blank.

Chapter 3: Which Measure to Use When Comparing Stochastic Algorithms

One problem in performance analysis of evolutionary computation systems is that there is a diversity of measures in use for determining when one evolutionary computation (EC) system is to be preferred over another. Methods in common use, as of 2006, include the following five techniques: mean best fitness (MBF), success probability (SP), computational effort (CE), average evaluations to solution (AES), and mean fitness of population (MFP). We have generated a great deal of data for the artificial ant on the Santa Fe trail problem described in Chapter 1. We will use these data to illustrate the problems that can arise when making comparisons between these different techniques. We will also define new versions of two of the statistics which are better than their conventional forms. We begin by discussing the meaning of these five different techniques and their tradeoffs, which are shown in Fig. 1.

Figure 1

Statistic	*Analyzes only best*	*Efficiency measure*	*Work-balanced comparisons*	*Matches best against best*
Mean best fitness	yes	no	parameter choice	no
Success probability	yes	no	parameter choice	no
Computational effort	yes	balanced	always	across G
Average evaluations to solution	yes	yes	parameter choice	no
Mean fitness of population	no	no	parameter choice	no

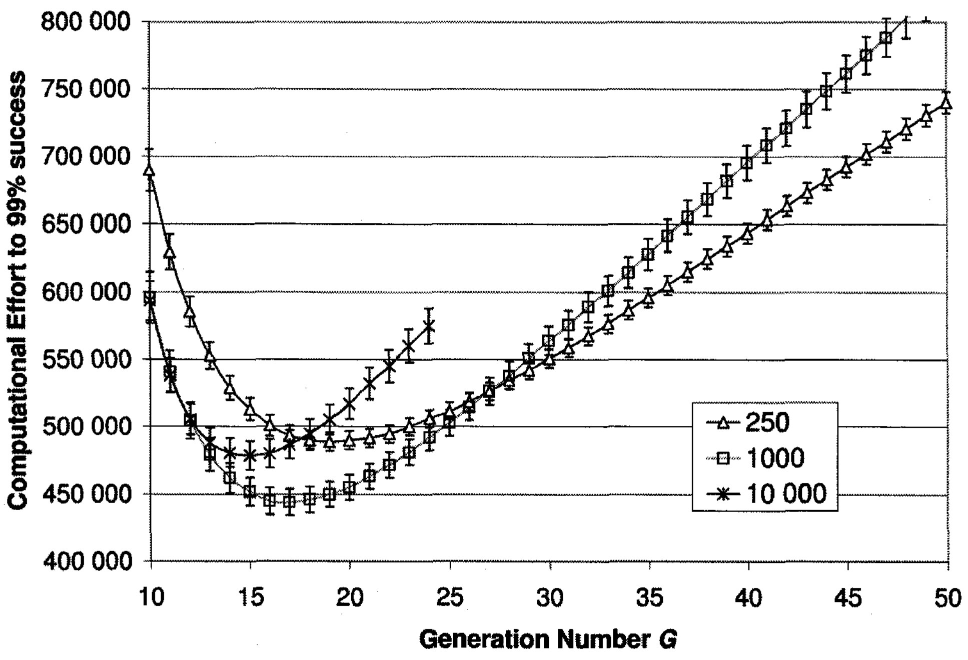
This figure shows the most common methods of comparing evolutionary computation systems in common use. For each comparison statistic, we indicate whether it confounds the best-of-population performance with population performance; whether it is intended as an efficiency measure; whether they allow comparisons with differing numbers of performance evaluations; and whether they allow matching the best parameter setting against the best parameter setting across treatments.

Computing the mean best fitness normally involves comparing the mean of several runs of one process with the mean of several runs of another, or of several other processes. As with all stochastic methods, the success rate or quality of solutions should improve with the amount of work performed. More work should not generally result in poorer performance! To make mean best fitness comparisons fair, the number of fitness evaluations is usually fixed across different treatments, so that the amount of work performed is equal across all compared treatments. For instance, in [Whitley 2006], Whitley et al. ran each EC system that they investigated for $100\,000$ fitness evaluations. We will demonstrate some of the biases and inaccurate assumptions that imperil the conclusions of those using inappropriate statistics.

Why Restarts Must Be Considered When Evaluating EC Performance

The concern with using mean best fitness while keeping the number of evaluations constant is that the performance of EC systems is not optimized simultaneously for the parameter settings chosen. For instance, we have shown in Fig. 2 the computational effort for three different population sizes for the Santa Fe trail problem. We can see that with $M = 10\,000$, genetic programming performs 10% worse than at $M = 1000$, even with the best possible generation number for each setting. Furthermore, the optimal settings for generation number are different for each series, moving from $G = 17$ for the $M = 1000$ treatment to $G = 15$ for the $M = 10\,000$ treatment.

Figure 2



Estimated computational effort to 99% success at each generation for the artificial ant on the Santa Fe trail problem as a function of population size. 12 280 runs of the M = 10 000 series, 50 010 runs of the M = 1000, and 430 625 runs of the M = 250 series were performed. 95% confidence intervals are indicated on the graph; they are independent across series but not along each series.

We use $M = 10\,000$ since the confidence intervals are very narrow for this treatment. Since a statistical test does not know the difference between different parameter settings and different performance methods, we can treat these two different parameter settings as two different EC techniques. We can then ask how a work-balanced comparison of mean best fitness between these two techniques behaves. We could conduct the comparison in one of three ways: run the $M = 1000$ treatment at its optimal generation number, run the $M = 10\,000$ treatment at its optimal generation number, or pick an intermediate generation number not optimized for either. One complication presents itself immediately: the performance of the $M = 1000$ treatment is optimized at $G = 15$, which would give $G = 1.5$ for the $M = 10\,000$ treatment. To avoid having to cope with fractional generations, we will compare the two data sets in this case at $G = 20$ for the run with fewer individuals. Thus adapted, the three possibilities are listed in Fig. 3.

Figure 3

M_1	G_1	M_2	G_2	*Evaluations*
1 000	20	10 000	2	20 000
1 000	150	10 000	15	150 000
1 000	50	10 000	5	50 000

Three different synthetic treatments illustrate the importance of using restarts when comparing two different EC techniques. The first comparison is nearly optimal for the smaller population size; the second is optimal for the larger population size; and the third is intermediate for both. In reality, $M = 1000$ is the superior choice of population size for this problem.

We can now show what happes when we compare both success probability and mean best fitness on the same runs. Using mean best fitness as our figure of merit, the fitness scores and 95% confidence intervals are indicated in Fig. 4. In each case, we test the data against one another at the listed number of fitness evaluations, and are comparing single runs against one another in pairs.

Figure 4

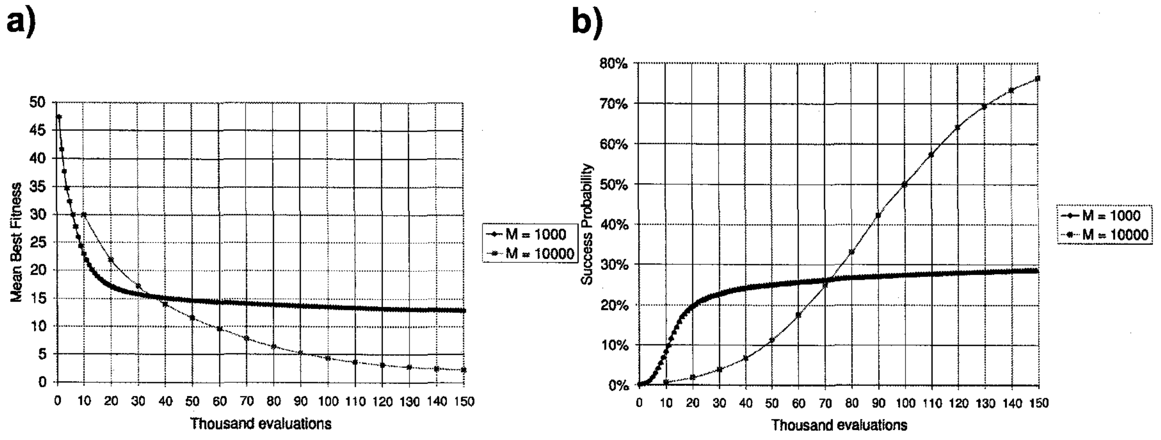
					M = 1 000		*M = 10 000*				
M_1	G_1	M_2	G_2	*Evals.*	*MBF*	*95% CI*	*MBF*	*95% CI*	*Best*	*Significance*	*d'*
1 000	20	10 000	2	20 000	17.16	0.23	21.82	0.22	*M = 1000*	$p < 10^{-175}$	0.28
1 000	150	10 000	15	150 000	12.94	0.21	2.24	0.09	*M = 10000*	$p < 10^{-1512}$	0.88
1 000	50	10 000	5	50 000	14.71	0.22	11.61	0.19	*M = 10000*	$p < 10^{-96}$	0.20

The three treatments listed in Fig. 3, compared on mean best fitness after performing the listed number of evaluations. The column labelled "MBF" is the mean best fitness of the run, and the column labelled "95% CI" is the width of 95% confidence intervals around each mean. The column labelled "Best" gives the decision of which treatment performed better, and the column labelled "Significance" gives the p-value corresponding to a t-test between each set of data. The column labelled d' as Cohen's d', which gives an indication of the size of a significant difference. A d' value of 0.25 is considered a small difference; 0.75 is a large difference. In reality, $M = 1000$ is the best choice of population size for this problem.

Considering Fig. 4, we can see that the mean best fitness statistic gives inconsistent results, even *after* correcting for number of evaluations. In Fig. 4, we have performed many runs to get highly significant results – therefore none of these data are spurious. Cohen's d' statistic [Cohen 1988] gives some indication of the difference between the two population means, independent of the number of runs performed. As we can see, while the differences between the groups are small, they are not tiny, which indicates that important differences exist. This is a very disconcerting finding for those using mean best fitness to compare treatments!

We can see an explanation for these data by considering Figs. 5, in which we show both the mean best fitness and probability of success as a function of generation number for each data set.

Figure 5



Mean best fitness and success probability as a function of the number of evaluations performed for $M = 1000$ and $M = 10\,000$. To facilitate fair comparisons, the data from the larger population is plotted every 10th generation. Each point has 95% confidence intervals indicated through error bars; these error bars are on the order of the size of markers and so are hard to see. We can see the very slow progress on the $M = 1000$ runs after about 40 generations on both panes. The $M = 10\,000$ runs do not saturate in this way over the 15 generations shown here.

From an inspection of Fig. 5, we can see that both the mean best fitness and the success probability for $M = 1000$ saturate, as many runs do not make much progress after about 40 runs. This explains the central importance of restarts to evolutionary computation: restarts allow poor runs to be truncated early. This allows work to be allocated where it will have the most effect. We have shown that mean best fitness gives inconsistent results in this case. Success probability does not do much better, as inspection of Fig. 5b shows. We can see that $M = 1000$ dominates using the constant work paradigm if less than 70 000 evaluations are performed; $M = 10\,000$ dominates thereafter. Of course, this is an unfair comparison: a fair comparison would involve comparing the best parameter settings – in this case, generation number – for $M = 1000$ against the best parameter settings for $M = 10\,000$. Suppose that we know that the best performance of $M = 1000$ happens at generation $G = 17$, and that we always want to compare against $M = 10\,000$. One way of performing an honestly fair comparison would be to compute the results of mean best fitness and success probability of 10 runs of $M = 1000$ against 1 run of $M = 10\,000$. This differs in what we tried above, in that we compared a *single* run of 150 generations of $M = 1000$ against a single run of 15 generations of $M = 10\,000$. To do this, we will need an expression for the mean best fitness of *k* runs of a process.

Computing Mean Best Fitness with Restarts

To perform a fair comparison between groups with different efficiencies, we will need to adapt mean best fitness to account for multiple runs, as we did for the median with the y-test. As before, we assume that we only have the observed data to go from. To get an expression for the mean of *r* runs of the process, we will weight the observed data. One perhaps obvious way to approach this would be to use the quantile-equivalency equation which we will derive in Chapter 4. It is reprinted here as (1).

$$q_b = 1-(1-q_a)^{b/a} \tag{1}$$

We might try to resample the observed best fitness values $d_1,...,d_n$ with quantiles chosen from a uniform distribution that is biased using (1). While effective and generally reliable, resampling is a computationally expensive operation. A better approach would be to weight the data in a simple mean so that the weight on each datum corresponds to its proportion of being chosen using (1). We know that there are *n* independent data. If we take each data d_i in the original data set as being representative of the quantile range $q_a \in \left[\frac{i-1}{n}, \frac{i}{n}\right]$, we can transform this range using (1) to get the equivalent range in the transformed space, (2).

$$q_b = \left[1-(1-\frac{i-1}{n})^{b/a}, 1-(1-\frac{i}{n})^{b/a}\right] \tag{2}$$

To convert this range into a weight that we can apply, we can view the simple mean as having equal weights of the probability of choosing data from within their respective ranges. This equivalency is shown in (3).

$$\bar{d} = \int_{q=0}^{q=1} f(q)\,dq = \sum_{i=1}^{n} \frac{f(\frac{i}{n})}{n} = \sum_{i=1}^{n} \frac{d_i}{n} = \frac{1}{n}\sum_{i=1}^{n} d_i \tag{3}$$

The advantage of writing the mean in this nonstandard way is that now we can apply the weights from (2) to get a weighted mean for the best of a number of runs. (4) provides the algebra for the weighted mean.

$$\begin{aligned}\bar{d}_{weighted} &= \int_{q=0}^{q=1} f_{weighted}(q)\, dq \\ &= \sum_{i=1}^{n} d_i \left(1-\left(1-\frac{i}{n}\right)^{b/a} - \left[1-\left(1-\frac{i-1}{n}\right)^{b/a}\right]\right) \\ &= \sum_{i=1}^{n} \left(\left(1-\frac{i-1}{n}\right)^{b/a} - \left(1-\frac{i}{n}\right)^{b/a}\right) d_i\end{aligned} \qquad (4)$$

This can be viewed as a simple weighted mean, where the weights w_i are given by (5).

$$w_i = \left(1-\frac{i-1}{n}\right)^{b/a} - \left(1-\frac{i}{n}\right)^{b/a} \qquad (5)$$

We should consider carefully what the variance of this weighted data is. We can get an accurate approximation by using a simple argument. Suppose that $b/a = 2$, that is, we are looking for the mean of pairs of runs from a given distribution. A trivial method, much less sophisticated than the technique expanded upon above, would be to actually pair the data, compute the best value of each pair, and then compute the mean and variance of these best values. Precisely, we form pairs of data $\{d_{2j-1}, d_{2j}\}$, where *j* varies over $1 \ldots \frac{n}{2}$. The mean of these pairs is given by the simple mean of the minimum of each pair, which is shown in (6). Here we have invented the subscripted notation $\bar{d}_{(2)}$ to mean the mean of the minimum of pairs of samples. The estimated standard error of the mean of the pairs is given by the sample standard deviation of the pairs, (7), divided by the square root of the number of samples, here *n*/2.

$$\bar{d}_{(2)} = \frac{2}{n}\sum_{j=1}^{n/2} \min(d_{2j-1}, d_{2j}) \qquad (6)$$

$$s_{d_{(2)}} = \sqrt{\frac{\sum_{j=1}^{n/2} \min(d_{2j-1}, d_{2j})^2 - \frac{\left(\sum_{j=1}^{n/2} \min(d_{2j-1}, d_{2j})\right)^2}{n/2}}{n/2 - 1}} \tag{7}$$

There is nothing magical about which pairs to take in these expressions. That is, we can use a pairing function to choose indices to pair together and take the minimum. If we represent the indexing function using the notation $f(1,j); f(2,j)$, giving the pair $\{d_{f(1,j)}, d_{f(2,j)}\}$. We can then envision two different classes of pair allocation functions: one where we allow $f(1,j) = f(2,j)$, and another where we disallow such duplicates. In the case where we allow duplicates, we can apply a geometric argument like that given in Chapter 4, Fig. 1 to get the probability of selecting d_i. The probability of d_i winning a tournament of size *r* with replacement is given by $P(d_i \; selected) = \frac{(n-i+1)^r - (n-i)^r}{n^r}$.

We can use this result to give a closed-form expression, analogous to (6), but taken over all possible tuples of size *r*. We begin with (8), which is a generalization of (6) for an arbitrary number of runs *r*.

$$\overline{d}_{(r)} = \frac{r}{n} \sum_{j=1}^{n/r} \min(d_{f(1,j)}, \ldots, d_{f(r,j)}) \tag{8}$$

We know from the above that we can replace the minimum in (8) using $P(d_i \; selected) = \frac{(n-i+1)^r - (n-i)^r}{n^r}$, giving (9a).

$$\overline{d}_{(r)} = \frac{r}{n} \sum_{j=1}^{n/r} \sum_{i=1}^{n} \frac{(n-i+1)^r - (n-i)^r}{n^r} d_i \tag{9a}$$

The inner summation is then constant, which allows immediate rewriting and cancellation to get (9b).

$$\bar{d}_{(r)} = \sum_{i=1}^{n} \frac{(n-i+1)^r - (n-i)^r}{n^r} d_i \quad \text{(9b)}$$

We can collect and factor (9b) into the final form of (9c), which is then identical (!) to (4), above. The technique expressed in (4) is therefore identical to taking the average of (8) over all possible set allocation functions *f*.

$$\begin{aligned} \bar{d}_{(r)} &= \sum_{i=1}^{n} \frac{(n-i+1)^r - (n-i)^r}{n^r} d_i \\ &= \sum_{i=1}^{n} \left(\left(\frac{n-i+1}{n} \right)^r - \left(\frac{n-i}{n} \right)^r \right) d_i \\ &= \sum_{i=1}^{n} \left(\left(1 - \frac{i-1}{n} \right)^r - \left(1 - \frac{i}{n} \right)^r \right) d_i \end{aligned} \quad \text{(9c)}$$

This identity can now be used to correctly derive the estimated standard error of the mean of this estimate. As before, we begin with (10), a generalization of (7) for arbitrary *r*. We then substitute our probabilistic model in place of the minimum in (10) to get (11a). Some simplifications give (11b), which we can then rewrite as (11c).

$$s_{d_{(r)}} = \sqrt{\frac{\sum_{j=1}^{n/r} \min(d_{f(1,j)},\ldots,d_{f(r,j)})^2 - \frac{\left(\sum_{j=1}^{n/r} \min(d_{f(1,j)},\ldots,d_{f(r,j)}) \right)^2}{n/r}}{n/r - 1}} \quad \text{(10)}$$

$$s_{d_{(r)}} = \sqrt{\frac{\sum_{j=1}^{n/r} \left(\sum_{i=1}^{n} \frac{(n-i+1)^r - (n-i)^r}{n^r} d_i^2 \right) - \frac{\left(\sum_{j=1}^{n/r} \sum_{i=1}^{n} \frac{(n-i+1)^r - (n-i)^r}{n^r} d_i \right)^2}{n/r}}{n/r - 1}} \quad \text{(11a)}$$

$$s_{d_{(r)}} = \sqrt{\frac{n/r\left(\sum_{i=1}^{n}\frac{(n-i+1)^r-(n-i)^r}{n^r}d_i^2\right)-\frac{\left(n/r\sum_{i=1}^{n}\frac{(n-i+1)^r-(n-i)^r}{n^r}d_i\right)^2}{n/r}}{n/r-1}}$$

$$= \sqrt{\frac{n/r}{n/r-1}\left(\left(\sum_{i=1}^{n}\frac{(n-i+1)^r-(n-i)^r}{n^r}d_i^2\right)-\left(\sum_{i=1}^{n}\frac{(n-i+1)^r-(n-i)^r}{n^r}d_i\right)^2\right)} \quad (11b)$$

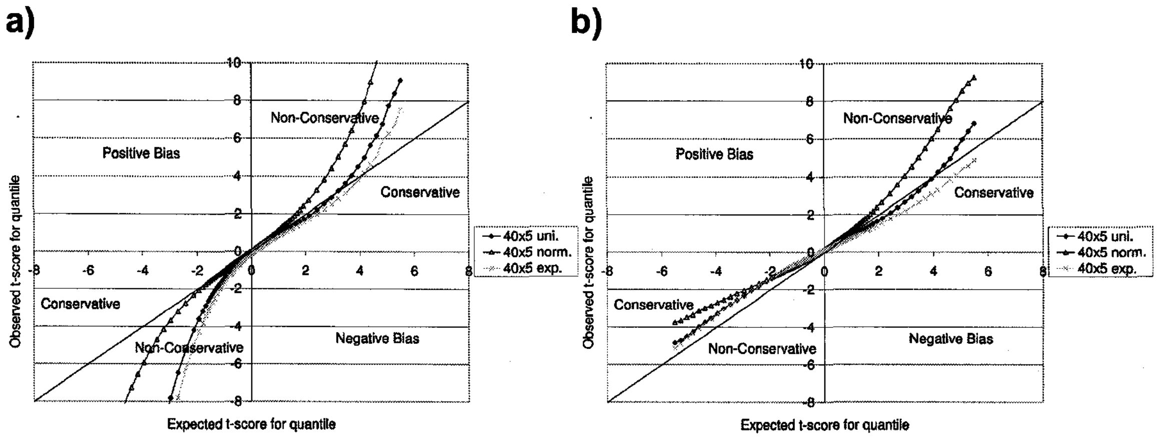
$$s_{d_{(r)}} = \sqrt{\frac{n/r}{n/r-1}}\sqrt{\sum_{i=1}^{n}\left(\left(1-\frac{i-1}{n}\right)^r-\left(1-\frac{i}{n}\right)^r\right)d_i^2-\left(\sum_{i=1}^{n}\left(\left(1-\frac{i-1}{n}\right)^r-\left(1-\frac{i}{n}\right)^r\right)d_i\right)^2} \quad (11c)$$

Using the effective sample size of *n*/*r* in (11c) gives us an expression for the estimated standard deviation of the mean of *r* runs, (12).

$$s_{\bar{d}_{(r)}} = \sqrt{\frac{\sum_{i=1}^{n}\left(\left(1-\frac{i-1}{n}\right)^r-\left(1-\frac{i}{n}\right)^r\right)d_i^2-\left(\sum_{i=1}^{n}\left(\left(1-\frac{i-1}{n}\right)^r-\left(1-\frac{i}{n}\right)^r\right)d_i\right)^2}{n/r-1}} \quad (12)$$

With estimates for the mean and standard deviation of the mean, we can perform a *t*-test as normal. As with any new statistic, we should validate this new technique against a tried and true technique. One simple technique against which to compare would be to randomly bin generated data together in groups of *r*, and to compute the minimum of each group. This gives *n*/*r* samples, from which we can compute the mean and standard error of the mean as normal. If we intend to do significance testing with a new statistic, it would be useful if the statistic were unbiased and appropriately controlled Type-I errors. We can indicate how closely an actual distribution approaches this ideal by considering a graph of how the *t*-score computed for a quantile regresses against the *t*-score observed for the quantile. Fig. 6 shows such a graph, for a small effective sample size (8 units).

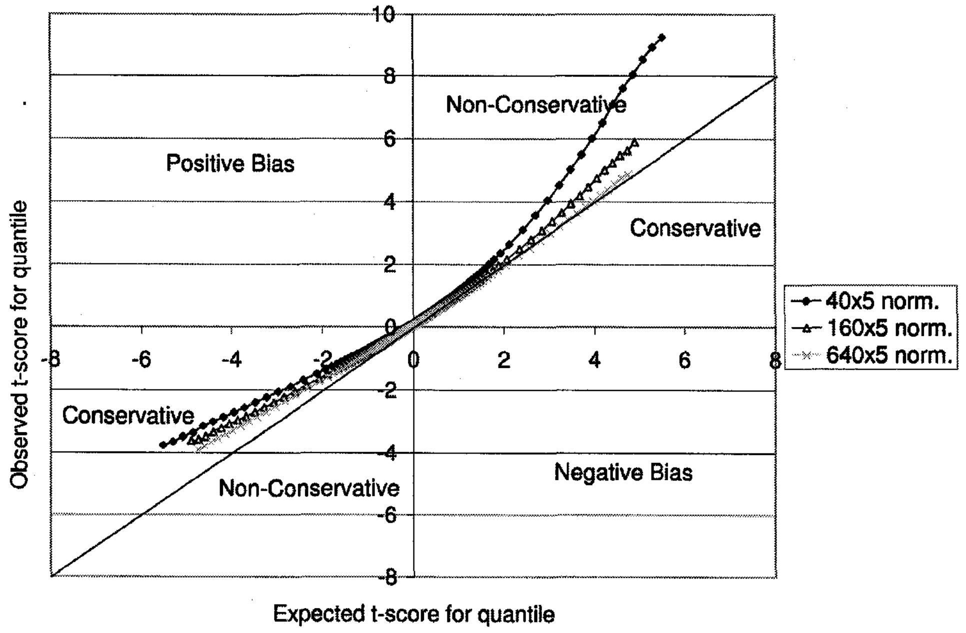
Figure 6



Graphs of expected versus observed t-scores for a range of quantiles for the mean of the best of 5 runs of three standard distributions. A perfect estimator would follow the diagonal straight line. Both figures involve taking 40 individual samples from the listed distribution, to exaggerate the significance of errors. Fig. 6a shows data obtained from grouping the 40 samples in groups of 5 at random, and taking the best; Fig. 6b uses the integral method described in the text. We have marked regions where each test is conservative (safe), non-conservative (unsafe), and biased in the positive and negative directions. The three distributions graphed are the standard normal distribution, the uniform distribution on [0, 1), and the standard exponential distribution. Each line is the result of 500 000 independent runs. The plotted points are not independent in the x-axis.

Since the effective sample size is so small, the graph deviates strongly from the ideal; however, in every case, our synthetic method is more conservative than a simple random sampling. Therefore, we can take some confidence in the use of our technique. Fig. 7 shows the expected $O\left(\sqrt{n}\right)$ convergence to the true distribution, at least in the region where our test is non-conservative. In the region where our test is conservative, our test appears to converges sub-linearly. The effect of this behaviour in practice will be shown later on in this chapter.

Figure 7



Graphs of expected versus observed *t*-scores for a range of quantiles for the mean of the best of 5 runs of the standard normal distribution. Here a series increasing in sample size is used to show the convergence properties of the technique described in the text.

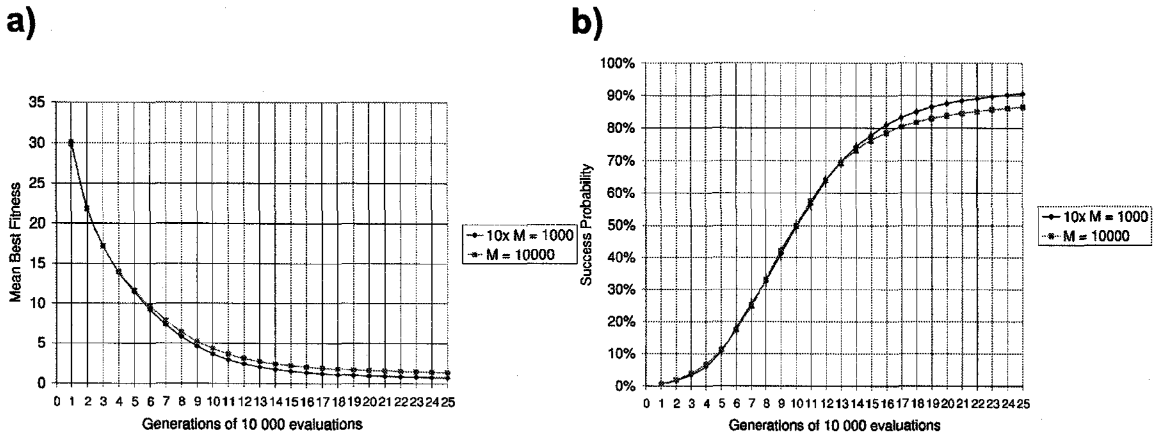
Fair Comparisons for Mean Best Fitness and Success Probability

Armed with these tools, we can perform a fair comparison between runs with different population sizes. We use (9c) and (12) in the usual way to compute and compare the mean best fitness of different runs. For success probability, we can use (13) to compute the probability of success for the best of *r* runs, since the runs are independent and identically distributed.

$$p_{(r)} = 1 - (1 - p)^r \tag{13}$$

Returning to our artificial ant on the Santa Fe trial problem, we can now compensate for varying number of evaluations to get a solution between different treatments. The first improvement we can make would be to compare the best of 10 runs of $M = 1000$ against 1 run of $M = 10\,000$. This is shown for both measures in Fig. 8.

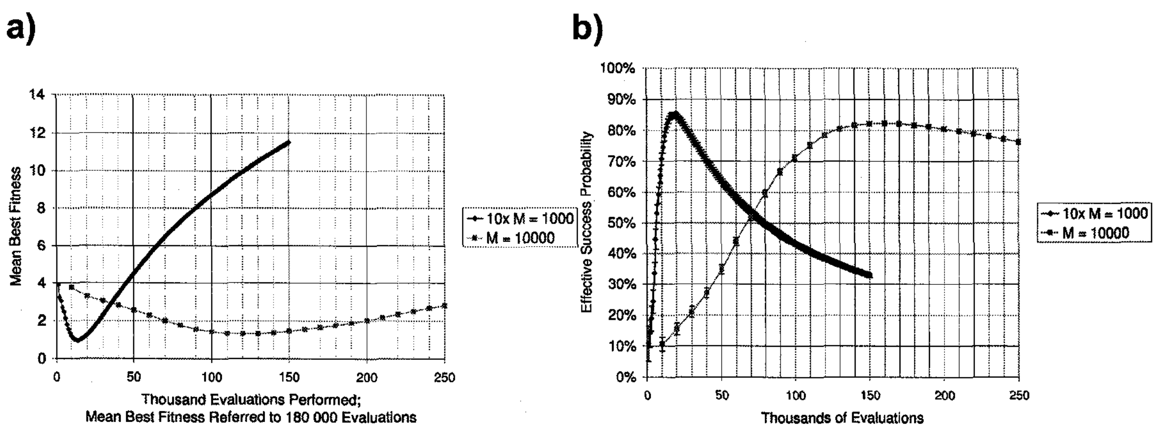
Figure 8



Mean best fitness and success probability as a function of the number of evaluations performed for the best of 10 runs at M = 1000 and 1 run of M = 10 000. Each point has 95% confidence intervals indicated through error bars; these error bars are on the order of the size of markers and so are difficult to see. These balanced data are much more closely matched, with none of the distortions of the data of Figs. 5a and 5b. M = 1000 wins out slightly at large numbers of individuals, for both mean best fitness and success probability. Notice, however, that the two graphs are not the same – the series diverge at larger evaluation counts later in Fig. 8b than in Fig. 8a.

We can see that $M = 10\,000$ does not clearly dominate anywhere, unlike in Figs. 5a and 5b. The comparison technique shown in Figs. 8a and 8b, while much improved, are not the final word. The problem is that we have no way of determining what the optimal generation number is for achieving success or giving the best performance. As before, suppose that we want to compare against strong settings for $M = 10\,000$. The appropriate way to perform this comparison is to continuously vary the number of runs to use when taking the minimum. An appropriate schedule would be $r = \frac{k}{Mg}$, where k is a constant that optimizes the larger population, and r has its usual meaning of the number of runs over which to take the best. For the moment, we ignore the fact that r is non-integral, and can be less than one. The results of using this schedule are shown in Figs. 9a and 9b.

Figure 9



Effective mean best fitness and effective success probability as a function of the number of evaluations performed, referred to a constant number of evaluations. Each point has 95% confidence intervals indicated through error bars, though they are largely invisible behind the points. The number of runs over which the given data are synthesized for each series varies in inverse proportion to the number of evaluations performed through the equation $r = \frac{180\,000}{evals}$. We can see clear optima in the success probabilities for optimizing both mean best fitness and success probability.

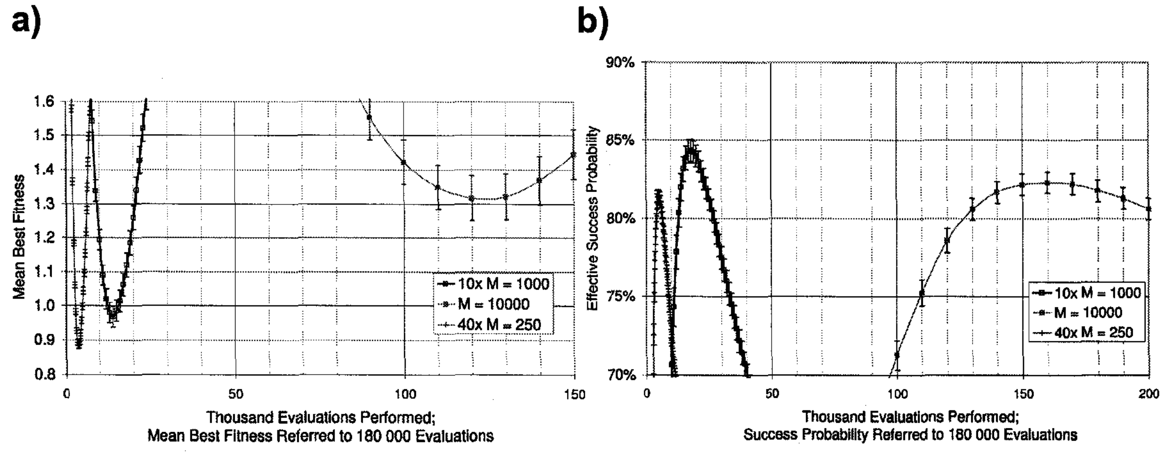
We turn now to a discussion of the relative merits of our effective mean best fitness and effective success probability. We will show that for problems which regularly achieve success, effective success probability should be used.

Deciding Between Mean Best Fitness and Success Probability

Armed with these tools, we can perform a fair comparison between runs where the work and best generation number (equivalently, number of evaluations) is unknown. There are necessarily limits on this process – for example, if we have performed only 100 runs, we could not expect to get any information on the best-of-1000 distribution. The limits are different for referred mean best fitness and success probability. For mean best fitness, we can get a zeroth order approximation by noting that the effective sample size is given by $O\left(n/k\right)$. Therefore, if we wish an estimate of effective mean best fitness accurate with a relative error of ε, we will need a few times $k\varepsilon^2$ samples. The calculation for success probability is treated carefully in the second half of Chapter 4, so we defer a detailed discussion until that point. We will only repeat the final result, namely that if we wish to compare two treatments with a relative work ratio between them of r at some specified quantile q, with two-sided probability of error α, we will require $N = \frac{(1-q)^{1/r} z^2}{1-(1-q)^{1/r}}$ samples, where z is the upper $1-\frac{\alpha}{2}$ cutoff of the standard normal distribution.

The availability of these two tools raises an interesting question, however. Are these two measures equivalent? That is, are we justified in using success probability and mean best fitness interchangeably to decide which of two treatments is superior? If so, we should prefer the test with more power; if not, we should have some other way of choosing the test for the data in question. To shine some light on this question, we repeated the analysis of Figs. 9a and 9b with an additional treatment at $M = 250$. Figs. 10a and 10b shows the results zoomed in around the optima of each treatment.

Figure 10



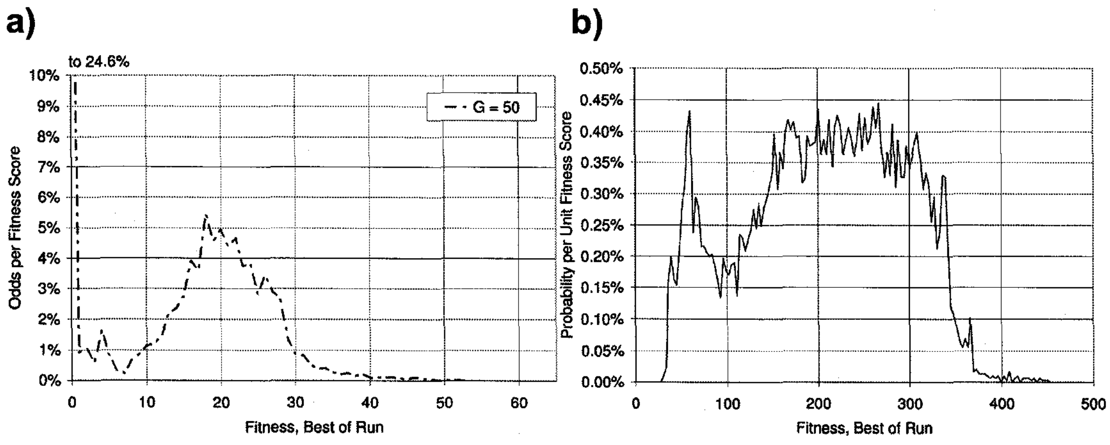
Effective mean best fitness and effective success probability as a function of the number of evaluations performed, referred to a constant number of evaluations. We have added data for $M = 250$ to the series compared. We also have rescaled the domain and range from that shown in Figs. 9a and 9b to easily discriminate between the series. 95% estimated confidence intervals are indicated, but these error bars are not independent in evaluations performed. In all, 12 280 runs of the $M = 10\,000$ treatment, 50 000 runs of the $M = 1000$ treatment and 500 000 runs of the $M = 250$ treatment were performed. On both charts, the leftmost series is the $M = 250$ series.

From careful inspection of Figs. 10a and 10b, we can see that the two measures are not measuring the same thing. The best treatment among these three for effective mean best fitness is $M = 1000$, while $M = 250$ is the best on effective success probability. Both of these distinctions are highly significant; in the case of success probability, $p < 6 \cdot 10^{-12}$ after correcting for multiple comparisons using the Bonferroni inequality [Bonferroni 1935]. So which decision method should we prefer? This is a bit of a philosophical question, but we can make an argument based on the evidence available.

Effective Mean Best Fitness versus Effective Success Probability: Which Comparison To Use?

We can discriminate between two different kinds of problem: those where "success" is the desired outcome and those where we are instead aiming for the best possible outcome. We term the former a "success-based" problem, exemplified by the familiar artificial ant on the Santa Fe trail. For a success-based problem, there is generally a "natural" definition of success. A program that eats all 89 units of food within the allotted time is said to succeed on the Santa Fe trail problem. We term the second type of problem "improvement-based". A symbolic regression problem that lacks a closed-form solution would be an excellent example: a perfect success is known to be unachievable. An improvement-based task typically has many local optima – which may have closely spaced fitness scores. For example, the aggregate fitting error for a symbolic regression problem might take on one of 2^{40} possible values near the optimum if the fitness function is stored as 80-bit IEEE double-precision numbers. There is, of course, no distinction in principle between these two kinds of problem – for an improvement-based problem, *one* of the possible fitness outcomes will be the best. Unfortunately, knowing *which* value is optimal is usually an NP-complete problem, as shown in [Polynomial 19xx] for the problem of finding the exact optimum of a set of polynomial equations. Since there are often many optima, the odds that EC can find the exact optimum are probably exponentially small for many hard problems. We can therefore some advice for which comparison to prefer. If the problem is has a natural success criterion that can readily be achieved in practice, success probability is the appropriate comparison technique. If the problem does not have a natural success criterion, then we are in a bit of a bind. When better solutions can in principle always be found, we should use effective mean best fitness as a discriminator. When it seems that the best fitness can be found reliably, there is a gray area introduced; have we found the truly best fitness, or are we getting caught up on a local optimum? We can hope that this intermediate case happens rarely in practice. We can demonstrate the typical difference between the two cases using a little example. Fig. 11a shows the distribution of best-of-run outcomes after 50 000 fitness evaluations for the Santa Fe ant problem with $M = 1000$. Fig. 11b shows the distribution for a symbolic regression problem fitting a rational function to the log density of Earth's atmosphere as a function of altitude.

Figure 11



Distribution of best fitness for a typical success-based and an improvement-based problem after 50 000 fitness evaluations. Fig. 11a scores data from 12 288 runs of the Santa Fe ant problem with $M = 1000$, and shows the histogram of best outcomes after 50 generations. Fig. 11b shows the best outcomes of 10 186 runs of a symbolic regression problem where we are fitting a rational polynomial to the logarithm of the density profile of Earth's atmosphere. For Fig. 11b, we have normalized the probability mass function to unit fitness; the bin width is 3 fitness units. Much fine structure is obscured by the binning; for instance, there is a local optimum with a probability per unit fitness of at least 300% at a fitness of 167.02.

The first thing to notice in Figs. 11a and 11b is how the distributions differ as we move towards smaller fitness values. Fig. 11a shows that there is a good chance of achieving the exact optimum, specifically a 24.6% chance. Fig. 11b behaves more like a continuum of states. In fact, among our 10 186 runs, exactly one fitness value occurred twice, and it was an example of convergent evolution to the same phenotype in two independent runs. This differs from the success probability case, where we typically have different phenotypes with the same fitness value. We can use these statistics to make an estimate of the number of fitness levels in the local optima populated by our different EC runs. We have seen 10 185 different fitness values in as many phenotypes, we can use a birthday paradox argument to estimate a lower bound for the total number of fitness levels available to successful solutions. Let us assume that there are *n* fitness levels available for EC to discover. Assume that all the fitness levels will be populated with equal probability by the EC operation; that is, the odds of observing a particular level *i* of the *n* available is simply $p(X = i) = \frac{1}{n}$. Independent runs are identically distributed, so the odds of the *k*-th consecutive draw being different from all previous draws is given by $\frac{n-k+1}{n}$, as there are $k-1$ values which will cause a collision. The probability of observing *k* values without any matches among the values is then given by $p(k\ \mathit{all\ different}) = \prod_{i=1}^{k} \frac{n-i+1}{n}$. We can then set this probability to any desired confidence interval, and solve for *k*. If we take the 10 185 unique fitness values to be distinct, $k > 17\,300\,000$ with 95% confidence. Alternately, we might count the one duplicated phenotype as a legitimate collision between fitness values. In this case, we can derive a 95% confidence interval for *k* of $[9\,310\,000, 214\,000\,000]$. Either way, we can get the sense that the number of fitness levels is many orders of magnitude higher than in the discrete case where there are 90 fitness levels. With many closely spaced fitness optima, the probability of finding the exact maximum becomes effectively zero, even using a powerful optimization technique like evolutionary computation and many independent runs. In this case, performing comparisons between techniques using success probability is well-nigh impossible – all techniques will score 0%! We therefore suggest that the best possible technique would be to use the effective mean best fitness in this case.

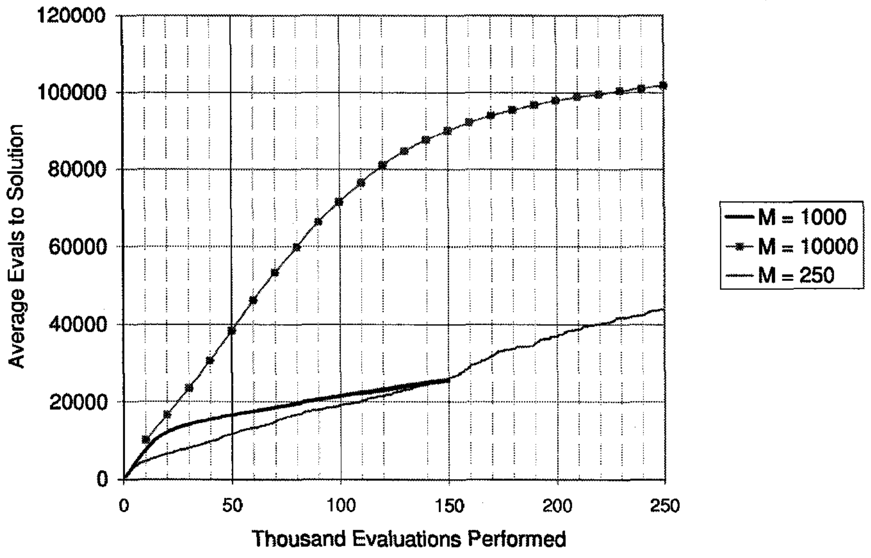
Binning continuous-valued data to turn an improvement-based problem into a success-based problem is also a dubious practice. The common practice of defining a success as fitness better than some arbitrary cutoff value does enable the use of computational effort, but at a cost. For improvement-based problems in optimization, we are normally interested in achieving the best possible outcome. Indeed, much of the benefit of using evolutionary techniques comes from the improved solutions that they provide, even though they may be slower than more aggressive techniques like iterated hill-climbing. Therefore, applying an arbitrary cutoff seems to us logically ill-founded – better to treat the data directly using the effective mean best fitness. We will show in the last section of this chapter that effective mean best fitness and effective success probability give different results on a comparison, suggesting that we may not get the correct decision if we bin.

The Problems with Average Evaluations to Success

What of the other two measures mentioned in the introduction – average evaluations to success and mean population fitness? We can determine what they are measuring by investigating how they perform in practice. In this section, we address average evaluations to success; in the next, we will take up mean population fitness.

The average evaluations to success (AES) statistic is derived by considering only those runs which result in a success. To avoid catastrophic use of computer resources, a cutoff is normally applied beyond which data are not collected. For instance, we might perform 100 000 fitness evaluations, of which 16% succeed. For these successful runs, we take the average of the number of evaluations performed and use it as a statistic. It turns out that average evaluations to success is fairly sensitive to the cutoff used, which makes it a statistic of questionable utility. In the limit, the success probability of any EC system that can generate new individuals will approach 100%; however it may take very many fitness evaluations to achieve this. There are two possibilities: if the success probability converges quickly enough to some limiting success probability, AES will be a meaningful statistic. If the success probability converges slowly, however, AES will tend to creep upwards as the cutoff generation, or equivalently the number of evaluations, goes up. We should be able to discriminate between the two cases by considering long runs with many fitness evaluations. To this end, we repeated our Santa Fe runs above, but we performed 8 000 runs with $M = 250$ out to $G = 1000$ generations. This long-range case is interesting. The success probability increases slowly from 9% at 100 generations to 10.5% at 200, 12.4% at 500 and 14.4% at 1000 generations. We can get a sense of how useful AES is by considering how it increases as we explore longer and longer runs. For instance, if AES were to increase by a factor of 50% as we go from 100 generations to 1000, then we might surmise that AES as an absolute statistic is only reliable to 50%. The results of our average evaluations to success experiments are shown in Fig. 12.

Figure 12



Average evaluations to success for three different series, graphed against number of evaluations performed. The $M = 250$ series has been run for 1000 generations. We can see that the average evaluations to solution continues to increase without bound, albeit slowly, as the number of evaluations increases. 95% confidence intervals are shown on the graph, but are not significant save for the $M = 250$ series; they range to 3500 by the last generation.

Fig. 12 shows that AES behaves a little strangely. First, AES seems to be correlated with population size. We will return to this observation later. Second, we can see that AES creeps upwards for each of our treatments. For the $M = 250$ case, 100 generations is equivalent to 25 000 evaluations. At 25 000 evaluations, Santa Fe trail has an AES of 7 500; this increases to an AES of 44 000 after 250 000 evaluations – a factor of ~7 higher! We know that the success probability increases by a factor of 1.6 in this interval, from the data quoted earlier. It would seem that for this problem, AES is not a very stable statistic over the number of evaluations performed.

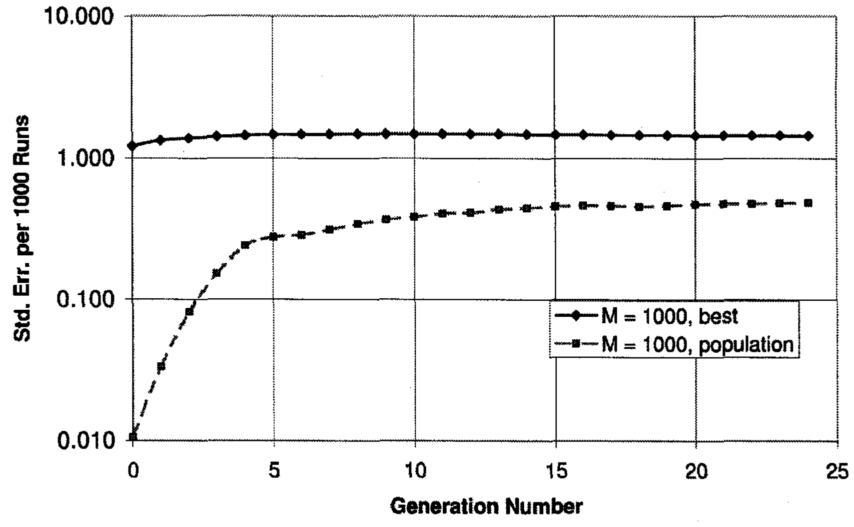
One other way to save AES as a useful statistic would be to hold the number of individuals generated constant. That is, instead of a global AES statistic, we might speak of the AES of a set of treatments given that 100 000 evaluations were performed. However, even this usage is compromised, as Fig. 12 shows. Consider the treatments measured at the vertical line at 100 000 evaluations in Fig. 12. At this point, the $M = 250$ treatment is the clear winner – that is, when successful, the number of evaluations is minimized. This is due to the fact that, of the 12.0% of the runs that succeed, most of them succeed early. More of the $M = 1000$ and $M = 10\,000$ treatments succeed – 26.9% and 57.3%, respectively, which means that it would be difficult to actually make use of the rapid success provided by the AES statistic. Another way to consider this objection is that we can always shrink AES without bound by simply choosing smaller and smaller population sizes – so long as we are willing to accept a very small probability of success in the bargain. If we wish to combine AES and success probability fairly, we must include restarts; leading us back to computational effort, or the effective success probability.

The Problems with Mean Population Fitness

Mean population fitness has one significant advantage: it converges very quickly. Fig. 13 shows the standard error of the best-of-run fitness compared to the standard error of mean population fitness. At high generation number, mean population fitness is about three times more precise than mean best fitness; at lower generation numbers, this approaches 100 times more precise. Since the number of runs required to get a given level of precision goes as the inverse square of the standard error, many researchers have adopted it as a measure. A second potential "advantage", which we feel is somewhat illusory, is that much of population genetics and GA theory refers to the behaviour of the mean fitness of the population. However, when performing optimization research to solve problems, we are usually interested in the finding the most successful individuals.

The behaviour of an auxiliary set of solutions used during the searching process is not normally of great interest.

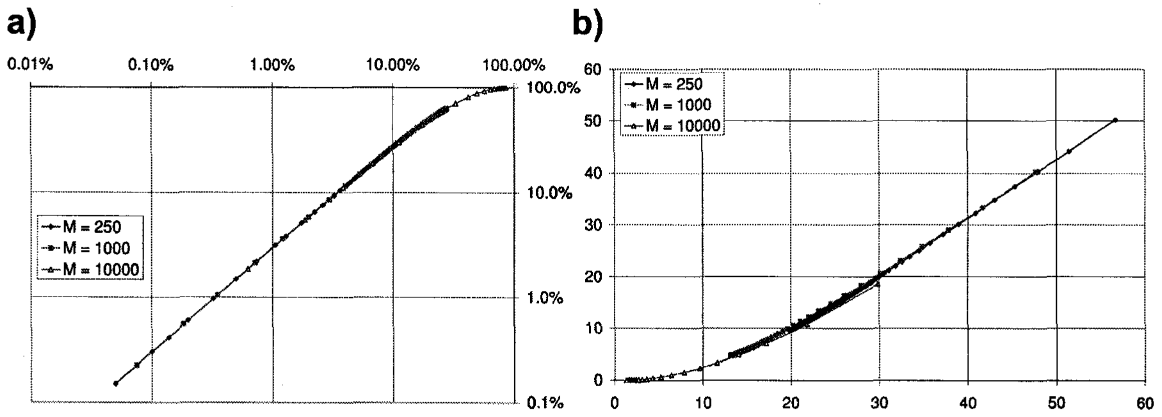
Figure 13



Behaviour of the estimated 95% standard error width for mean best fitness and mean population fitness on the Santa Fe ant problem, as referred to 1000 runs. Note the logarithmic spacing on the y-axis. The convergence of the population bounds upwards at generation 5 is due to differential performance between runs, as some runs begin to perform well and others poorly.

It would be ideal if mean fitness were a good predictor of performance of one of the two key measures, mean best fitness and success probability. If this were the case, then we could use mean population fitness, which converges quickly, as a proxy for success probability or mean best fitness, which converge slowly. Lacking an obvious theoretical tool to use for such a test, we can at least look at how well mean population fitness predicts our two fundamental measures. A useful relation of this form need not be linear, but it should be one-to-one and onto; that is, it should be a function. Figs. 14a and 14b show examples of what a good predictor between a measurable variable and a target variable looks like. Here we have graphed success probability and mean best fitness of the best-of-3 distributions versus the statistics for the best-of-1 distribution.

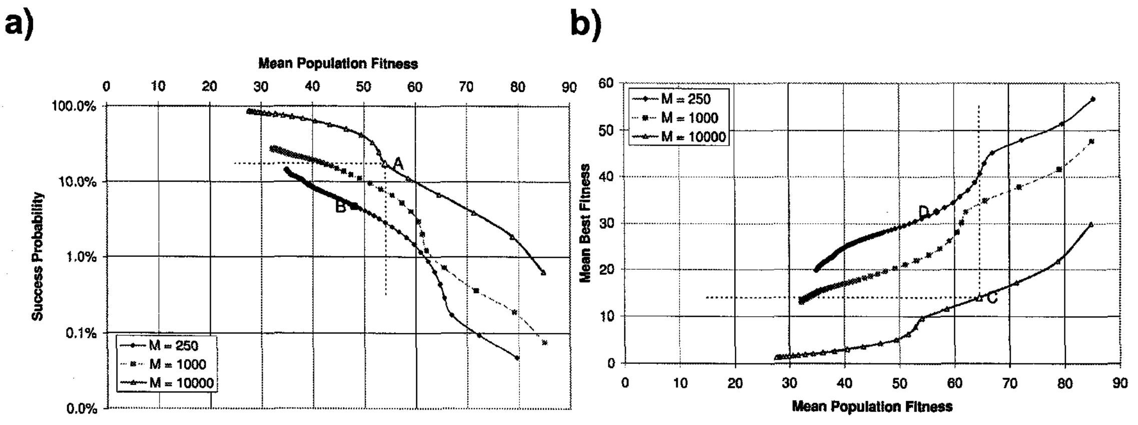
Figure 14



Examples of well-predicted relations. Fig. 14a shows the success probability of the best-of-3 runs as a function of the observed success probability for a single run. Fig. 14b shows estimated mean best fitness of the best-of-3 runs as a function of the observed mean best fitness for single runs. Both curves were computed using the algorithms presented in this chapter. The deviation from a single curve in Fig. 14b at generation 30 for the $M = 10\ 000$ series is due to estimation error in determining the best-of-3 fitness.

For the purposes of predicting performance, it is clear that the best-of-3 distributions accurately reflect the individual run distribution, provided that enough runs are performed. We should emphasize that these graphs do not involve plotting the fitness statistics and success probability of particular runs against one another, as a moment's reflection will show. Since individual runs must either succeed or fail after a given number of generations, it follows that the success probability for individual runs must either be 0 or 1. Instead, in this series of graphs, we show the mean response estimated from a large number of runs, as we vary the generation number. With that proviso, Figs. 15a and 15b show how well mean population fitness predicts success probability and mean best fitness.

Figure 15



Performance of mean population fitness as a predictor of mean best fitness and success probability for three treatments. Fig. 15a shows success probability as a function of the mean population fitness; Fig. 15b shows estimated mean best fitness as a function of mean population fitness. In the ideal case of perfect prediction, all series fall on the same curve. The points labelled A, B, C, and D are explained in the main text, as are the dotted lines radiating from A and C.

To illustrate how we can interpret Figs. 15a and 15b, we use a simple example. Suppose that we were to compare the point labelled A of the $M = 10\ 000$ series against the point labelled B in the $M = 250$ series. If we use mean best fitness, the decision is clear: A has a mean fitness of 54.2, while B has a mean fitness of 48.0, so the treatment labelled B is superior. However, in terms of success probability, A actually succeeds 3.75 times more often than B – 17.5% versus 4.7%! The same sort of error will obtain for any two points where one point is diagonally transposed from another in a different series. The dotted lines radiating from A in Fig. 15a delimit the region in points in other series will induce this same error.

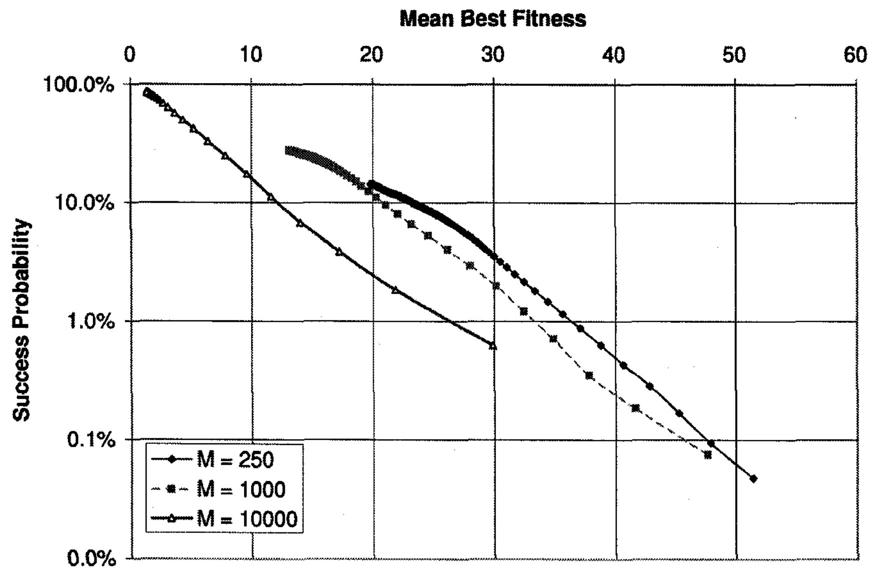
Using mean population fitness in situations where mean best fitness is appropriate doesn't perform much better. Consider points C and D from Fig. 15b. Here, D appears to better than C on mean best fitness – 56.8 to 64.5. In reality, C is nearly twice as good as D in terms of mean best fitness; D averages a best fitness of only 32.6, while C averages a much better 14.0 on this problem. Indeed, the dotted region to the northwest of C, including nearly all of the $M = 1000$ and the entire $M = 250$ series will seem to be better than C, and will perform more poorly on mean best fitness.

We can even hazard a guess of how many fitness evaluations will be required to evince this kind of error, using the data of Fig. 13. We'll use Fig. 15a and the A and B error to illustrate. The 95% error-bound-per-thousand-runs for mean population fitness of the $M = 250$ series is about 0.5 to 0.6 in fitness. The bound-per-thousand-runs around A is a little smaller, between 0.3 and 0.4, giving a joint error between both series of about 0.65 for 1000 runs. The span between A and B is about 6 in mean fitness, which is about 9 times larger than our error bound. The error bound decreases as $\varepsilon = O\left(\frac{1}{\sqrt{n}}\right)$, so we would expect to make this mistake with 95% confidence when $n > 11$. That is, using mean population fitness to predict mean best fitness or success probability will run the risk of making incorrect decisions in nearly all situations where we might be comparing two runs!

The reason that mean population fitness predicts our desired outcome variables so poorly is partly because its variance is very sensitive to population size, unlike best fitness and success probability. A large population samples the space of potential solutions very well, and so will have almost no relation to the best value found therein in the initial generation. A small population samples poorly, and is more sensitive to a single good value. This goes some way towards explaining why the small population sizes in Figs. 15a and 15b actually do better than the large populations on mean population fitness.

Before we go into the computational effort and the issues with that statistic in the next chapter, we can offer one more bit of advice. We argued above that mean best fitness and success probability are not interchangeable. Fig. 16 illustrates this point by showing mean best fitness versus success probability, as in Figs. 14a, 14b, 15a, and 15b. Any diagonal displacement between the two series will result in incorrect decisions when those runs are compared. That is, we can easily choose pairs like A and B or C and D above for Figs. 15a and 15b. However, the degree of bias will be smaller for Fig. 16, as the lines are more closely spaced.

Figure 16



Performance of mean best fitness of run as a predictor of success probability for three treatments. In the ideal case of perfect prediction, all series would fall on the same curve. Any diagonal displacement of the graphs allows a user to be in error when making conclusions about one variable when measuring the other.

Fig. 16 shows that there is still a significant risk of error if mean best fitness is used to compare treatments on success probability, or vice versa. We are now in a position to update Fig. 1 with some advice about comparing treatments by each of our commonly-used statistics. In Fig. 17, we add in the best-of-*k*-runs success probability and best-of-*k*-runs mean best fitness. For all these methods, we offer an informed estimate of the risk of error involved when using them inappropriately. We hope that this will serve the community in evaluating work that has already been done using these statistics as to the biases that are likely to have been introduced. A more detailed discussion of the issues with the computational effort statistic, not otherwise discussed here, is deferred to the next chapter.

Figure 17

Statistic	***Suitability for problems***		***Problems***
	success-based	***improvement-based***	
Mean best fitness	very poor	poor	results depend on number of evals performed
Success probability	poor	very poor	results depend on number of evals performed
Computational effort	excellent	fair	converges slowly; biased for small *n*; arbitrary success criterion for improvement-based problems
Average evaluations to solution	very poor	awful	measures population size and cutoff point; ignores success probability; arbitrary success criterion
Mean fitness of population	very poor	very poor	poor estimator of accurate fitness measures; converges very quickly
Effective success probability	excellent	fair	arbitrary success criterion for improvement-based problems; limited work ratio range for small *n*
Effective mean best fitness	fair	excellent	best fitness values for all runs must be available; biased for extreme work ratios

An evaluation of common methods of comparing evolutionary computation systems. For each comparison statistic, we offer an estimate of how suitable the technique is for problems that are success-based or improvement-based. These problem types are defined in the main text. For each method, we list some significant issues and problems with the technique. Details are explored in the text of this chapter.

Chapter 4: An Analysis of Koza's Computational Effort Statistic for Genetic Programming and a Proposed Solution

Introduction

As early as 1990, John Koza realized the utility of having a statistic to estimate the computational effort to solve a given problem using genetic programming (GP). This statistic, denoted by $I(M, i, z)$ in [Koza 1992], measures the effort required to solve a given problem with 99% probability, and has allowed a generation of GP researchers to compare their results. As the field of evolutionary computation progressed, many researchers realized the need for including statistical information, including confidence intervals and significance testing, along with other results. The state of the art in considering confidence intervals is to perform bootstrap analysis of one's data, as was done in [Keijzer 2001]. There Keijzer, Babovic, Ryan et al. used the bootstrap method to establish confidence intervals on their data for Koza's computational effort statistic, I_{min}. However, the confidence intervals provided by the bootstrap method were so wide as to prompt them to say, "For the Santa-Fe problem ... the width of the confidence interval is nearly as large as 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." This comment motivated the analysis of this chapter.

In the following sections, we will analyze the computational effort statistic. In particular, we provide a detailed analysis of the biases that this statistic introduces, which may leads to counterintuitive results. Some of the issues with the computational effort statistic presented here apply equally to the other statistics introduced in the previous chapter, namely the referred mean best fitness and referred success probability.

Definition of Koza's Computational Effort Statistic

As a GP system progresses towards a solution of a given problem, it is hoped that more and more of the runs will pass a given success criterion. Following Koza, we define the cumulative probability of success *P(M, i)* as the proportion of runs which, after *i* generations, have reported true for the predetermined success predicate for any of the *M* individuals in the current population. If we are hoping to use our GP system to succeed with 99% probability, we should not perform a single run with a huge number of generations until we achieve a 99% chance of success. Indeed, because of premature convergence, this may never occur. Instead, what is commonly done is to use the fact that successive runs are independent and identically distributed to compute the number of independent runs *R(z)* that would be required to solve a given problem with 99% probability. Since the *R* runs are independent, the probability of failure in all of them simultaneously is given by (1).

$$P_{all\ fail} = (1 - P(M,i))^R \tag{1}$$

Let *z* be the desired probability of success. We can compute the number of independent runs required to achieve a solution to a confidence level of at least *z* by taking logarithms and solving for *R*, as in (2).

$$R(z) = \left\lceil \frac{\ln(1-z)}{\ln(1-P(M,i))} \right\rceil \tag{2}$$

Koza's computational effort statistic *I(M, i, z)* is intended to measure how many individual fitness evaluations must be performed to solve a problem to a probability of *z*. It is derived from the cumulative probability of success *P(M, i)* by multiplying by the number of individuals processed at the end of generation number *i*, *Mi*, as shown in (3).

$$I(M,i,z) = Mi\left\lceil \frac{\ln(1-z)}{\ln(1-P(M,i))} \right\rceil \tag{3}$$

This statistic is defined over all generation numbers *i*; to find the "minimum" computational effort $I_{\min}(M, z)$ required to solve a given problem, Koza simply takes the minimum of all sampled *I(M, i, z)*. This gives us Koza's defining equation for $I_{\min}$, (4).

$$I_{\min}(M,z) = \min_i Mi\left\lceil \frac{\ln(1-z)}{\ln(1-P(M,i))} \right\rceil \tag{4}$$

Without loss of generality, we can extend Koza's computational framework to other program induction systems that are not generational in nature, or that use clone pruning or elitism to reduce the number of individuals evaluated, by using instead the total number of individuals tested, *A(M, i)*. This gives us (5).

$$I_{\min}(M,z) = \min_i A(M,i)\left\lceil \frac{\ln(1-z)}{\ln(1-P(M,i))} \right\rceil \tag{5}$$

Analysis of Koza's Minimum Computational Effort as a Sampled Estimator

A number of statistical issues arise when we carefully consider what is happening when we enter our discrete observable data into (5). Importatly, we have not performed an infinite number of runs. Therefore, our values *P(M, i)* are but point estimates of the unknown true probabilities. GP practitioners estimate *P(M, i)* by dividing the number of successes over the number of trials; and since GP runs are computationally intensive, the same runs are typically used to compute the estimators at each generation. Let us denote the number of trials performed by n, and the cumulative number of successes achieved by generation i by $k(M, i)$. What is commonly reported as $I_{\min}(M, z)$ as computed (5) is actually (6). We will show that the discrete formula (6) has very different mathematical properties of interest to EC researchers.

$$\hat{I}_{\min}(M,n,z) = \min_i A(M,i)\left\lceil \frac{\ln(1-z)}{\ln(1-\frac{k(M,i)}{n})} \right\rceil \tag{6}$$

The Experimental Framework

In this chapter, we will illustrate the effect of various aspects of Koza's statistic by performing a simple experiment. We make use of a hypothetical GP problem for which the true computational effort is infinite for generations 1 through 5, and is a constant I_{true} for all generations from 6 on. This is done for concreteness, to make the mathematical examples meaningful and intuitive for GP researchers, and because exact analysis of discrete order statistics is unwieldy and very mathematically intensive. Conversely, it is straightforward to code up a simulation and perform enough runs to establish statistical significance of the results. Consider the case where the exact computational effort $I(M,i,z)$ is constant across all generations. We investigate a hypothetical simple generational GP where the population size is fixed at $M = 500$. Since the specific work $A(M, i)$ is fixed at $500i$ and we do not vary the success probability $z = 0.99$, we can solve for the cumulative success probability schedule $P(M, i)$. For the moment, we will rewrite (5) to ignore the ceiling operator as (7); this caveat will be explained in the next section.

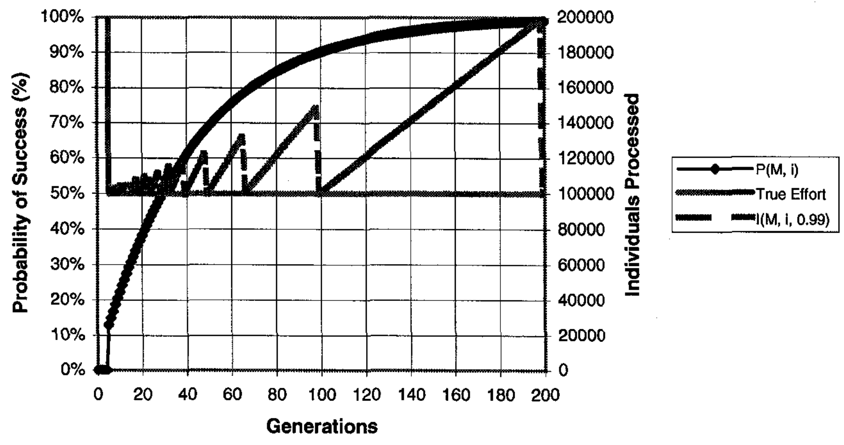
$$I_{true} = Mi\left(\frac{\ln(1-z)}{\ln(1-P(M,i)}\right) \tag{7}$$

Solving for *P(M, i)* in (7) gives (8).

$$P(M,i\,|\,z) = \begin{cases} 0 & 1 \le i \le 5 \\ 1-e^{\left(\frac{Mi\ln(1-z)}{I_{true}}\right)} & i > 5 \end{cases} \tag{8}$$

These equations are graphed in Fig. 1, and are explained in the next section.

Figure 1



Synthetic performance curves for a hypothetical GP run where the exact computational effort is 100 000 individuals processed. The sawtooth curve indicates the effect of the ceiling function on the reported computational effort.

The Potentially Harmful Effect of the Ceiling Operator

Our analysis begins with understanding the effect of the ceiling operator ($\lceil \cdot \rceil$) in (5) and (6). We will begin with a brief discussion of why this operator is inappropriate for sampled data. Note that the estimated probability $\hat{P}(M,i)$ is not equal to the exact probability *P*(*M, i*). The distribution of $k(M,i)$ is binomial. To the degree to which the binomial distribution can be approximated by a normal distribution, there is approximately a 50% chance that the true probability *P*(*M, i*) lies above the estimate $k(M,i)/n$; otherwise it will lie below. It follows that we cannot correctly conclude there is a 99% chance of success if we rerun our process *R(z)* times. An example may make this clear. Suppose that for a particular problem, we observe 78 successes in 100 trials. Our point estimate of the probability of success is then 0.78. We can then compute $R(0.99)$ using (2) as $\left\lceil \frac{\ln 0.01}{\ln(1-0.78)} \right\rceil = \lceil 3.0415 \rceil = 4$. However, we know that our estimate has some variability associated with it; the odds of the unknown true probability being exactly 0.78 are vanishing. Should the true probability be as high as 0.785, $R(0.99)$ becomes $\left\lceil \frac{\ln 0.01}{\ln(1-0.785)} \right\rceil = \lceil 2.9980 \rceil = 3$. If so, the estimated computational effort will be 4/3 of the true computational effort. Because 0.785 is so close to 0.78, we can expect that there is a nearly 50% chance that the true probability will exceed 0.785. An exact calculation gives 41.6%. We have now found a situation where, roughly half the time, our estimate will be in error by $4/3 - 1 = 33\%$! This calls into question the utility of the ceiling operator. The ceiling operator was put into place to capture the fact that we cannot effectively perform a non-integral number of runs. However, this ceiling operator is useful only when our knowledge of the probability of success is exact; when we have only approximate knowledge, there is a risk that our data will lie close to a jump discontinuity of (6). This renders the utility of the ceiling operator somewhat marginal.

In Fig. 1 we see the effect of the ceiling operator on the reported workload when the actual workload is constant. The net effect is to overestimate the true workload by an amount that is a function of the estimated probability *P(M, i)*, and of the desired success probability *z*. The maximum error imputable from this source is a 100% overestimate, at *P(M, i)* = *z*. However, if the total success probability does not attain 90%, the maximum error would be a 50% overestimate. It is worth noting that if the success probabilities are everywhere small (< 25%), this effect becomes modest (< 6.3%).

For the remainder of this section, we will forgo treating the number of runs as an integer-valued quantity, and instead work on the continuous, infinitely differentiable statistic, (7).

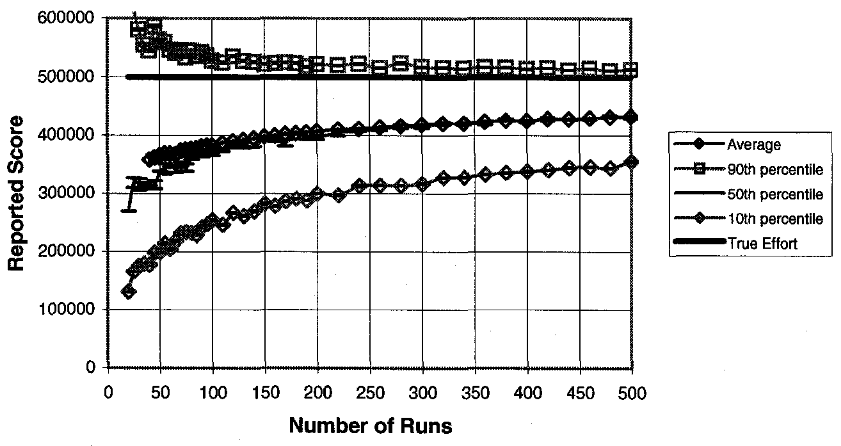
Why Koza's $I(M, i, z)$ Underestimates the True Computational Effort

The central statistical phenomenon discussed in this section arises when we consider the effects of the minimum operator in (7). The minimum of J random variables falling around a given average value will tend to lie below this point; and the difference will tend to be larger as J increases. In the context of GP analysis, the random variables are the estimates of the computational effort $I(M, i, z)$, although the results of this section will apply equally well to the minimum of any set of random variables. Estimating the effective mean best fitness and effective success probability over a series of M and G values suffer from this problem as well. The exact analysis of discrete order statistics is challenging due to a dimensional explosion, so we resort to simulation to illustrate this effect and its magnitude.

Consider our example where the true computational effort is held constant over a span of generations. We modelled a hypothetical GP problem where the cumulative probability of success is 0 for generations 1 to 5, and then climbs from 2.7% at generation 6 to 37.2% at generation 101. This schedule maintains a fixed true computational effort of 500 000 fitness evaluations from generations 6 through 101.

For this simulation, we generated 10 000 synthetic data sets with the cumulative probability of success given by the schedule shown by (8). In order to do this, we chose the data so that exactly $\lfloor 10\,000\,P(M,i) \rfloor$ runs succeed after generation *i*. For example, to obtain a true computational effort of 500 000 individuals at generation 10, we substitute into (8): $P(M,i \mid z) = 1 - e^{\left(\frac{500 \cdot 10 \ln(1-0.99)}{500\,000}\right)}$ to get *P(M, i)* = 0.045007. We then choose the data so that there are $\lfloor 10\,000\,P(M,i) \rfloor$ = 450 successes by generation 10. From this data set, we then sample *n* independent runs with replacement, and compute $I_{\min}$ using (7) from this sample. In Figure 2, we present the observed distribution of $I_{\min}$ after 53 simulated "generations". This isolates the effect of taking the minimum of $53 - 5 = 48$ random variables possessing the same *I(M, i, z)*. The number 48 was chosen because of the many integral submultiples made possible by this choice.

Figure 2



Reported means, 10th, 50th and 90th percentiles of computational effort $I_{\min}$ estimated for a problem where the true computational effort is 500 000 evaluations, graphed as a function of the number of runs *n* performed. 95% confidence interval error bars are shown. The 95% CI error bars are per-experiment error bars; that is, we would expect that the true value lies within the given error bars 95% of the time for each experiment. Holding the error probability at 95% across all 22 experiments would increase the error bar span by about 60%.

Fig. 2 shows that not only does $I_{\min}$ underestimate the true computational effort, but that it does so with high probability – nearly 90% of the time. The magnitude of this underestimate decreases as the number of runs performed n increases.

A second effect of the minimum in (7) stems from the non-decreasing nature of $P(M,i)$. We can use a Taylor series expansion to show this effect. Modeling the sampling error in the probability estimation $\varepsilon(M,i) = \frac{k(M,i)}{n} - P(M,i)$ as an error term in (7) gives (9).

$$I_{est.} = A(M,i)\left\{\frac{\ln(1-z)}{\ln(1-(P(M,i)+\varepsilon(M,i))}\right\} \tag{9}$$

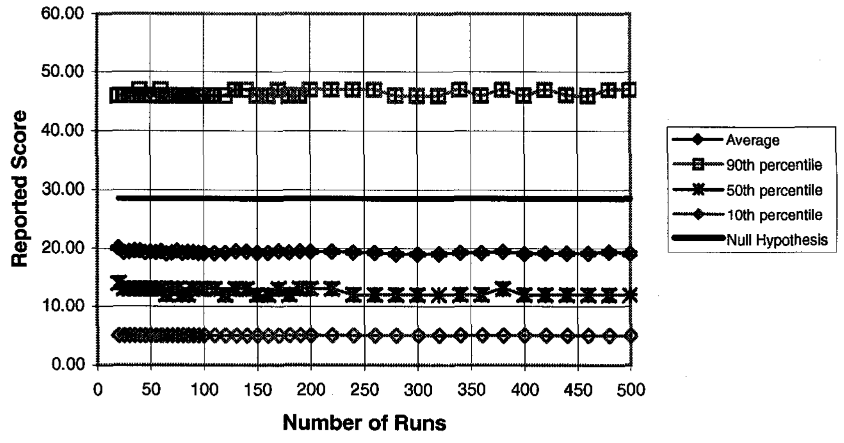
Expanding this expression as a Taylor series in *ε(M, i)* to first order, and abbreviating *P(M, i)* as *P* for terseness, we get (10).

$$I_{est.} = A(M,i)\left\{\frac{\ln(1-z)}{\ln(1-P)} + \frac{\ln(1-z)}{\ln(1-P)^2(P-1)}\varepsilon + O(\varepsilon^2)\right\} \tag{10}$$

The error term in (10) increases dramatically as the true probability *P* approaches 0, so the specific errors tend to be larger in regions where the true probability *P(M, i)* is low – that is, at earlier generations. Since the minimum in (7) will be more sensitive to larger specific errors, the generation at which computational workload is reported to be "minimized" in a finite experiment will tend to underestimate the actual generation.

To quantify the magnitude of this effect, we computed the generation at which computational effort was reported as minimized in the simulations performed above. The results are presented in Fig. 3. As we can see, the median lies well below the mean, so this distribution is therefore markedly skewed towards larger values.

Figure 3



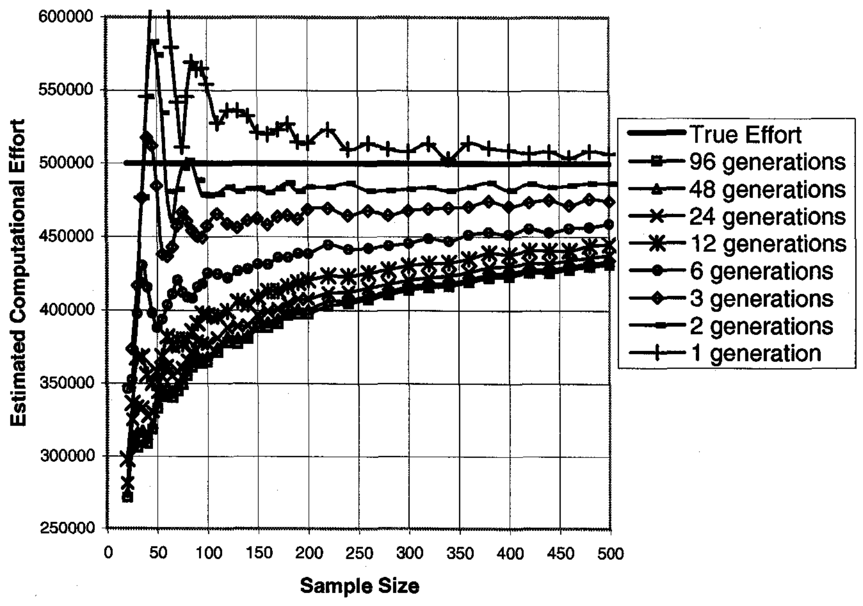
Reported means, 10th, 50th and 90th percentiles of generation at which the computational effort is seen as minimized, graphed as a function of the number of runs *n* performed after 53 generations. 95% CI error bars are shown.

The Magnitude of the Underestimate Depends on the Number of Generations that are (Nearly) Identical

We will now demonstrate the effect that the number of generations has on the magnitude of bias. However, we should explain one mathematical effect before continuing. In (6), we see that *k(M, i)* and *n* are both integer-valued. If we are estimating the minimum computational effort, the achievable values of *k(M, i)* are constrained to be the non-negative integers less than the number of runs performed, *n*. Since the set of achievable values is discretized, the usual summary statistics will not report useful values. The mean of *I(M, i, z)* will be infinite, for instance, as there is a nonzero probability that 0 successes were obtained across all *n* experiments – hence producing an infinite estimate for *I(M, i, z)*. In order to alleviate this problem, we have taken a compromise position of reporting the 80% trimmed mean in the results that follow. The 80% trimmed mean is computed by reporting the mean of all data between the 40th and the 60th percentiles. This is still suboptimal in some sense because large values of

$I(M, i, z)$ continue to bias the trimmed mean more than small values do. Consider the data for the 1-generation case shown in Fig. 4. We would expect the estimate to approximate the true effort and be smooth as a function of sample size, but the requirement that *k(M, i)* must take on integer values causes oscillations and causes the trimmed mean to lie above the theoretical mean. Both of these effects become more pronounced for small sample sizes as shown in Fig. 4.

Figure 4



80% trimmed mean of the reported computational effort $I_{\min}$ as estimated for a problem where the true computational effort is 500 000 evaluations, graphed as a function of the number of runs *n* performed for the given number of generations. The hypothetical population size is 500 individuals, and the success probability is 0 from generations 1 through 5. It then increases from 2.7% at generation 6 to 37.2% at generation 101 according to the schedule of (8).

Comparison with Real Life: Experiments with Ants on the Santa Fe Trail

The first question that needs to be asked, in light of these data, is how likely is it that real GP computational effort data is affected by the biases considered herein? After all, the key variable that influences the amount of negative bias is the number of generations for which the true $I(M, i, z)$ is approximately constant. Perhaps this is nearly or exactly 1 for real GP data, leading to a very small actual bias in practice. To investigate this question, we performed a large GP run to get a baseline, and then subsampled this data set to examine the distribution of reported I_{min} values.

We used ECJ version 7.0 to generate 27 755 runs of Koza's Artificial Ant on the Santa Fe trail [Koza 1992, Luke 2001]. Space does not permit a detailed description of the parameter settings used; we used the defaults for Koza GP and the Santa Fe trail problem, with the exceptions that tournament selection of size 7 was performed, and the ant was allowed to run for 600 time steps [Koza 1992]. From this large data set, the best estimate of the true computational effort was I_{min} = 479 345 individuals (minimum reported at generation 19; 2 421 successes observed in 27 755 trials; 95% confidence interval is 460 633 to 498 841). We then selected 10 000 random subsamples of size 50 with replacement from this data set, and computed I_{min} and the generation at which computational effort was minimized for each. The median of the observed I_{min} values was 381 670 individuals, 25.6% below the population value (95% CI, 372 133 to 386 610). 69.5% of the observed I_{min} values were below the population value, indicating the presence of significant bias (95% CI, 68.6% to 70.5%). The mean generation at which convergence was reported in the sample was 15.41 (95% CI, 15.25 to 15.56). 69.8% of the observed reported generations were earlier than the observed minimum at generation 19.

A replicate of these data, for the case where the ceiling operator was applied throughout yielded slightly higher absolute results (population I_{min} = 484 500 individuals), but differed by less than 2% in all relative ratios.

Conclusions and Future Work

We have performed a statistical analysis of Koza's *I(M, i, z)* statistic and have found it wanting in several important ways. Firstly, the effect of the ceiling operator introduces a somewhat arbitrary non-linear bias in the estimation process. This tends to ignore those data that fall in a regime where the fractional part of $R(z)$ is large. Second, the effect of taking the minimum of a number of estimators tends to underreport the true computational effort. This underestimate increases with the number of estimators that compete for the title of best generation and decreases with increasing sample size n. That is, the worst behaviour is seen where many consecutive generations have similar computational effort, and when the cumulative probabilities of success are small. In addition, this process tends to report that the optimum generation is found somewhat earlier than its true value, although this result has considerable variance. One subtle caution is in order for users of GP systems with small population sizes, such as multi-objective optimization and steady-state GPs. For such systems, the number of generations is effectively very large, approaching the total number of individuals generated in the case of steady-state schemes. Since the magnitude of the bias increases with the number of generations over which we minimize, the estimation error would be expected to be relatively large, as the minimum of essentially hundreds of thousands of random variables is computed!

Ideally, we would have an analytic or simulation-based model with best-fit regression curves to be better able to estimate the magnitude of these purely statistical effects. The computational effort statistic is far too important to give up on! Ideally, we would like to be able to give a set of *k(M, i)* data to a program and have it generate an unbiased estimator of our minimum computational effort $I_{\min}$. An interim solution, the y-test, is given in the second half of this chapter.

These results suggest several recommendations that would be of use in future GP research. The first is to always indicate the number of runs performed in one's experiments, preferably directly in the legend of the performance curve figure, if one is given, else in a summary table. We should be careful to report the number of effective generations over which we take the minimum in (6). The GP community might be well served by dropping the ceiling operator from (6), although this may be subject to debate. We also would recommend that practitioners choose relatively large run counts, on the order of 500 runs, to produce data that minimize the systematic errors presented herein. Finally and most importantly, we must ensure that we make available the exact values of $k(M,i)$ that we obtain from our experiments. With such data, future researches will be able to provide an unbiased or less-biased estimator of computational effort for legitimate comparisons of data from different authors. Without such data, there can be only an estimated correction applied based on historical data, or an approximation derived through a laborious and error-prone digitization of performance curve figures.

We now continue with a modest attempt at overcoming these problems by introducing a new statistic test particular to evolutionary computation: the y-test.

The Y-Test: Fairly Comparing Experimental Setups with Unequal Effort

Introduction

Computational effort is a point statistic, and so gives no measure of the variability about the mean. It is therefore unsuitable for significance testing as is. We have shown earlier that computational effort has some undesirable statistical properties, even as a point statistic. For instance, we showed that on average it underestimates the true computational effort, by 25% or more for commonly encountered success probabilities and run counts. Worse, the computational effort is underestimated more significantly when fewer runs are performed. This results in the unfortunate circumstance that already-published results may not be reproducible when more runs are performed on the exact same problem.

Instead of pursuing modifications of the computational effort statistic, in this setion we look at the problem from a different perspective. Suppose we take an established statistical test and modify it so that it can be used to address the sorts of questions that commonly occur in evolutionary computation? A common problem is that method A delivers better results than method B, but takes more fitness evaluations to do so. We will show that 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.

We adapt an underused variant of the Mann-Whitney-Wilcoxon (MWW) test

[Mann 1947, Wilcoxon 1945], a *t*-test on the ranks of the data. The *t*-test on the ranks, henceforth designated the t_R-test, has been demonstrated to be equivalent to the familiar Mann-Whitney-Wilcoxon test by Zimmerman and Zumbo [Zimmerman 1989], after theoretical developments disclosed the equivalence between the two tests. The Mann-Whitney-Wilcoxon test has recently been used for evolutionary computation research, as it has some very nice statistical properties, such as its relative distribution-invariance [Zimmerman 1992] and high power [Gibbons-Jean 1991]. The t_R-test adds to this list the virtue of being easy to compute, hence its adoption as the base for the *y*-test. As the MWW test and the t_R-test are non-parametric tests, they compare the medians of two samples rather than their means. This is generally a good idea for comparing fitness function-based outcomes. Fitness functions used in evolutionary computation are often non-linear, and sometimes have arbitrary large penalty values. This pushes up the mean population fitness and can hurt the mean best fitness as well. It is not uncommon to have a distribution with an infinite mean due to a non-vanishing probability of encountering an infinite fitness value. Using rank-based statistics addresses all these concerns simultaneously, as it requires only that a total order be defined on the outcomes. We adopt this approach in developing the *y*-test.

The next section discusses the derivation of key components of the *y*-test, in four parts:

- determining the distribution of the minimum of several samples drawn from a reference distribution;
- considering order statistics to derive an equivalency between one quantile of a distribution and an arbitrary quantile of its iterated distribution;
- computing the sampling distribution of a specific cumulative probability; and

- computing the maximum run count usable when comparing two quantiles.

Readers interested primarily in applying the y-test are directed to the section labelled "Y-Test", where the test is derived. In the section labelled "Y-Test Discussion," we discuss some statistical properties of the y-test. Practitioners interested in using the y-test may be interested in an Excel spreadsheet [Christensen 2006a], and C++ code [Christensen 2006b], developed by the author for the benefit of the community.

Order Statistics and Iterated Distributions

Distribution of the Minimum of *r* Samples Drawn from a Given Distribution

Suppose that we are interested in the distribution of the minimum of 2 independent runs of a particular process, given a set of observed outcomes. Let the distribution D be the distribution of the actual scores obtained by running the process. For simplicity, let us assume D is discrete and has uniform probability at each of its n values. Let d_i be the i-th smallest member of D. Then $P(x=d_i)=\frac{1}{n}$ for all i = 1 ... n. In Fig. 5, we show the simultaneous distribution of 2 independent draws of D.

Figure 5

		Sample 1				
		d_1	d_2	d_3	...	d_n
Sample 2	d_1	d_1	d_1	d_1	...	d_1
	d_2	d_1	d_2	d_2	...	d_2
	d_3	d_1	d_2	d_3	...	d_3
	...	...	...	...	...	...
	d_n	d_1	d_2	d_3	...	d_n

Simultaneous distribution table for the minimum of 2 independent runs, where $d_1 < d_2 < ... < d_n$ are equiprobable outcomes from each run. Each outcome has equal probability.

Let d_{it} be a new random variable taking on the value of the minimum of each run. It can be seen from Fig. 5 that the number of occurrences of the i-th smallest value is given by the difference between two nested squares differing in side length by 1. That is,

$P(d_{it} = d_i) = \frac{(n-i+1)^2 - (n-i)^2}{n^2}$. This relation will similarly hold for 3 or more runs, where the differences are taken between two nested cubes or hypercubes instead. Let r be the number of runs performed. The correct probability in the general case is $P(d_{it} = d_i) = \frac{(n-i+1)^r - (n-i)^r}{n^r}$. The probability that we will observe a value less than or equal to d_i is then given by (11).

$$P(d_{it} \le d_i) = \frac{n^r - (n-i)^r}{n^r} \tag{11}$$

We can identify a particular cumulative proportion q with a discrete value from D for the case where qn is integral using the relation $Quantile(q,D) = d_{qn}$. (12a) and (12b) rewrite (11) to refer to the case where a cumulative proportion q of D is desired.

$$P(d_{it} \le Quantile(q,D)) = \frac{n^r - (n-qn)^r}{n^r} \tag{12a}$$

$$P(d_{it} \le Quantile(q,D)) = 1-(1-q)^r \tag{12b}$$

(12b) is strictly valid only when $qn \in Z_n$. We can extend this relation to rational values of q by taking the limit of (12a) as n increases to infinity. This gives a relation with the same formula as (12b). We can then extend (12b) to an arbitrary irrational number $\phi \in R; 0 \le \phi \le 1$ by writing ϕ as $\phi = \lim_{b\to\infty} \frac{\lfloor \phi b \rfloor}{b}$. Taking this limit in (12b) again gives the same function form, but now shown to be valid for an arbitrary positive cumulative proportion q. If we take q to be a real-valued cumulative proportion, (12b) applies equally well to continuous distributions through their probability distribution functions as for discrete distributions through their discrete cumulative probability functions.

Let D^r be the distribution of the minimum of *r* independent samples drawn from *D*. We term this distribution the iterated distribution of *D*.

Let q_r be a cumulative proportion drawn from D^r.

Notice that $q_r = P(d_{it} \leq Quantile(q, D))$, as defined above. We can then manipulate (12b) to solve for *q*, giving (13).

$$q = 1 - (1 - q_r)^{1/r} \tag{13}$$

We can use (13) to determine any desired cumulative proportion of D^r in terms of an equivalent cumulative proportion of *D* and hence the entire distribution of D^r exists and is computable whenever *D* is.

Treatments with Differing Fitness Evaluation Counts

We would also like to generalize these results for non-integral values of *r*. In the following, we assume that *D* is a continuous distribution. Consider two treatments with different integral number of runs; label the numbers of runs *a* and *b*. (14) equates an arbitrary cumulative proportion q_a from D^a with a corresponding cumulative proportion q_b from D^b. The derivation proceeds from (13) as follows:

$$\begin{aligned} q = 1 - (1 - q_a)^{1/a} &= 1 - (1 - q_b)^{1/b} \\ (1 - q_b)^{1/b} &= (1 - q_a)^{1/a} \\ 1 - q_b &= (1 - q_a)^{b/a} \\ q_b &= 1 - (1 - q_a)^{b/a} \end{aligned} \tag{14}$$

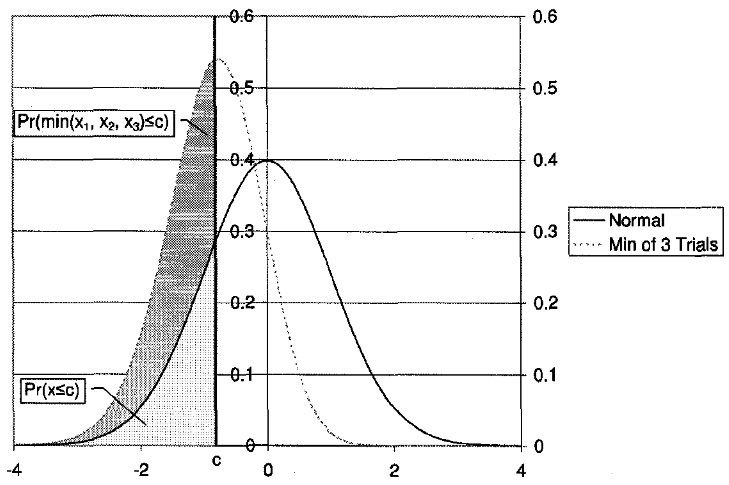
(14) holds when $a, b \in N$. Comparing (14) with (13), we can see that we have effectively extended (13) to the case where $r \in Q, r > 0$. We can extend (13) to the

positive reals by letting $r \in R; r > 0$, writing r as $r = \lim_{b\to\infty} \frac{\lfloor rb \rfloor}{b}$, and substituting into (14).

This again gives an equation that is symbolically identical to (13).

A quick example of how to apply (13) and (14) may be illustrative. Suppose that we want to know the median of the minimum-of-3-runs of the standard normal distribution. Set $r = 3$ and $q_3 = 0.5$ in (13), and we find that $q = 1-(1-0.5)^{1/3}$. Numerically, this is equal to 0.2063. Therefore, the 20.63rd percentile of the standard normal distribution is equal to the median of the minimum of 3 independent runs of the standard normal. This is illustrated schematically in Fig. 6.

Figure 6



Probability density functions of the standard normal distribution and of the distribution of the minimum-of-3-trials of the standard normal. The vertical line marked by c indicates the median of the minimum-of-3 distribution. The area labeled $\Pr(x \le c)$ contains 20.63% of the area under the solid curve. 50% of the area under the dotted curve lies in the area labelled $\Pr(\min(x_1, x_2, x_3) \le c)$. These two proportions are related by (13), as $0.2063 = 1-(1-0.5)^{1/3}$.

Testing the medians of two processes where one uses fewer fitness evaluations than the other can thus be easily achieved by considering a higher cumulative proportion of the faster process. In practice, we normally only have access to experimental data drawn from a finite number of runs of the competing processes. We will therefore have a limit on the highest cumulative proportion that we can reliably estimate – or, the maximum number of runs considered at once. We elaborate on this in the following section.

Sampling Distribution of a Cumulative Proportion

In the following, it will be beneficial to estimate the sampling distribution of the median. We can view the sampling distribution of a cumulative proportion *q* as a binomially distributed random variable. Suppose we are interested in the distribution of cumulative proportions *q* from some distribution *D*, and we have a set of independent samples from *D* as data. We can model the distribution of the cumulative proportion *q* as that of the numerically equivalent proportion π. It is well known that the mean of the binomial distribution with probability of success π is $\mu_\pi = \pi$ and that the standard deviation is given by (15).

$$\sigma_\pi = \sqrt{\frac{\pi(1-\pi)}{N}} \tag{15}$$

We can therefore give a confidence interval for a given cumulative proportion *q* as follows. First, determine a confidence interval about the equivalent proportion and determine upper and lower bounds for a desired significance level α. Second, determine the quantiles in the sample corresponding to the upper and lower bounds of the proportion's confidence interval.

Let *s* be the set of *N* independent samples drawn from the distribution *D*. The probability that $Quantile(\pi, D) \in [Quantile(\pi - z\sigma_\pi, s), Quantile(\pi + z\sigma_\pi, s)]$ is approximately equal to $P(x < z)$, where *x* is a cumulative proportion drawn from the standard normal distribution. This is strictly correct only when there are no plateaus in the cumulative distribution function of *D*; that is, when $\forall x, \Pr(d = x \mid d \in D) = 0$. This suggests a natural limit on the most extreme cumulative proportion we can estimate from a given data set; if (16a) or (16b) are satisfied then we are obviously in trouble.

$$\pi - z\sigma_\pi < 0 \tag{16a}$$

$$\pi + z\sigma_\pi > 1 \tag{16b}$$

We can use this criterion to determine the smallest and largest feasible cumulative proportion that we can estimate from a given sample of size *N*. We invert the inequalities in (16a) and (16b) to get criteria for reliable values for π:

$$\pi - z\sigma_\pi > 0 \tag{16c}$$

$$\pi + z\sigma_\pi < 1 \tag{16d}$$

We combine (15) and (16c) to solve for π in (17a).

$$\pi - z\sqrt{\frac{\pi(1-\pi)}{N}} > 0$$

$$\pi^2 > z^2\frac{\pi(1-\pi)}{N}$$

$$N\pi > z^2 - \pi z^2$$

$$\pi > \frac{z^2}{N+z^2} \tag{17a}$$

Similarly, we can combine (15) and (16d) to get (17b).

$$\pi + z\sqrt{\frac{\pi(1-\pi)}{N}} < 1$$

$$z^2\frac{\pi(1-\pi)}{N} < (1-\pi)^2$$

$$z^2\frac{[1-(1-\pi)]}{N} < 1-\pi$$

$$z^2 - (1-\pi)z^2 < N(1-\pi)^2$$

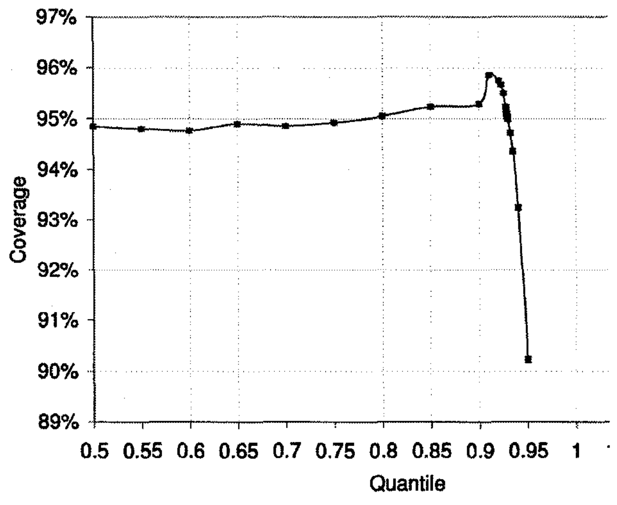
$$\frac{z^2}{N+z^2} < 1-\pi$$

$$\pi < 1 - \frac{z^2}{N+z^2} \tag{17b}$$

$$\frac{z^2}{N+z^2} < \pi < 1 - \frac{z^2}{N+z^2} \tag{17c}$$

Since we use this relation extensively later, it is worthwhile checking the accuracy of this approximation. For a 95% two-sided confidence interval, the critical value of z is 1.96. So if $N = 50$, we can estimate values confidently for cumulative proportions in the range $0.0714 < \pi < 0.9286$. In Fig. 7, we have graphed the observed coverage of several cumulative proportions using (18) as a function of the target cumulative proportion q.

Figure 7



An example of experimentally verifying the upper limit of 92.86% as the maximum upper bound for which we can safely determine confidence intervals. For each data point graphed, we generated 5 000 000 sets of 50 random variates on [0, 1]. We counted the data set as covered if the 95% confidence interval on the target cumulative proportion as determined in (18) covered the cumulative proportion in the distribution. There is a 0.1% chance that any of the data points lie outside the indicated confidence intervals.

$$Quant.(D,p) \in [Quant.(s, p-\sigma_p), Quant.(s, p+\sigma_p)] \tag{18}$$

Fig. 7 shows that the fit to the expected coverage is quite good.

Maximum Run Count When Comparing Medians

Another variable of interest is the largest number of runs for which we can estimate the median, given a particular data set. We can rewrite (13) to solve for r, based on a particular reference cumulative proportion, as in (19). For instance, if we are interested in the median, (20) gives the number of runs as a function of the cumulative proportion q.

$$r = \ln(1-q_r)/\ln(1-q) \tag{19}$$

$$r = \ln(0.5)/\ln(1-q) \tag{20}$$

We can then substitute our lower bound derived in (17a) for q into (19) to get the answer to an interesting question: given a set of N samples and a target type I error α, what is the largest number of runs over which can we minimize at once? (21) gives the closed-form for r.

$$r = \frac{\ln(1-q_r)}{\ln\left(\frac{N}{N+z_\alpha^2}\right)} \tag{21}$$

For instance, with $N = 50$ samples, and $z = 1.96$ for $\alpha = 5\%$, we can estimate the distribution of the median of at most $r = 9.36$ runs reliably.

Another way to consider this result is that a valid comparison with this sample size and desired significance level can be performed so long as the efficiency ratio between the two groups is less than 9.36; that is, so long as one process does not produce results more than 9.36 times as quickly as the other process. For a known efficiency ratio, we can solve (21) for N to get a minimum sample size to use when comparing two groups as in (22). Of course, we must respect the usual statistical considerations in the choice of sample size as well.

$$\begin{aligned} \ln(\frac{N}{N+z^2}) &= \ln(1-q_r)/r \\ \frac{N}{N+z^2} &= (1-q_r)^{1/r} \\ N &= \frac{(1-q_r)^{1/r} z^2}{1-(1-q_r)^{1/r}} \end{aligned} \tag{22}$$

For instance, if the efficiency ratio between two processes is $r = 10$, using $z = 1.96$ for 95% coverage, we require $N = 54$ samples to compare the median of the faster process against the equivalent cumulative proportion of the slower process.

Y-Test

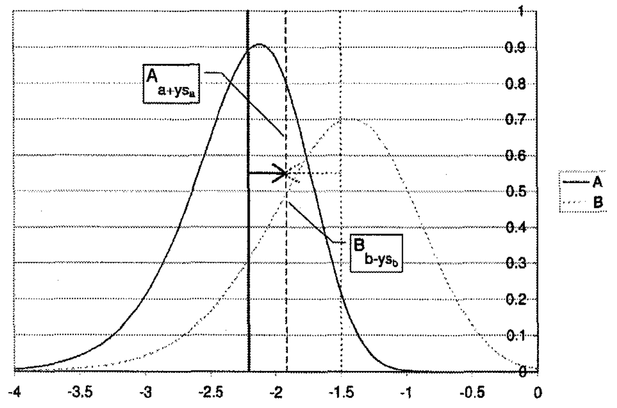
We are now ready to introduce the y-test. Suppose that we are comparing the two processes A and B with different known efficiencies. Let q_a be a target cumulative proportion among the outcomes of A. Let q_b be the corresponding cumulative proportion in the outcomes of B, computed using (13) or (14). Assume that (17c) is satisfied for both q_a and q_b. We are then justified in using (15) to approximate the cumulative proportion standard deviation for q_a and q_b. We denote standard deviations for the two cumulative proportions s_a and s_b, and denote the two sample sizes N_a and N_b. Without loss of generality, suppose that $Quantile(q_a, A) < Quantile(q_b, B)$. There is then some nonnegative value y which satisfies (23).

$$Quantile(q_a + ys_a) = Quantile(q_b - ys_b) \qquad (23)$$

This situation is illustrated in Fig. 8.

We can determine this value of y numerically for two given data sets by applying a simple root-finding algorithm, such as in [Press 1992a], to a user-supplied algorithm that interpolates to find a given cumulative proportion from a finite data set. This is made easier by noting that for any distribution, the area to the left of a given cumulative proportion is monotonic non-decreasing. y in (23) therefore must have a unique value, or possibly a

Figure 8



A diagram illustrating how the y-test is defined. Both curves are the probability density functions of their respective distributions. $N = 25$ has been chosen as the sample size of each distribution, and we are comparing the medians of each distribution. Here y has been chosen to satisfy (23) under the assumption that a perfect sample of 25 was chosen from each true distribution.

unique interval if there are plateaus in both A and B. In the latter case, we define y to be the midpoint of the range of acceptable ys.

Once we have the numerical value of y, we can compute the significance level for a given value of y using an isomorphism. Consider a parallel statistical system where two normally-distributed data sets A' and B' are being compared using a standard t-test. Let A' have a sample mean of 0 and a sample standard deviation of s_a. Let B' have a sample mean of $y(s_a/\sqrt{N_a}+s_b/\sqrt{N_b})$ and a sample standard deviation of s_b, where y is a free parameter. The upper confidence interval of A' just touches the lower confidence interval of B', exactly as in (23). A t-test between A' and B' would give the t-score given in (24).

$$t=\frac{y(s_a/\sqrt{N_a}+s_b/\sqrt{N_b})}{\sqrt{s_a^2/N_a+s_b^2/N_b}} \tag{24}$$

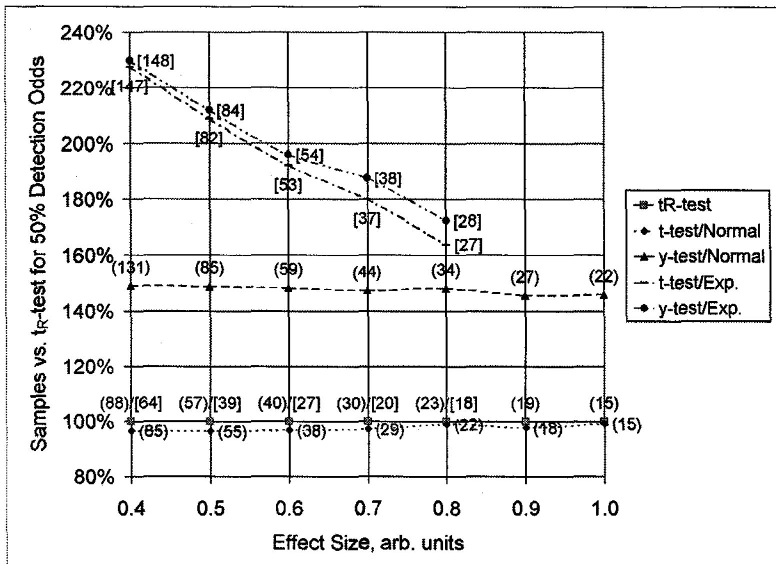
We now have an isomorphism between a test for which we can compute p-values, and a novel test. Due to the isomorphism, the p-values of the two tests are the same. Therefore, we can use (24) to convert the y-score to a t-score and determine the p-value for the t-score directly using $N=N_a+N_b-2$ degrees of freedom. The final p-value then gives an approximate probability that the quantile at q_a is superior to that at q_b, after allowing for the differing computational efficiencies between A and B.

Y-Test Discussion

In introducing a new statistical test, one should test it for the probability of committing both Type-I and Type-II errors. The probability of committing a Type-I error, denoted by α, represents the probability of falsely detecting a significant difference when none exists. A simple test of the Type-I errors on the y-test shows that Type-I errors are appropriately controlled.

It is the probability that a Type-II error is committed where the y-test leaves something to be desired. We tested the medians of two sets of 100 exponentially-distributed random variates with mean 1, where one set had a constant bias of 0.2 added. In this case, the y-test was only able to discriminate 29.5% of 100 000 independent trials, where the t_R-test discriminated 60.5% of them. In a separate trial on standard normally distributed variables with a bias of 0.4, the y-test discriminated 62.0% of 100 000 independent trials, where the t_R-test correctly detected a significant difference in 78.5%. This limitation is fortunately in the correct direction; that is, the y-test is conservative. A loss of power is however suboptimal for a statistical test – we would like the most sensitive test available, all else being equal. A more useful measure to the practitioner of evolutionary computation is the number of independent runs required to achieve a certain level of sensitivity.

Figure 9



A diagram illustrating the number of runs required to detect significance with a 50% probability, as a function of effect size. Numbers of runs are reported as a fraction of the runs required for the t_R-test. Two distributions are considered: the normal distribution, for which raw run counts are given in parentheses (50); and the exponential distribution, for which run counts are given in brackets [50]. Run counts are spline-interpolated from power tables estimated from 100 000 trials of the test in question; error bars should be small.

Fig. 9 illustrates the number of runs required to have to a 50% chance of correctly reporting a real difference, for various levels of effect size. As we can see, the y-test requires more runs to detect an effect of a given size than the t_R-test, typically 50% more for normally distributed variables and as much as 100% more for exponentially distributed variables. However, it does perform as well as the t-test on exponentially distributed variables. This may not always be a problem in a computational environment where additional runs are of modest cost.

As a result of this loss of power, we can recommend the y-test as an interim solution to the problem posed at the beginning of this section. If it reports that a difference between setups is significant, it is correct to within α. However, if it reports "no significant difference", there may be a significant difference between the two groups that lies beyond its ability to resolve.

A sample application of the y-test is in order. Suppose we have access to the outcomes of 100 runs of two evolutionary computation methods. Method A generates outcomes after 25 000 fitness evaluations, while B generates outcomes after only 10 000 fitness evaluations. We presume that this is a minimization problem, so smaller outcomes are better. We might decide that one run of the slower process is to be compared against several runs of the more efficient process. In this case, we would choose $q_a = 0.5$ and use (14) to compute that $q_b = 0.8232$. Note that we have to compare a higher cumulative proportion for the faster process, as higher cumulative proportions are poorer performers. We would then use the procedure given in the section labelled "Y-Test" to solve for y, and then compute the p-value for the two data sets.

This last example raises an interesting point: why stop at the median of the slower process? After all, so long as both distributions satisfy (17c), we can go to as high a cumulative proportion as we like. When using evolutionary computation as an optimization technique, we are usually either trying to solve a very hard problem, or trying to solve a problem of modest difficulty very well. In the former case, we probably cannot do statistics properly and use many runs, since fitness evaluations are likely to be expensive. In the latter case, it is of great interest to us to know how the values are distributed. For example, an evolutionary computation (EC) technique may be competing against simulated annealing (SA) in a particular problem. The EC technique may be slower, but have higher variability in the results. The SA solution might require fewer fitness evaluations, but have smaller variance. In this case, EC would appear to be completely dominated by SA. However, if the variability is two-sided, EC may still be indicated as the superior algorithm. If EC produces both better and worse results than SA on successive runs, then we can perform several runs of the EC and take the best. In doing so, we are essentially probing the high-performance tail of the distribution.

It is precisely in this case where the present paper's strengths lie. We have given a recipe for performing careful hypothesis testing on the high-end tails of unknown distributions, out to the limit given by (21). Indeed, this is the best that can be hoped for, without explicitly creating a model of outcome distribution as a function of input parameters.

This page is intentionally left blank.

Chapter 5: No Free Lunch, Trees and Improving Genetic Programming

No Free Lunch and Genetic Programming

The No Free Lunch Theorem states that no single algorithm outperforms memorizing random search or enumeration when amortized over all possible functions [Wolpert 1995]. This is easy to see and, indeed, unsurprising: if nothing is known about the structure of a domain, then all problems in it are equally likely; and any particular search or learning algorithm can be expected to perform better than chance on some randomly selected problems and worse than chance on others. On average it can be expected to do no better, and no worse, than random guessing. The apparently disheartening conclusion of NFL depends on the assumption that nothing is known about the structure of a domain; by the same token, the theorem may be taken to point out the importance of assumptions about non-uniformities in a domain for an understanding of the observed successes of search and learning algorithms.

In earlier work [Christensen 2001], we showed that by making a simple tradeoff between functions that in which we are interested those that we aren't, we can on average outperform memorizing random search. We demonstrated there that the crux of the No Free Lunch theorem lies in the vast size of the function set considered, and as a result, the constraints implied by it are rather modest.

The existence and effect of the No Free Lunch theorem is not restricted to numerical search, of course. It applies equally well to genetic programming systems as well, and therefore to programming. Langdon and Poli, in [Langdon 1998], introduce the idea of GP-hard and GP-easy problems. We will consider this distinction in some depth later on in this chapter; for now, it suffices to note that problems for which random search beats genetic programming are called "GP-hard". To discuss the interaction of the No Free Lunch theorem and genetic programming, we will require both an enumeration of function trees (hence, programs) and efficient equations for counting the number of trees of a given size.

Generation and Enumeration of Function Trees

We require an indexing function for function trees to be able to argue about NFL and enumeration difficulty. This section follows closely the pioneering work on enumeration of function trees for genetic programming by Langdon and Poli [Langdon 1998]. In a very significant way, this chapter follows on extends the work first done by these authors in 1998. The simplest case for tree enumeration is the case where there is exactly one main function tree. Let us consider a prototypical GP problem: artificial ant on the Santa Fe Trail [Koza 1992]. Candidate solutions in this function space are made up of nodes taken from the following function and terminal set:

- Nullary functions (terminals): Left, Right, Move
- Unary functions: none
- Binary functions: IfFoodAhead, Progn2
- Trinary functions: Progn3

Define the arity vector $\mathbf{a}$ as the vector which records the number of nodes of a given arity for a problem. For Santa Fe Trail, $\mathbf{a} = [3,0,2,1]$. Let m be the maximum arity node of the problem. Without size limits, the number of trees of a given size is infinite. Of course, memory will limit program size for any real GP system, so we can, in practice, limit the size to some maximum size $s_{\max}$ without loss of generality. We wish to associate with each tree an index *i* that provides its position in a canonical ordering of trees of a given problem.

We use a four-parameter technique to index trees, where the four parameters are:

- size *s*
- configuration vector *c* - that is, how many nodes exist of which arity
- geometry vector *g* - that is, what is the exact shape of the tree under consideration
- labelling vector *l* - that is, which nodes are assigned to which arity nodes

We will show how to generate each term in turn. The configuration vector *c* can be enumerated recursively by the greedy algorithm ENUMERATE-CONFIGURATIONS, which attempts to place as many high-arity nodes as possible when filling. We take the vectors *a*, *c*, *g*, *l* to be zero-indexed, to simplify array accessing and labelling.

Algorithm 1a: ENUMERATE-CONFIGURATIONS

Input: size *s*, arityVector *a*, and configurationStack *configStack*
Output: *configStack*, filled with all the legal configurations for the given size

```
    $c \leftarrow vector(m)$
    $configStack \leftarrow stack()$
    ENUMERATE-CONFIGURATIONS-HELPER $s - 1, m - 1, c, a, configStack$
```

Algorithm 1b: ENUMERATE-CONFIGURATIONS-HELPER

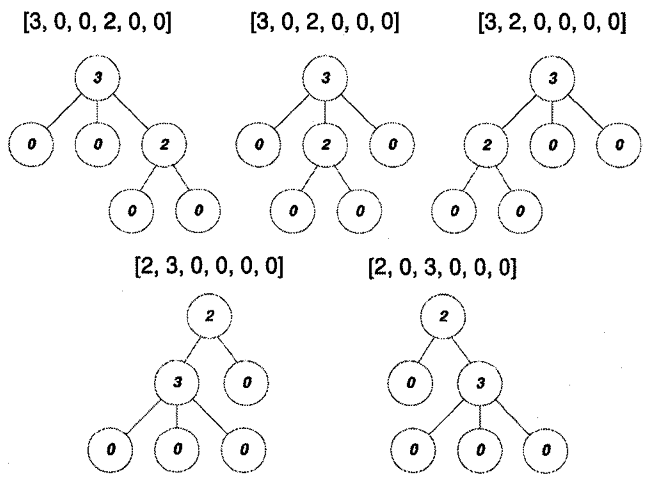
Input: nodes left *n*, index *i*, configuration *c*, arityVector *a*, and configurationStack *configStack*
Output: none - configurations are pushed into *configStack*

```
if $i > 1$ then                              -- allocate binary, trinary, etc. nodes
    if $a_i > 0$ then
        $c_i \leftarrow \left\lfloor \frac{n}{i} \right\rfloor$
        $n_{left} \leftarrow n - i * c_i$
    else
        $c_i \leftarrow 0$
        $n_{left} \leftarrow n$
    end if
    while $c_i \geq 0$ do
        ENUMERATE-CONFIGURATIONS-HELPER $n_{left}$, $i$ - 1, $c$, $a$, $configStack$
        $c_i \leftarrow c_i - 1$
        if $c_i \geq 0$ then $n_{left} \leftarrow n_{left} + i$
    end while
else                                         -- allocate unary and nullary nodes
    if $a_i > 0$ or $n_{left} = 0$ then
        $c_1 \leftarrow n_{left}$
        $c_0 \leftarrow s - \sum_{i=1}^{m} c_i$
        $configStack$.push($c$)
    end if
end if
```

To see an instance of this algorithm, suppose that we want all configurations for Santa Fe trail of size 12. ENUMERATE-CONFIGURATIONS generates the three legal configurations, namely $c \in \{[9,0,0,4],[8,0,3,2],[7,0,6,0]\}$. The configuration vector $c = [9,0,0,4]$ means that there are 4 trinary nodes (of arity 3), and 9 nullary nodes (or terminals, of arity 0). We can then order configurations by the order in which ENUMERATE-CONFIGURATIONS generates them.

Once we have the configuration vector for a tree, we then need its geometry vector. For instance, suppose that we have the configuration vector $c = [4,0,1,1]$. There are 5 distinct geometries available to us, as shown in Fig. 1.

Figure 1



The five tree geometries that use the configuration vector $c = [4,0,1,1]$. Dyck words, known here as the geometry vector g, are given above the trees in question.

As we can see in Fig. 1, the geometry vector g is simply the vector of the arities of nodes encountered while performing a preorder traversal of the tree. This can be used in reverse to generate a tree from a given geometry vector. What we call "geometry vector" in this work is also known as the Dyck word for the given tree [von Dyck 1882]. There is an elegant and efficient algorithm for randomly generating a geometry uniformly from a given configuration [Alonzo 1995]. It works by randomly permuting a generic geometry vector until a valid tree is obtained. It is fully described in GENERATE-RANDOM-GEOMETRY. A valid tree is verified by the subroutine IS-LEGAL-TREE: it is the only geometry where the tree is terminated; that is, it has no dangling edges and all the nodes are used. Each node of arity i has one in-edge and i out-edges, so the number of dangling nodes after adding a new node at the end of the tree increases by i - 1. To make the algorithm uniform, we begin with an imaginary edge dangling down to the root node. If at any point we end up with no free edges before we have gone through all the nodes, the given geometry vector is not valid. As it happens, if we consider all cyclical rotations of a fixed putative geometry vector, there is always exactly one rotation that results in a valid tree. Suppose we randomly generate the putative geometry vector $g = [0,0,0,2,3,0]$. If we try to lay out a tree from this vector, we immediately terminate the tree after the first node. If we rotate all the nodes cyclically to get $g = [2,3,0,0,0,0]$, however, then we will have a properly terminated tree.

Algorithm 2a: GENERATE-RANDOM-GEOMETRY

Input: configuration *c*
Output: a random geometry *g*

```
$g \leftarrow vector(n)$
$j \leftarrow 0$
for $i$ from 0 to $m$ do
   for $k$ from 0 to $c_i$ do
  $g_j \leftarrow i$
  $j \leftarrow j+1$
   od
od
do
   PERMUTE-RANDOMLY $g$
while not IS-LEGAL-TREE $g$
```

Algorithm 2b: IS-LEGAL-TREE

Input: a putative geometry g
Output: true if g represents a valid (closed) tree; false otherwise

```
  $j \leftarrow 0$
  $danglingNodes \leftarrow 1$
  for $i$ from 0 to $n$ - 1 do
     $danglingNodes \leftarrow danglingNodes + g_i - 1$
 if $danglingNodes < 0$ then return false
  od
  return true
```

It is not too difficult to show that there is exactly one rotation that gives a valid tree, and we will get a nice equation for the number of legal geometries for a given configuration out of the work. Clearly, there must be at least one such legal rotation, otherwise there would be a fixed set of nodes that could not be arranged to form a fully-terminated tree. The proof that there must be no more than one is a bit less intuitive. Suppose that there are two such geometries – call them g_1 and g_2. They must both form legal trees under rotation of the geometry vector. Notice that rotation of a geometry vector does not change the relative order of nodes in a pre-order traversal; that is, it preserves subtrees. Lay out the nodes as for g_1, with an edge leading to the root node. There are then n nodes in the fully formed subtree, with n in-edges to them. Each rotation of the geometry vector of g_1 can be identified with choosing a different particular node to be the root of the tree. However, any node other than the actual tree root will result in a smaller tree than g_1 does, and so it will not be properly terminated. There is therefore exactly one valid rotation of a randomly permuted geometry vector which forms a closed tree using all n nodes. With this proof, we can count the number of trees of a given geometry using (1). This is simply the number of ways to permute of a random geometry vector, computed using the permutation-with-duplicates relation, divided by a factor of n for the single degree of freedom used in the rotation.

$$geometries(c) = \frac{(n-1)!}{\prod_{i=0}^{m} c_i!} \tag{1}$$

Unfortunately, there appears to be no elegant algorithm to generate *all* legal geometry vectors for a given configuration. For tree generation purposes, we will need such an algorithm. We present here a correct algorithm, ENUMERATE-GEOMETRIES, which satisfies this task. It uses a local version of a configuration that we term an *allocation*. A given allocation records the number of nodes of which arity are allocated to each subtree of a given node. For instance, for a tree with configuration $c = [10,0,1,3]$, we may allocate the topmost node to be a node with arity 3. We are then left with nodes having the configuration $[10,0,1,2]$ to allocate to our children. ENUMERATE-GEOMETRIES calls an auxiliary function, ENUMERATE-SUBTREE-ALLOCATIONS, to make a list of all the legal ways to distribute these subtrees among the three children of the root node. ENUMERATE-SUBTREE-ALLOCATIONS, in turn, calls GET-SUBSETS to get a list of all the ways that a given configuration can be allocated to a single child. Nullary nodes are neglected for this algorithm, as they are required only to terminate the subtrees, and provide no degrees of freedom in tree geometry allocation. For the configuration $[10,0,1,2]$, there are 6 such ways, namely $\{[x,0,1,2],[x,0,0,2],[x,0,1,1],[x,0,0,1],[x,0,1,0],[x,0,0,0]\}$. ENUMERATE-SUBTREE-ALLOCATIONS then iterates through each of these configurations, allocating the configuration to the first subtree of our root node. It then calls itself recursively to try all the allocations of the remaining children, if any. For instance, we might allocate

$[x,0,0,2]$ to the first child, leaving $[x,0,1,0]$ for the other two children. This latter has only two possible ways, namely $\{[x,0,1,0],[x,0,0,0]\}$. The choice of the second child forces the hand of the third child, by the law of conservation of nodes. Since we have termed a complete allocation of nodes to a subtree an *allocation,* we can say that two allocations are thus generated from the above: $[[x,0,0,2],[x,0,1,0],[x,0,0,0]]$ and $[[x,0,0,2],[x,0,0,0],[x,0,0,1]]$. These allocations are then used recursively to allocate trees all the way down to tree termination by nullary nodes. In the following algorithms, we use stacks for efficiency; however, lists are probably a more appropriate choice.

Algorithm 3a: ENUMERATE-GEOMETRIES

Input: configuration *c*, and geometry stack *geomStack*
Output: *geomStack*, filled with all legal geometries for the given configuration in a deterministic order

```
treeSoFar ← stack()
ENUMERATE-GEOMETRIES-HELPER geomStack, treeSoFar, c
```

Algorithm 3b: ENUMERATE-GEOMETRIES-HELPER

Input: geometry stack *geomStack*, tree stack *treeSoFar*, and configuration *c*
Output: none - geometries are pushed into *geomStack*

```
addedAny ← false
for i = m downto 1 do
if $c_i > 0$ then
  nextTree ← stack(treeSoFar)           -- clone the tree so far
  nextTree.push(i)                      -- and add a high-arity node
  addedAny ← true
  $c_i \leftarrow c_i - 1$               -- use up one of the highest-arity nodes
  allocations ← ENUMERATE-SUBTREE-ALLOCATIONS c, i
  for each allocation in allocations do
    geometries ← stack()
    for each childConfiguration in allocation do
      subtree ← stack()
      subGeomStack ← stack()
      ENUMERATE-GEOMETRIES-HELPER subGeomStack, subtree,
                                                  childConfiguration
          geometries.push(subGeomStack)
        end while
        $counts_j \leftarrow length(geometries_j) - 1 \; \forall j = 1 \ldots length(geometries)$
        $subsetList \leftarrow$ GET-SUBSETS $counts$
        for each subset in subsets do
          $thisGeometry \leftarrow stack(thisTree)$
          for j from 1 to length(subset) do
            $g \leftarrow geometries_{j,subset_j}$
            for k from 1 to length(g) do
              thisGeometry.push( $g_k$ )
            end for
          end for
        end for
      end while
    end for
  if $a_i > 0$ then
    $c_i \leftarrow \left\lfloor \frac{n}{i} \right\rfloor$
    $n_{left} \leftarrow n - i * c_i$
  else
    $c_i \leftarrow 0$
    $n_{left} \leftarrow n$
  while $c_i \geq 0$ do
    ENUMERATE-CONFIGURATIONS-HELPER $n_{left}$, i - 1, c, a, configStack
    $c_i \leftarrow c_i - 1$
    if $c_i \geq 0$ then $n_{left} \leftarrow n_{left} + i$
  end while
else                              -- allocate unary and nullary nodes
  if $a_i > 0$ or $n_{left} = 0$ then
    $c_1 \leftarrow n_{left}$
    $c_0 \leftarrow s - \sum_{i=1}^{m} c_i$
    configStack.push(c)
  end if
```

Algorithm 3c: ENUMERATE-SUBTREE-ALLOCATIONS

Input: configuration *c* and slots available counter *slots*
Output: a stack of legal subtree allocations *allocationStack*

```
    $allocationStack \leftarrow stack()$
    $allocation \leftarrow stack()$
    ENUMERATE-SUBTREE-ALLOCATION-HELPER $allocationStack, allocation, c, slots$
    return $allocationStack$
```

Algorithm 3d: ENUMERATE-SUBTREE-ALLOCATION-HELPER

Input: *allocationStack*, current subtree allocation *allocation*, configuration *c*,
and slots available counter *slots*
Output: none; *allocationStack* is filled with legal subtree allocations

```
if $slots > 1$ then
  $subsetList \leftarrow$ GET-SUBSETS $c$
  for $subset$ in $subsetList$ do
    $allocation$.push($subset$)
    $leftover_i \leftarrow c_i - subset_i \forall i = 0 \ldots m$
    ENUMERATE-SUBTREE-ALLOCATION-HELPER $allocationStack, allocation,$
                                        $leftover, slots - 1$
    $allocation$.pop()
  end for
else
  $newAllocation \leftarrow stack(allocation)$
  $newAllocation$.push($c$)
  $allocationStack$.push($newAllocation$)
end if
```

Algorithm 3e: GET-SUBSETS

Input: a configuration *c*
Output: a list of all legal subsets of *c*

```
    $subsetList \leftarrow list()$
    for all configurations $t$ with $t_i = 0 \ldots c_i \forall i = 1 \ldots m$ do
      $subsetList$.addEnd($t$)
    end for
    return $subsetList$
```

Once we have the geometry of the tree in question, the only thing remaining is to allocate the specific node choice themselves. Fortunately, this is the simplest of the four choices, as we can simply number each node in a given tree. Define the labelling vector l as a parallel vector to g, where each l_i identifies which of the arity-g_i nodes the tree will use. There are c_i nodes in the tree of arity i, and a_i nodes to choose from in the function set; the number of labellings in (2) follows. Interestingly, this number is independent of the specific geometry g chosen.

$$labellings(c,a) = \prod_{i=0}^{m} c_i^{\mathbf{a}_i} \tag{2}$$

We can now write down an expression for the number of trees of a given size. Define $C(n,\mathbf{a})$ as the set of all legal configurations for a problem's arity vector **a** and number of nodes n. (3) gives the number of trees for a given number of nodes n by combining (1) and (2).

$$trees(n,\mathbf{a}) = \sum_{c \in C(n,\mathbf{a})} \left(\frac{(n-1)!}{\prod_{i=0}^{m} c_i!} \prod_{i=0}^{m} c_i^{\mathbf{a}_i} \right) \tag{3}$$

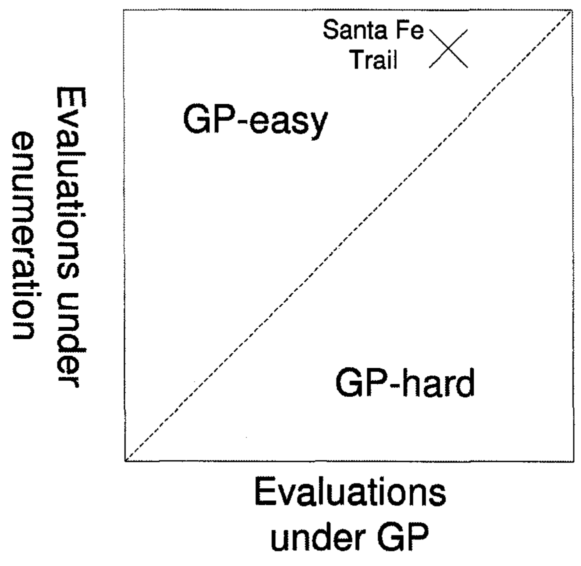
The No Free Lunch theorem for Genetic Programming

Equipped with both an enumeration over trees and a counting of the number of such trees, we can restate the No Free Lunch theorem for genetic programming. Let $T(n,\mathbf{a})$ be the set of trees of size n using arity vector **a**, and let $T^{+}(n,\mathbf{a}) = \bigcup_{i=1}^{n} T(n,\mathbf{a})$ be the set of trees of size n or smaller using **a**. The No Free Lunch theorem for function trees is given in (4), where ϕ is any measure of algorithm performance (see [Wolpert 1996] for details on such measures), A is an algorithm under test, k is an arbitrary constant, and $F : T^{+}(n, \mathbf{a}) \rightarrow \Re$ is any function set that forms a permutation set over the set of trees.

$$\sum_{f \in F} \phi_A(f) \geq k \qquad (4)$$

Bill Langdon and Riccardo Poli introduce the very productive and important idea of a GP-hard problem in [Langdon 1998]. A GP-hard problem is defined as a problem for which genetic programming performs worse than enumeration, on average. He uses the computational effort statistic introduced by Koza for the definition of "worse". [Langdon 1998] notes that Artificial Ant on the Santa Fe trail with 600 time steps to solve the problem is GP-hard. Since k from (4) is equal to the average performance of functions on enumeration, this can be viewed as a sort of specialized case of the No Free Lunch theorem. That is, genetic programming partitions the space of possible functions into two sets: GP_{easy}, those for which GP outperforms enumeration; and GP_{hard}, those for which enumeration outperforms GP. The surprising result, for many of those of us used to numerical optimization, is that such a simple problem as Artificial Ant on the Santa Fe trail is in GP_{hard}. The results of Chaper 4 suggest that the No Free Lunch is a ridiculously weak constraint on optimization algorithm performance. Are we to believe that the types of problems that genetic programming attacks are really so much harder than numerical optimization tasks? This seems counterintuitive, to say the least! This result is all the more surprising given that Artificial Ant on the Santa Fe trail appears to be such an easy problem, from a human programmer's point of view. Admittedly, there are significant differences between the problem that humans solve when programming and that posed of a genetic programming system, as raised in the allegory of the GP room in Appendix 1. Still, given the fundamental weakness of the NFL theorem shown in [Christensen 2001], we might expect more of genetic programming than this. Fig. 2 shows this space graphically, with a dotted line indicating the "frontier" between GP-easy and GP-hard problems. Our overriding goal in this thesis is to use some tricks derived from how human programmers solve problems to improve the performance of genetic programming. Indeed, the title of this thesis, "Towards Scalable Genetic Programming," indicates that we intend to move in the direction of making algorithms that work for arbitrary problem size, much as human programmers do, not merely on fixed-size problems. In approaching this task, we will require a significant infrastructure, consisting of both better and more accurate analysis of EC performance; and a set of utility algorithms which will be useful in improving GP performance. With the reader's indulgence, in the remainder of this chapter we will detail one method of improving performance. It happens that systematically considering small GP trees to automatically induce subroutines useful for a candidate problem can greatly improve computational efficiency. We will discuss more sophisticated methods of improving performance and generating subroutines in Chapter 7.

Figure 2



A depiction of the division of the set of possible problems into GP-easy and GP-hard problems, as indexed by the number of evaluations required on average to find a solution. The density of problems on the GP-easy and GP-hard sides of the line are not necessarily uniform. A cross indicates the approximate location of the Artificial Ant on the Santa Fe trail problem. Justification for this position is given in the following subsections.

Analysis of Artificial Ant on the Santa Fe Trail

We can actually expand on Fig. 2. In [Christensen 2002], we performed $27\,755$ runs of artifical ant on the Santa Fe trail with standard parameters to validate the computational effort statistic. There, we found that for a problem difficulty of 600 steps, the work required to give a 99% chance of successfully solving this problem with the best possible choice of generation number and $M = 500$ individuals lies within $[474\,000, 485\,000]$, with 95% confidence. By using the enumeration algorithms above, we can independently confirm the results of Langdon in [Langdon 1998]. First we will need a way of fairly comparing enumeration-style algorithms with a stochastic algorithm like genetic programming. If we enumerate the outcomes for a genetic programming function tree, there must be a minimum size tree that solves the problem. Call this minimum size $n_{\min}$ for the problem; this is essentially the Kolmogorov complexity in the context of genetic programming with a given function set. There will often be more than one solution of the minimum size. In this case, we can imagine a memorizing random search algorithm which proceeds as MEMORIZING-RANDOM-TREE-SEARCH.

Algorithm 4: MEMORIZING-RANDOM-TREE-SEARCH

Input: an oracle *O*, which takes a function tree and answers *true* if the function tree is correct and *false* otherwise
Output: a valid tree *t* that solves the problem

```
$n \leftarrow 1$
$usedTrees \leftarrow \{\}$
for ever
  do
     $t \leftarrow$ GENERATE-RANDOM-TREE $n$, $\mathbf{a}$
  while $t \in usedTrees$
  if $O(t)$ then return $t$
  if $|usedTrees| = trees(n, \mathbf{a})$ then
       $n \leftarrow n+1$
       $usedTrees \leftarrow \{\}$
  end if
end for
```

For this algorithm, we can compute the number of function evaluations required to solve a problem with a 99% probability of success. Suppose that there are exactly $s(n)$ successful programs of size *n*. The odds of success are clearly 0 for all trees with sizes from 1 through $n_{\min}-1$; they increase to 1 after the algorithm has explored all the trees of size $n_{\min}$. It remains to find the odds of success at intermediate numbers of nodes chosen.

Suppose that there are exactly 2 solutions in 10 trees. In this case, Fig. 3 enumerates all possibilities for the number of trees which must be examined before a solution is found.

Figure 3

	First solution									
Second solution	1	2	3	4	5	6	7	8	9	10
1		1	1	1	1	1	1	1	1	1
2	1		2	2	2	2	2	2	2	2
3	1	2		3	3	3	3	3	3	3
4	1	2	3		4	4	4	4	4	4
5	1	2	3	4		5	5	5	5	5
6	1	2	3	4	5		6	6	6	6
7	1	2	3	4	5	6		7	7	7
8	1	2	3	4	5	6	7		8	8
9	1	2	3	4	5	6	7	8		9
10	1	2	3	4	5	6	7	8	9	

An enumeration of the selection at which success is first obtained in 10 trials with 2 successes present. We have chosen two unequal positions uniformly at random for the two solutions.

From this table, the pattern is clear – the odds of success in k trials or fewer is given by the ratio of the area of the square excluding k rows and columns to the area of the entire square excluding diagonals. Mathematically, this is described by (6a), where $t_{success}$ is the trial at first success.

$$P(t_{success} \le k) = 1 - \frac{(n-k)(n-k-1)}{n(n-1)} \tag{6a}$$

This generalizes in the case of s successes to the difference of two s-dimensional hyperboxes excluding all diagonals. (6b) gives the relevant formula, which simplifies to (6c). Unfortunately, we are looking for the trial at first success for which the probability $P(t_{success} \le k)$ is 99%, which is a little challenging.

$$P(t_{success} \le k) = 1 - \frac{(n-k)!/(n-k-s)!}{n!/(n-s)!} \tag{6b}$$

$$P(t_{success} \le k) = 1 - \frac{(n-k)!(n-s)!}{n!(n-k-s)!} \tag{6c}$$

Rather than go into an exact formula using root finding over the logarithms of factorials, there is a trick we can use. If $s << n$, then we can neglect the diagonal terms, which gives us the more elegant and solvable (6d).

$$P(t_{success} \le k) \cong 1 - \left(\frac{(n-k)}{n}\right)^s \tag{6d}$$

This can be solved for *k* to get (6e), which is in closed form.

$$k \cong n\left(1 - \left(1 - P(t_{success} \le k)\right)^{1/s}\right) \tag{6e}$$

After generating some data for the artificial ant on the Santa Fe Trail with 600 time-steps, we get the results shown in Fig. 4.

Figure 4

n	***trees*(*n*, a)**	***s*(*n*)**
≤ 7	5 043	0
8	20 412	0
9	95 256	0
10	516 132	0
11	2 554 416	12
12	13 712 490	48
13	71 521 461	470

Number of trees $trees(n, \mathbf{a})$ and successes $s(n)$ for different tree sizes *n*, for the standard problem for artificial ant on the Santa Fe Trail with 600 time steps.

From Fig. 4, we can say that the computational effort to 99% success using MEMORIZING-RANDOM-TREE-SEARCH is the number of trees of size 10 and lower, plus the appropriate value of *k* from (6e) substituting 0.99 for $P(t_{success} \le k)$. The appropriate value of *k* from (6e) is 814112. We performed a check using this *k* using the exact equation (6c) to validate our approximation of neglecting the diagonal terms. The closest obesrved probability to 99% success rate occurs when $k = 814\,111$, which is off by 1 fitness evaluation from our approximation.

Therefore, the computational effort of artificial ant on the Santa Fe trail is $636\,843 + 814\,111 = 1\,450\,954$. This differs from the optimal value of $460\,000$ given in [Langdon 1998], as the authors estimate the work required to solve the problem assuming that only solutions of the optimal size. While following the convention of genetic programming where preliminary work is neglected in estimating computational effort, this value ignores the significant work required to determine that solutions of size 18 are optimal, which probably should not be neglected in a fair comparison. The nice thing about MEMORIZING-RANDOM-TREE-SEARCH is that the computational effort statistic derived for it is insensitive to the specific ordering of configuration, geometry and labelling used during tree generation.

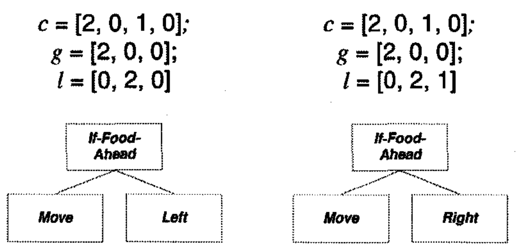
Using this value for computational effort, we can see that with the right parameter settings, artificial ant on the Santa Fe is not strictly GP-hard; however, it is close. Indeed, the traditional definition of computation effort is a little unfair as well, as we do not know which population size M and number of generations G to use. Since the *a priori* probability of success at the optimal choice of parameters is 8.7%, we are unlikely to encounter a single success while testing these parameter settings. A fair comparison would involve using an algorithm such as the progressive-grid algorithm introduced in Chapter 6. This will, of course, inflate the actual computational effort significantly, by a factor of roughly 4 or so. The fair computational effort for genetic programming would then be roughly $1\,800\,000$, very close to the fair computational effort required by enumeration. The claim that artificial ant on the Santa Fe trail is GP-hard is therefore justified by our analysis.

Improving on Standard Genetic Programming

Faced with the existence of GP-hard problems, is there anything that we can do? One option available to us, which will explored further in Chapter 7, would be to add in some successful subroutines derived from the analysis of successful small trees. To give a taste of this procedure, suppose that we use the algorithms presented above to enumerate all trees of size 1, 2, 3, etc. For the artificial ant on the Santa Fe trail problem, there are no trees of size 2, so we can begin with trees of size 3. There are two high-performing trees of size 3, namely the two trees shown in Fig. 5.

Engaging in a decidedly human behaviour, we can look for commonalities in these two high-performing subtrees. Fortunately, the common subtree is easily identified - the two subtrees differ only in a single node.

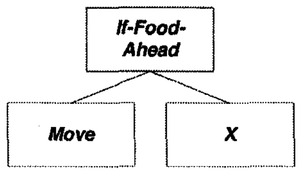
Figure 5



The two highest-performing trees of size 3 or lower on artificial ant on the Santa Fe trail. The three vectors *c*, *g*, and *l* generating each tree are shown for reference; they are defined in the main text.

Abstracting this subtree out, we get the tree of Fig. 6. This abstraction is, in fact, very close to a conventional subroutine. Suppose that we add the tree of Fig. 6 back into the original problem as a new unary function node, If-Food-Ahead-Move(X). If we then run MEMORIZING-RANDOM-TREE-SEARCH on the extended problem, we get the performance and trees shown in Fig. 7.

Figure 6



An abstraction of the two highest-performing trees of size 3 for the artificial ant on the Santa Fe trail problem. The "X" marks the node that becomes a free parameter in the new subroutine.

Figure 7

n	***trees*(*n*, a)**	***s*(*n*)**
≤7	15 771	0
8	74 091	0
9	432 183	12
10	2 573 859	142
11	15 538 719	1 172
12	94 936 152	
13	585 952 788	

Number of trees $trees(n,\mathbf{a})$ and successes $s(n)$ for different tree sizes n, for the artificial ant on the Santa Fe Trail problem with 600 time steps using the additional unary function node If-Food-Ahead-Move(X) shown in Fig. 6.

From Fig. 7, we can say that the computational effort to 99% success using MEMORIZING-RANDOM-TREE-SEARCH on this revised problem is the number of trees of size 8 and lower, plus the appropriate value of k from (6e). Substituting into (6e) gives $k = 137\,740$. Therefore, the computational effort of artificial ant on the Santa Fe trail for this restated version of the problem is $89\,862 + 137\,740 = 227\,602$. To be fair, we must add the number of trees of size 3 or smaller that we examined in the first place, which number 21. The total computational effort is then $227\,623$, which is a factor of 6.37 easier than the normal computational effort. We can repeat this procedure - the relevant fitnesses of the smallest trees are given in Fig. 8.

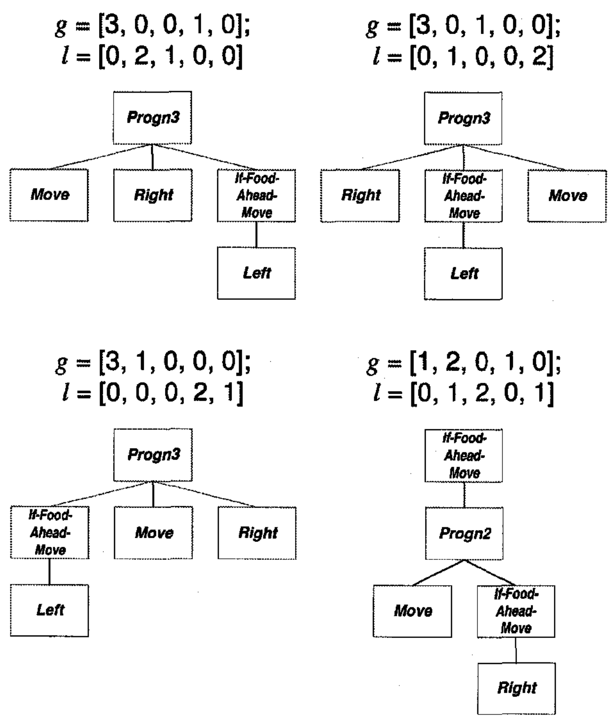
Figure 8

n	**trees(*n*, a)**	***best fitness***	***trees w/ best fit.***
2	3	78	2
3	21	78	4
4	84	78	16
5	435	65	4
6	2 343	51	1

Number of trees $trees(n, \mathbf{a})$, best fitnesses, and number of trees with the best fitness value for different tree sizes *n*, for the artificial ant on the Santa Fe Trail problem with 600 time steps using the additional unary function node If-Food-Ahead-Move(*X*) shown in Fig. 6.

Looking at the 4 trees with exceptional fitness of size 5 in this new format, shown in Fig. 9, we quickly see that they do not share any geometry vector, much less labelling.

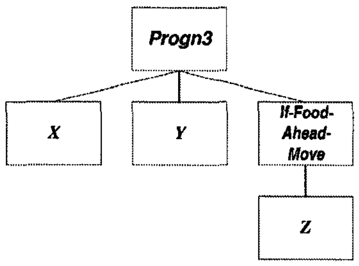
Figure 9



The four highest-performing trees of size 5 or lower on the Santa Fe trail problem, with the added function If-Food-Ahead-Move(*X*). The geometry vector *g* and the labelling vetor *l* that generate each tree are shown for reference. The configuration vector *c* is omitted for space reasons; it can readily be recovered from the geometry vector *g* by counting.

Let us, for argument's sake, choose the first tree to generalize into a subroutine. Without any additional knowledge, let us take all three terminals and make them parameters of an additional function. Call this function If-Food-Ahead-Move-3(*X*, *Y*, *Z*); it is diagrammed in Fig. 10.

Figure 10



An abstraction of one of the four highest-performing trees of size 5 for the Santa Fe trail problem. "X", "Y" and "Z" mark nodes that become free parameters in the subroutine, If-Food-Ahead-Move-3(*X*, *Y*, *Z*).

As before, we add the tree of Fig. 10 back into the revised problem as a new trinary function node. If we then run MEMORIZING-RANDOM-TREE-SEARCH on the extended problem, we get the performance and trees shown in Fig. 11.

Figure 11

n	*trees*(*n*, a)	*s*(*n*)
≤ 6	4 104	0
7	20 469	4
8	127 767	38
9	826 059	280
10	5 372 571	many

Number of trees $trees(n, \mathbf{a})$ and successes $s(n)$ for different tree sizes *n*, for the artificial ant on the Santa Fe Trail problem with 600 time steps using the additional unary function node If-Food-Ahead-Move(*X*) shown in Fig. 6 and the trinary function node If-Food-Ahead-Move-3(*X*, *Y*, *Z*) shown in Fig. 10.

From Fig. 11, the computational effort to 99% success using MEMORIZING-RANDOM-TREE-SEARCH on this revised problem is the number of trees of size 6 and lower, plus the appropriate value of k from (6e). Substituting into (6e) gives $k = 13\,996$. Therefore, the computational effort of artificial ant on the Santa Fe trail for this restated version of the problem is $4\,104 + 13\,996 = 18\,100$. To be fair, we must add the number of trees of size 5 or smaller that we examined previously, in doing the two previous steps. There are $21 + 543 = 564$ such, giving a total computational effort of $18\,664$. This value is a factor of 12.2 better than the previous, and an impressive 77.7 times better than the naïve approach!

The possibility remains that this is a fluke, that we were uncharacteristically lucky in choosing the first of the four trees to generalize in Fig. 9. Accordingly, in Fig. 12 we have illustrated key statistics from performing all-terminal generalizations from *each* of these four superior trees.

Figure 12

Variant	*Name*	$n_{\min}$	$\sum_{n=1}^{n_{\min}-1} trees(n,\mathrm{a})$	$trees(n_{\min},\mathrm{a})$	$s(n_{\min})$	CE_{99}	*ratio*
1	After	7	4 104	20 469	4	18 664	77.7
2	In	8	24 573	127 767	14	60 952	23.8
3	Before	8	24 573	127 767	8	81 055	17.9
4	Binary	8	32 286	171 615	35	54 008	26.9

Some statistics for the artificial ant on the Santa Fe Trail problem with 600 time steps using the additional unary function node If-Food-Ahead-Move(X) shown in Fig. 6 and for the each of the four possible generalizations of the trees shown in Fig. 9. The column headings refer to the number of trees $trees(n,\mathrm{a})$, minimum tree size with perfect solutions $n_{\min}$, successes at minimum size $s(n_{\min})$, 99% computational effort CE_{99}, and ratio of effort to the computational effort for the normal function set. The values for CE_{99} shown here include the work required to generate all preliminary trees.

As Fig. 12 illustrates, in each case the performance is better than that of the single-subroutine variant. The question arises: how to automatically choose which tree to make a subroutine from? One efficient method is to consider the best performance of all trees of size 2, 3, 4, … until we beat all previous small-tree records, or until we are generating too many trees for the potential benefit that we would gain. In Fig. 13, we show the best fitness at a given size and number of such trees with best fitness for all trees of sizes 2 through 5 for each of the function sets made reference to thus far. We skip trees of size 1 since no nullary functions were considered in this series.

Figure 13

n	*Normal*	*+ IFAM*	*+ IFAM + Var. 1*	*+ IFAM + Var. 2*	*+ IFAM + Var. 3*	*+ IFAM + Var. 4*
2		78 x2	78 x2	78 x2	78 x2	78 x2
3	78 x2	78 x4	78 x4	78 x4	78 x4	65 x1
4	86 x6	78 x16	65 x1	65 x1	65 x1	65 x3
5	78 x16	65 x4	51 x1	59 x2	51 x1	43 x1

Best fitness at a given size *n*, and number of trees with this fitness, for each of the function set augmentations discussed in this chapter. In the Santa Fe trail problem, smaller scores are better, and 0 is a perfect score. Fitnesses are shown before the 'x' character; number of trees follow. IFAM refers to the function If-Food-Ahead-Move(*X*); the four variants labelled "Var. 1", "Var. 2" and so on refer to the indexed variants shown in Fig. 12.

Fig. 13 offers the possibility of determining which trees to generalize efficiently. When we add a new subroutine of arity a, the new subroutine has the possibility of improving the performance of all small trees of size $a + 2$ and larger. Since a subroutine made from a tree with arity a can exactly duplicate the original tree at size $a + 1$, it will take a tree of size at least $a + 2$ to *improve* on the performance. If we compare the performance of superlative small trees, we can decide which path to pursue using a simple greedy algorithm, SYSTEMATIC-SUBROUTINE-GENERALIZATION.

Algorithm 5a: SYSTEMATIC-SUBROUTINE-GENERALIZATION

Input: an oracle *O*, which takes a function tree using zero or more subroutines and answers a numerical fitness value $O(t)$, where 0 is optimal, and larger numbers are worse;
a set of "built-in" function terminals *F*, which operate in the fitness space scored by *O*; and
a function ENUMERATE-TREES, which returns the set of all trees of size *n* using a given function set and subroutine set.

Output: a valid tree *t* that solves the problem, and a set of subroutines *S* which have been shown to improve performance on small trees

```
$S \leftarrow \{\}$
$n \leftarrow 1$
do
    $n \leftarrow n+1$
    $T \leftarrow$ ENUMERATE-TREES $n, F \cup S$
while $|T| = 0$
$[Heroes, bestScore] \leftarrow$ BEST-TREES $n, F \cup S, O$
for ever
    $s \leftarrow$ UNIFY-TREES $Heroes$
    if $s \neq nil$ then
        $Best \leftarrow$ EXPERIMENT-WITH-SUBROUTINES-TO-GET-BEST $\{s\}, F \cup S, O$
        $NewSubs \leftarrow \{s\}$
    else
        $Best \leftarrow$ EXPERIMENT-WITH-SUBROUTINES-TO-GET-BEST $Heroes, F \cup S, O$
        $NewSubs \leftarrow Best$
    end if
    if $NewSubs = \{\}$ then
        exit for
    end if
    $S \leftarrow S \cup NewSubs$
    $a \leftarrow \min_{s \in NewSubs} |\{i \mid labelling(s)_i = -1\}|$
    $n \leftarrow a+1$
    do
        $n \leftarrow n+1$
        $[Heroes, newBestScore] \leftarrow$ BEST-TREES $n, F \cup S, O$
    while $bestScore = newBestScore$ and not TOO-MUCH-WORK $n, F \cup S, O$
end for
do
    $n \leftarrow n+1$
    $bestScore \leftarrow$ MEMORIZING-RANDOM-TREE-SEARCH $n, F \cup S$
while $bestScore > 0$
```

```
return [$t, S$]
```

Algorithm 5b: Best-Trees

Input: an oracle O, which takes a function tree using zero or more subroutines and answers a numerical fitness value $O(t)$, where 0 is optimal, and larger numbers are worse;
a set of "built-in" function terminals F and a set of added subroutines S, which operate in the fitness space scored by O; and
a function Enumerate-Trees, which returns the set of all trees of size n using a given function set and subroutine set.
Output: a set of best trees *Heroes* and their score *bestScore*

```
$T \leftarrow$ Enumerate-Trees $n, F \cup S$
$Heroes \leftarrow \{\}$
$bestScore \leftarrow \infty$
for each $t \in T$
    if $O(t) < bestScore$ then
        $Heroes \leftarrow \{t\}$
        $bestScore \leftarrow O(t)$
    else
        $Heroes \leftarrow Heroes \cup \{t\}$
    end if
end for
return [$Heroes$, $bestScore$]
```

Algorithm 5c: Unify-Trees

Input: a set of trees T to unify, and a function Make-Tree g, l that makes a tree from the provided geometry and labelling vectors
Output: a single tree t which generalizes all the trees in T if possible, or *nil* if no such generalizing tree exists. We use -1 as a signalling value in the labelling array to mean "make this node a variable".

```
for each $t \in T$
    if $g = nil$ then
        $g \leftarrow geometry(t)$
        $l \leftarrow labelling(t)$
    else if $g \neq geometry(t)$ then
        return $nil$
    else
        for $i$ from 0 to $n$ - 1 do
            if $l_i \neq -1$ and $l_i \neq labelling(t)_i$ then
                if $g_i > 0$ then
                    return $nil$        -- can't unify functions >= 1 arity at present
                else
```

```
                    $l_i = -1$
                end if
            end if
        end for
    end if
end for
return MAKE-TREE $g, l$
```

Algorithm 5d: MAKE-SUBROUTINE-FROM-TREE

Input: a tree *t* to make into a subroutine, and a function MAKE-TREE *g*, *l* that makes a tree from the provided geometry and labelling vectors

Output: a single tree *t* that generalizes all the terminal nodes in *T*. We use -1 as a signalling value in the labelling array to mean "make this node a variable".

```
$g \leftarrow geometry(t)$
$l \leftarrow labelling(t)$
for $i$ from 0 to $n$ - 1 do
    if $g_i = 0$ then
        $l_i = -1$
    end if
end for
return MAKE-TREE $g, l$
```

Algorithm 5e: EXPERIMENT-WITH-SUBROUTINES-TO-GET-BEST

Input: a set of trees *T* to test for "subroutine fitness";
an oracle *O*, which takes a function tree using zero or more subroutines and answers a numerical fitness value $O(t)$; and
a set of "built-in" function terminals *F* and a set of added subroutines *S*, which operate in the fitness space scored by *O*

Output: a set of trees *Candidates* that identifies the most promising subroutines in *T*.

```
$Candidates \leftarrow \{\}$
for $t$ in $T$ do
    $l \leftarrow labelling(t)$
    $a \leftarrow |\{i \mid l_i = -1\}|$
    $n \leftarrow a + 2$
    $biggestN \leftarrow n$
    $bestSpecialSoFar \leftarrow \infty$
    while not TOO-MUCH-WORK $n, F \cup S \cup \{t\}, O$ do
        $[Heroes, bestNormal] \leftarrow$ BEST-TREES $n, F \cup S, O$
        $[Heroes, bestSpecial] \leftarrow$ BEST-TREES $n, F \cup S \cup \{t\}, O$
        if $bestSpecial \le bestNormal$ then
```

```
                if $bestSpecialSoFar = \infty$ then
                    $bestSpecialSoFar \leftarrow bestSpecial$
                    $Candidates \leftarrow Candidates \cup \{t\}$
                    exit while
                else if $bestSpecial < bestSpecialSoFar$ then
                    $Candidates \leftarrow \{t\}$
                    $bestSpecialSoFar \leftarrow bestSpecial$
                    exit while
                else if $bestSpecial = bestSpecialSoFar$ then
                    $Candidates \leftarrow Candidates \cup \{t\}$
                    exit while
                end if
                while $n < biggestN$ do
                    // Compare data at new size vs. existing to look for a new best
                    $n \leftarrow n+1$
                end while
            else if $bestSpecial > bestNormal$ then
                exit while
            end if
            $n \leftarrow n+1$
            $biggestN \leftarrow n$
        end while
    end for
    return $Candidates$
```

This algorithm also requires a work threshold that acts as a cutoff so that the algorithm doesn't spend all its computational effort looking for useful subroutines. For instance, if we set the cutoff at 10 000 evaluations, SYSTEMATIC-SUBROUTINE-GENERALIZATION will solve the Santa Fe trail in 20 573 evaluations, 70.5 times better than the naïve approach. Of course, this subroutine-generating function will be quite useful as a tool for the scientist algorithm introduced in Chapter 2. Alternately, we can exit SYSTEMATIC-SUBROUTINE-GENERALIZATION without calling MEMORIZING-RANDOM-TREE-SEARCH, and use the new subroutines to augment a conventional genetic programming run. An early indication of the performance advantages that can be achieved in this latter case is given in Fig. 14.

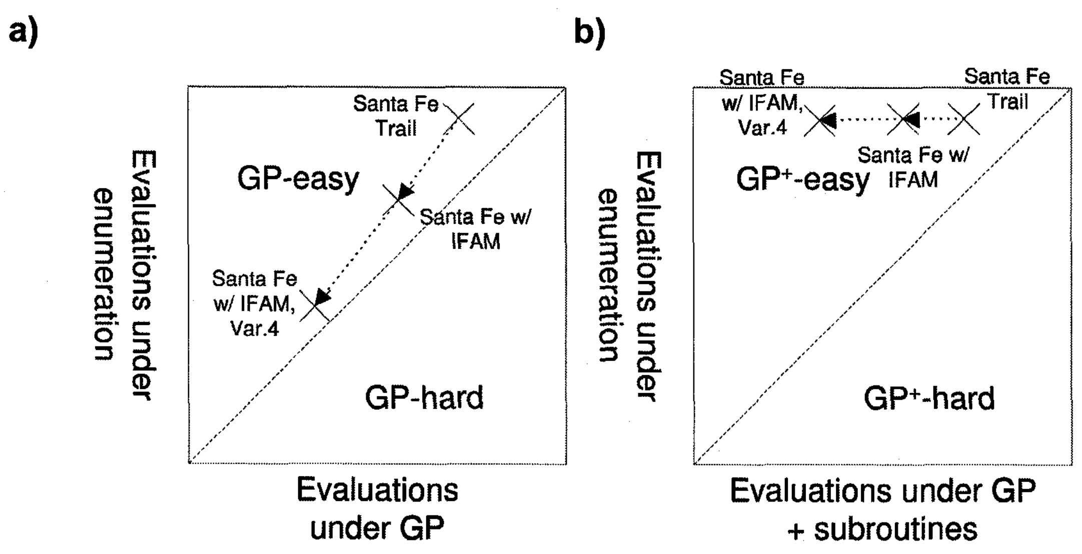
Figure 14

M	G	*Normal*	*+ IFAM*	*+ IFAM + Var. 1*	*+ IFAM + Var. 2*	*+ IFAM + Var. 3*	*+ IFAM + Var. 4*	*+ IFAM + Var. 4 + IFB3*
enumeration		1 450 954	227 623	18 664	60 952	81 055	54 008	20 573
1100	17	440 000						
1000	14		200 000					
250	1						40 000	

Near-optimal computational effort to 99% success for different function sets on the Santa Fe trail problem. The population size M and the number of generations G is given for each computational effort. Each value is based on at least 10 000 independent runs, giving confidence intervals of around 5%. IFAM refers to the function If-Food-Ahead-Move(X); the numbered variants refer to the indexed variants shown in Fig. 12; IFB3 is the third-level induced subroutine derived from Variant 4. We also show the 99% computational effort for Memorizing-Random-Tree-Search, here indicated as "enumeration". These latter numbers are exact.

To bring these developments back to the question of GP-hardness, we can revise Fig. 2 in light of the performance advances made possible by considering small trees, as in Fig. 15.

Figure 15



Two revised depictions of the division of the set of possible problems into GP-easy and GP-hard problems, under the influence of the small-trees subroutine-generating algorithm. In Fig. 15a, the GP-easy and GP-hard sides of the line do not change, but the work required declines significantly. In Fig 15b, we view the function set as fixed, and the subroutine analysis code as part of the genetic programming innovation system.

Chapter 6: Modelling Success Probability in Evolutionary Computing

Introduction

The standard technique for comparison between Genetic Programming techniques is to determine, as best as possible, the best-choice parameter settings for the Genetic Programming system. For the purposes of this section, we use the continuous version of computational effort given in (1), without using a ceiling operator on the number of runs performed.

$$CE_z = I_{\min}(M,z) = \min_i Mi \frac{\ln(1-z)}{\ln(1-P(M,i))} \qquad (1)$$

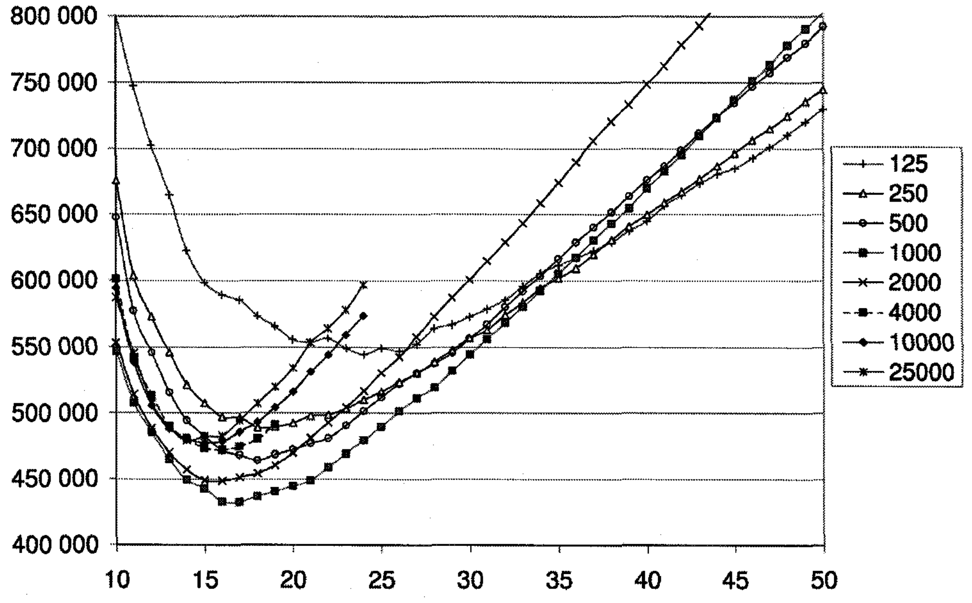
After a good choice for population size and number of generations is found, the performance at these settings is compared to similarly strong settings for a control technique. This computationally costly parameter setting normally forces comparisons between toy problems. We will argue that this is an unrealistic process for any problem of actual interest to humans, as it is too lengthy. A better approach would be more systematic. In this chapter, we will discuss two systematic approaches, the progressive grid algorithm and the model-based grid algorithm. The model-based grid algorithm, which forms a model of success probability given the data, enables superior estimation of unoptimized and perfectly-optimized computational effort. However, developing the model is challenging and involves many runs of the entire process. We will then argue that for a family of GP problems with certain characteristics, a kind of algorithm called a *progressive* GP can give results within a constant factor of the performance of the optimized result.

Computational Effort for Genetic Programming Problems

The computational effort statistic for a problem's performance is used today in much the same way as it was in Koza's original formulation. The derivation is given in Chapter 4; we repeat the equation for the continuous version of the computational effort here as (1), where M is the population size, i is the generation number, and z is the desired probability of success, typically 99%.

In Fig. 1, we show an estimate of the $I_{\min}(M,z)$ curves for the artificial ant on the Santa Fe trail.

Figure 1



Estimated 99% computational effort at each generation for the artificial ant on the Santa Fe trail problem as a function of population size. For each curve, at least 10 000 runs were performed, rendering these data largely immune from the generational effects described in Chapter 4. Error bars are not indicated on this graph, but 95% confidence intervals are at most on the order of 20 000 fitness evaluations around the minima of each curve. Small-number-of-success effects can be seen for the $M = 125$, the $M = 250$, and the $M = 25\ 000$ curves.

Inspection of Fig.1 shows that computational effort is not independent of population size, so we should report the best computational effort when computed over all values of M. (2) gives this version of computational effort.

$$I_{\min}(z) = \min_{M,i} Mi \frac{\ln(1-z)}{\ln(1-P(M,i))} \tag{2}$$

This poses a problem: we cannot minimize over all values of *M* without explicitly evaluating the computational effort with enough precision to discriminate the best value of *M*. Computing this value accurately is a little tricky, so we ask the reader to bear with us for a few paragraphs. The problem comes about from the fact that we need to evaluate data over several values of *M* and final generation number *G* to determine the best parameter settings for subsequent evaluation. Since we can only estimate computational effort from real data, we need a criterion for success of estimation. As a point to agree on, suppose that that we wanted to estimate computational effort so that we know the best parameter settings and ultimate computational effort to within $20\,000$ fitness evaluations. How many fitness evaluations do we, in fact, require to meet this goal?

We will then need to know how densely to sample in population size *M* and final generation number *G*. It is not clear, *a priori*, how many generations we will need to perform so that we can evaluate our runs to get optimal results. It is also not clear how far out in *M* we should go looking for strong candidate parameter settings. Fig. 2 shows the data used to generate the graphs in Fig. 1, as well as data from a few other runs. Fortunately, we can use the actual data of Fig. 2 to estimate how many evaluations would have been required if we were to be a little more systematic about the process.

Figure 2

runs	*pop. size*	*gens.*	*evals.*	*evals.*
N	***M***	***G***	***Best CE***	***95% CI @ Best CE***
100 005	4	1 000	1 260 000	110 000
100 000	7	600	1 060 000	62 000
50 000	10	400	1 020 000	110 000
100 000	15	80	940 000	50 000
150 000	31	80	810 000	56 000
90 000	63	50	705 000	87 000
40 000	125	50	540 000	35 000
30 000	250	50	490 000	28 000
20 000	500	50	460 000	22 000
10 000	1 000	50	430 000	21 000
10 000	2 000	50	450 000	17 000
10 030	4 000	20	470 000	14 000
11 778	10 000	25	480 000	11 000
5 362	25 000	25	480 000	21 000

Number of independent runs N, population size M, number of generations evaluated G, best computational effort across all generations *Best* CE, and the width of 95% confidence interval bounds at the best generation *95% CI @ Best CE* for many runs of artificial ant on the Santa Fe trail. Following the custom in genetic programming, runs were independent in population size M but not in generation number G.

To get the data of Fig. 2, we performed a roughly uniform sampling in $\log_2 M$, while choosing the maximum number of generations G according to a somewhat arbitrary schedule that seemed to be larger than the optimal number of generations. What would a systematic algorithm for computing the best parameter settings look like?

As an early estimate of the magnitude of this effect, we performed $10\,000$ runs for each population size charted in Fig. 2, which determines the best fitness to around $20\,000$ fitness evaluations. The number of fitness evaluations performed to generate each series of Fig. 2 are listed in Fig. 3, along with the standard errors at the minimum of each curve. We also compute the number of fitness evaluations required to get a standard 95% confidence interval of $20\,000$ fitness evaluations. While the number of generations and ultimate confidence interval were decided somewhat arbitrarily, $3\,777\,000\,000$

fitness evaluations would be required to get uniform confidence intervals in this case. This large number of fitness evaluations will approximately scale as the inverse square of the desired confidence interval; that is, for a desired confidence interval of 40 000 fitness evaluations, it would be a quarter of that size. As the best observed computational effort has approximately 430 000 fitness evaluations, to get the optimal fitness to within 10% relative error, 95% of the time, we would need to perform 820 000 000 fitness evaluations.

Figure 3

N	*M*	*G*	*CE CI at min CE*	*N(20 000)*	*Evals(20 000)*
40 000	125	50	35 112	123 283	770 517 077
30 000	250	50	27 917	58 453	730 657 914
20 000	500	50	22 426	25 146	628 646 150
10 000	1 000	50	21 214	11 251	562 556 785
10 000	2 000	50	16 787	7 045	704 497 099
10 030	4 000	20	13 778	4 760	380 813 409

Number of independent runs *N*, population size *M*, number of generations evaluated *G*, 95% confidence interval bounds at the best generation *CE CI* at min *CE*, number of runs required to get a 95% confidence interval bounds of 20 000 evaluations $N(20\,000)$, and number of fitness evaluations required to get 95% confidence interval bounds of 20 000 evaluations $Evals(20\,000)$.

There is clearly a significant hidden cost in discovering the computational effort at the optimal settings. We have no way of knowing *a priori* how much work will be required in "typical" genetic programming evolutions, rather than the well-defined performance under "optimal" settings. Indeed, if the ratio between these two values is not constant, our expensive efforts to determine which technique is best for a given problem will be in vain. The best value of computational effort that we achieved is 430 000 fitness evaluations, which is about 1 900 times less than the amount of work to secure a relative error of less than 10%. When attacking difficult real-world problems, we may not have the luxury of being psychic and choosing optimal values for the parameters. How may we achieve reasonably good performance on problems that are too challenging for which to optimize parameter settings?

Prerequisites for Good Evolutionary Computation Performance

We can make progress on the question of optimizing parameter settings by considering what sort of regularities would be useful to solve this problem. First, it would be useful to have a well-defined algorithm for determining performance and parameter settings. A model of the computational effort for different problems would also be useful. What we would ideally like, for the purposes of evaluating performance using computational effort, is a model of the probability of success as a function of population size and generation number.

Let $p(M, G)$ be the unknown probability of seeing at least one success in the population by generation G using the population size M and the number of generations G as model variables. Of course, there are many other parameters of interest, such as tournament size, presence or absence of elitism, and so on, but we hold all these variables constant for now. We can then compute the continuous version of the computational effort for any parameter settings using the (2), which we specialize into the nearly identical form (3), below. We have written G so that it begins at 0, as is typical for genetic programming. As before, z is the target success probability, which we normally take to be 99%. In some of our work, we use an easier criterion, 50% success probability.

$$CE(M,G;z) = M(G+1)\frac{\ln(1-z)}{\ln(1-p(M,G))}$$

$$CE(z)_{\min} = \min_{M,G} CE(M,G;z) \tag{3}$$

From (3), we can derive that using the 99% criterion involves $\frac{\ln 0.01}{\ln 0.5} = 6.64$ times more work than using the 50% criterion. For uniformity, if we are running a system without elitism, we employ the standard trick of counting a run as successful if it has ever seen a successful individual. With this proviso, what can we say about $p(M,G)$?

Since $p(M,G)$ is a probability, we know that it remains between 0 and 1, without achieving either limit. We also know that the absolute probability of success *should* increase as we add generations or as we increase the population size – that is, $p(M,G)$ should be monotonic increasing in *M* and *G*. Another property that we typically observe is that the computational effort statistic has but a single local minimum in the space defined by the parameters *M* and *G*. We can see this in Fig. 1, where the single minimum is evident. We name this last property the unique computational effort minimum conjecture, which is shown algebraically in (4).

$$\forall g \forall m \frac{\partial CE(m,g;z)}{\partial m} \geq 0, \frac{\partial CE(m,g;z)}{\partial g} \geq 0 \tag{4}$$

Whenever individuals in a population are independent, the computational effort will be constant across the population size *M*. In this case, the probability of success is governed by the usual formula for the success of *M* independent trials, given in (5).

$$p(M,g) = 1-(1-p(1,g))^M \tag{5}$$

This will usually be true for the first generation, $G = 0$, where individuals are randomly generated independently from a common distribution. This chapter considers a simple model of performance evaluation using (5) that can show some progress towards a model of genetic programming performance.

Modelling Evolutionary Computation Performance

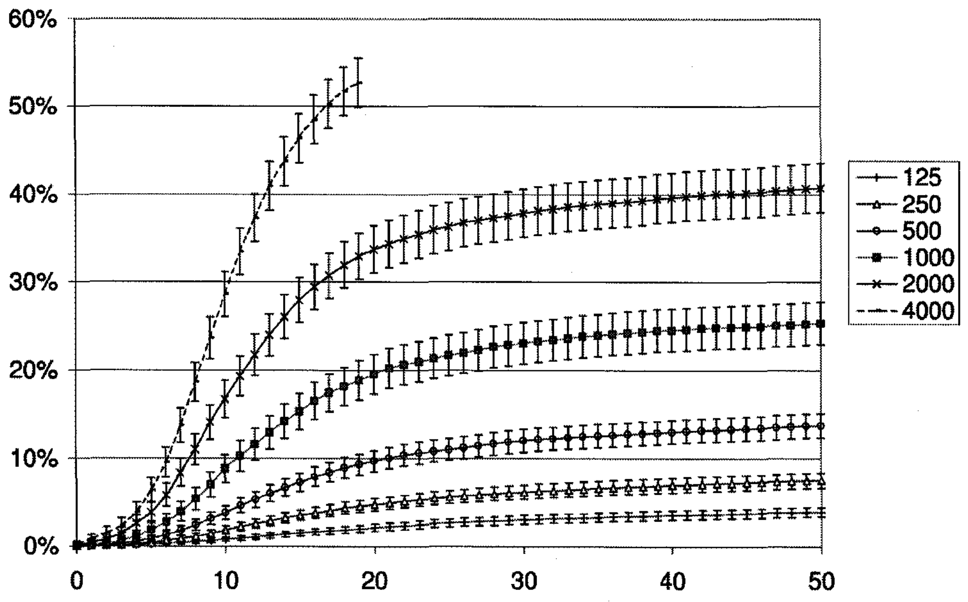
How can we model the performance of evolutionary computation systems generally? Genetic programming theory has much to say about the performance of individuals as a function of the previous generation using schema theory [Poli 2000a, Poli 2000b, Poli 2001a], and some work has been done on Markov-chain modelling of GP performance [Poli 2001b]. However, a functional form for success probability as a function of population size and generations performed remains elusive for non-trivial problems. Let us reprise (5). Define p_0 as the probability of success per randomly generated individual in the initial generation. The relation for the probability of seeing any success in the initial generation is given in (6).

$$p(M,0) = 1-(1-p_0)^M \tag{6}$$

Ultimately, we will be fitting observed data from conducted experiments to whatever model we develop. It would therefore be useful to have a model that has nice mathematical properties, and is as simple as possible. The simplest models are linear. We know immediately that no linear model can be appropriate for success probability, since any linear equation would give values that exceed 1 at large enough values of M and G. Let us consider some success curves, taken from the same data set that generated

Fig. 1, the artificial ant on the Santa Fe trail problem. Fig. 4 gives the success probability as a function of generation number, by population size.

Figure 4



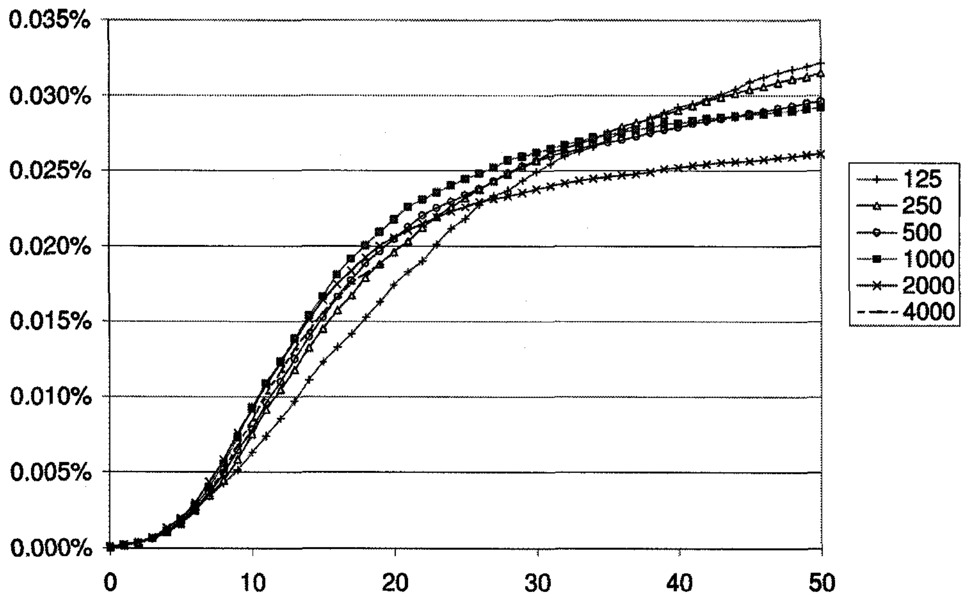
Estimated probability of success at each generation for the artificial ant on the Santa Fe trail problem as a function of population size. 95% confidence intervals across all 6 series are indicated; they have been corrected within each series by Monte Carlo modelling of the relevant errors, but no correction for multiple comparisons on the error has been performed.

While these curves are fairly accurate, it is difficult to perform interpolation between different values of M. For instance, we have no easy way of estimating from these data what the success probability for $M = 750$ at generation 20 will be, save that it should be between the values for $M = 500$ and $M = 1000$. An ideal model would be as accurate as possible when interpolating within ranges where we have observed data, and be reasonable when extrapolating beyond known data. There is a trick that we can do to make the data more uniform. Suppose that we had a relation like (6), but for an arbitrary generation number G – rather than for just the initial generation. If we had such a relation, then the addition of more data points would have a predictable nature, and we might easily interpolate between different values of M. An initial attempt would be, for modelling purposes only, to use the inverse of (6) to determine a hypothetical p_0 for each generation G – not just the initial generation. The relevant equation is given in (7), where $p_0(M,G)$ is now a quantity derived from observed data.

$$p_0(M,G) = 1 - (1 - p(M,G))^{1/M} \qquad (7)$$

Of course, since we will be using an EC system to determine the probabilities of success, (7) should in no way be interpreted to mean that the success probabilities for each individual are independent of one another after the first generation – that is normally only true in the initial generation. However, for modelling success probability as a function of population size, this trick may prove useful. We will know it is effective if a graph of $p_0(M,G)$ as a function of G shows that the curves for different values of M have some regular relationship to one another. Fig. 5 shows the same data as Fig. 4, but using (7) to compute $p_0(M,G)$.

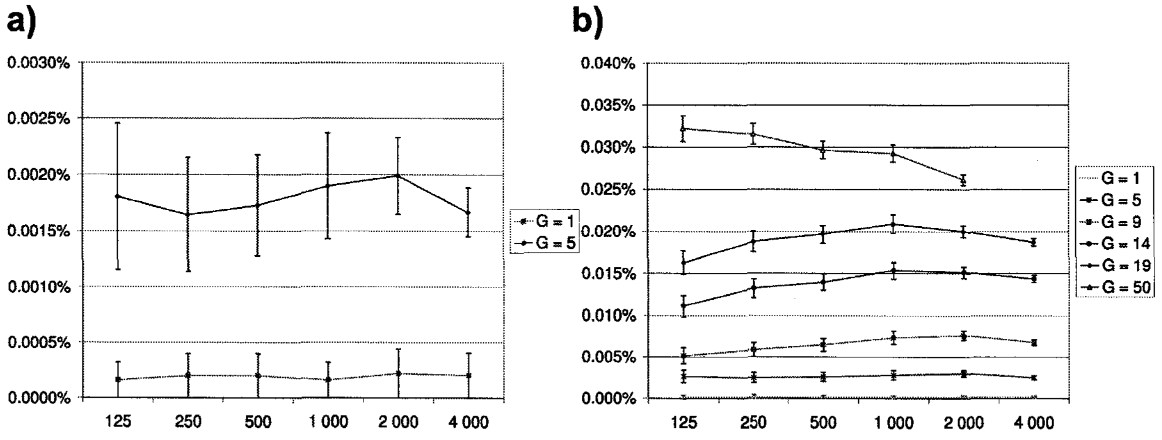
Figure 5



Estimated $p_0(M,G)$ at each generation G for the artificial ant on the Santa Fe trail problem as a function of population size. Error bars are not shown, as they overlap to a great degree, save for the curves labelled 125 and the tail of the curve labelled 2000. The curve labelled 4000 is difficult to see, as it largely overlaps the curve labelled 2000, and ends at generation 20.

As a casual inspection of Fig. 5 shows, these curves are much more similar than the curves of Fig. 4, which will lead to more accurate interpolation within the observed ranges. One obvious interpolation scheme for these data would be to fit a polynomial at each generation number to the population size M. Given the exponential nature of (7), however, we might get a better fit using the logarithm of M instead. Figs. 6a and 6b show $p_0(M,G)$ as a function of log M at various generation numbers.

Figure 6



Estimated $p_0(M,G)$ at several generation numbers G for the artificial ant on the Santa Fe trail problem as a function of logarithm of population size M. Fig. 6a shows a detail view of generations 1 and 5; Fig. 6b shows all the generations for which data were generated. Note that runs are not independent in G, so the smoothness in G is to some degree forced. The data are not particularly well constrained; 95% confidence intervals are shown on the graph.

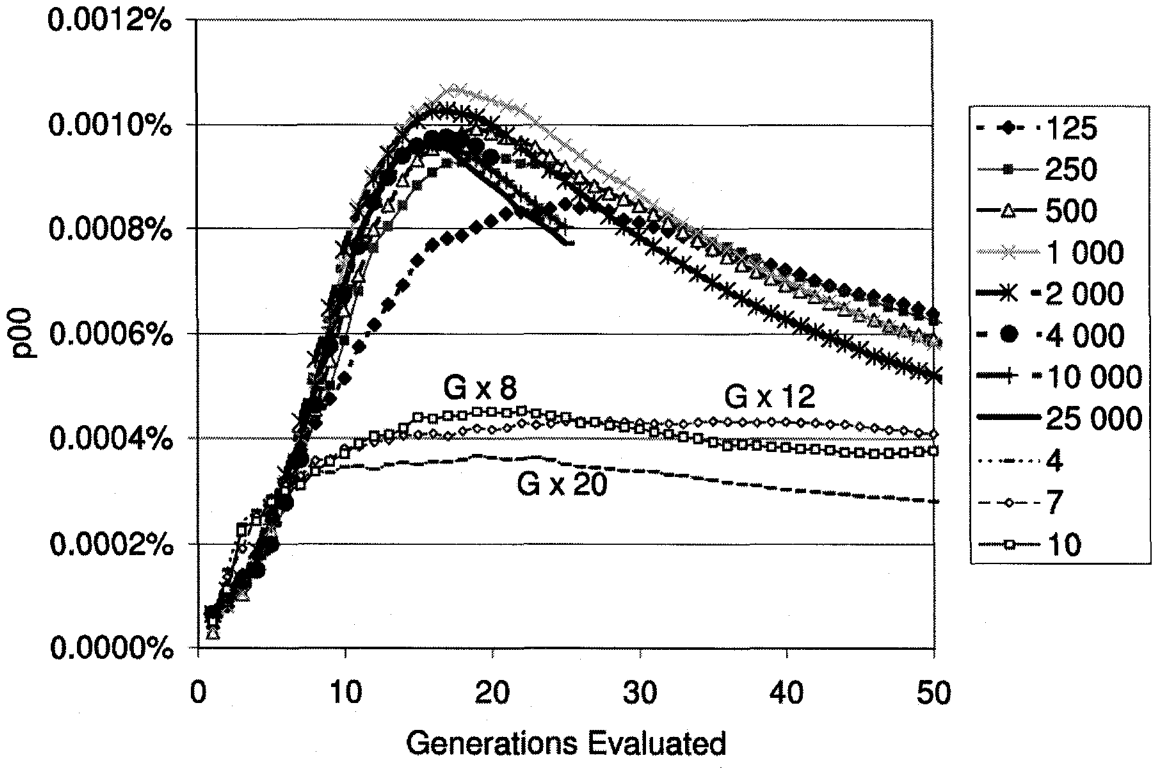
Fig. 6b indicates that the transform of (7) is quite productive, transforming a complex curvilinear trend into a nearly constant one! With respect to a simple polynomial regression model, the data of Figs. 6a and 6b indicate that there is no evidence for any trend for $G = 0$ and $G = 4$, and an increasingly curvilinear trend for the larger generation numbers. Of course, $G = 0$ must be constant if the initial generation is independent and identically distributed. Equivalently, a constant (zero slope) line in Figs. 6a and 6b corresponds to a line of constant computational effort as well.

Since the number of evaluations performed increases linearly with the generation number as well, we might try compensating for that, too. We might try to perform the transform of (7) for the generation number G as well. Define $p_{00}(M,G)$ as in (8).

$$p_{00}(M,G) = 1-(1-p(M,G))^{1/M(G+1)} \qquad (8)$$

It may not be apparent at first, but (8) is another way of specifying the computational effort of a task – there is a one-to-one and onto relationship between the computational effort and $p_{00}(M,G)$. In fact, $CE(M,G;z) \propto \frac{1}{p_{00}(M,G)}$, though it is not trivial to prove. A plot of $p_{00}(M,G)$ as a function of M and G is given in Fig. 7.

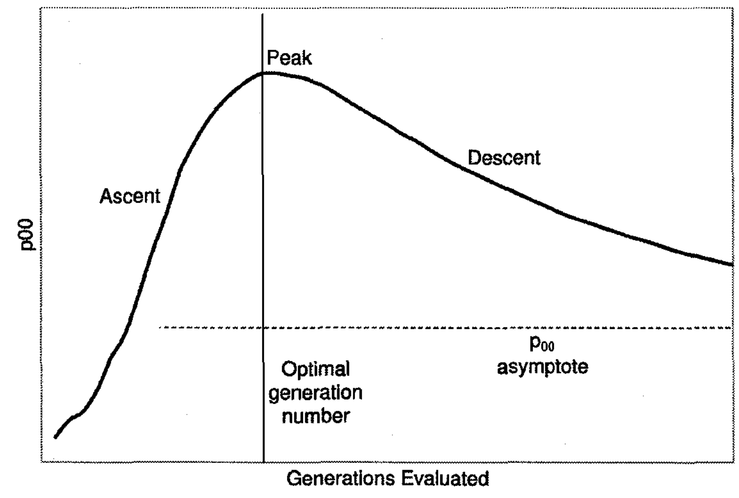
Figure 7



Estimated $p_{00}(M,G)$ at each generation G for the artificial ant on the Santa Fe trail problem as a function of number of generations evaluated. The legend shows the population size M for each series. The series $M = 4$, $M = 7$, and $M = 10$ are shown with their generation number divided by the indicated values. Error bars are not shown for clarity. Larger values of $p_{00}(M,G)$ indicate smaller computational effort for a set of parameter settings.

We will briefly comment on some trends observable in Fig. 7. These comments are specific to the artificial ant on the Santa Fe trail problem, but we suspect that they may hold true in general for any problem, as they derive from the "deep math" governing the performance of evolutionary systems. In Fig. 8, we have labelled portions of this curve for easy reference.

Figure 8



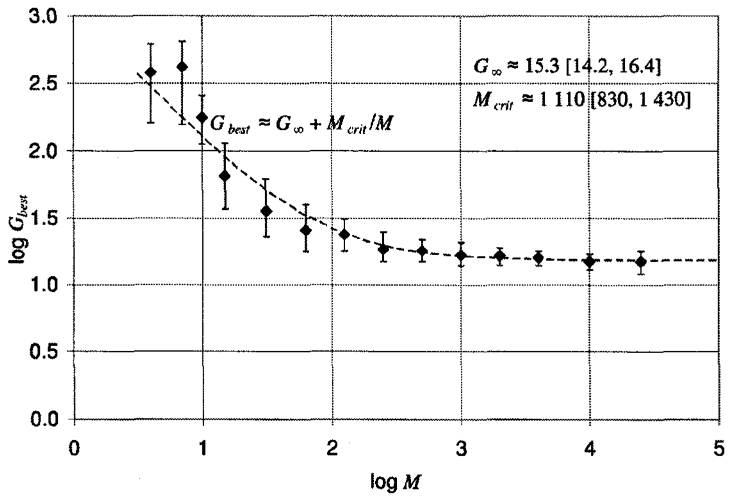
A sketch of the $p_{00}(M,G)$ curves of Fig. 7, labelling some salient features.

Firstly, it is clear from examining Fig. 7 that the ascent portions of the $p_{00}(M,G)$ curves appear to converge with increasing *M*. This is perhaps obscured in Fig. 7 due to the compression of the X-axis for the small-population curves. G_{best} seems to approach a limit $G_{\min}$ as the population size *M* goes to infinity. For these data, a simple analytical model of the best generation number as a function of *M* is given by (9). The model of (9) was chosen to interpolate smoothly between two trends: constant best generation number at large *G*, and a best generation number that maintains a constant number of evaluations at small *G*. Fig. 9 shows the observed estimates of G_{best}, best-fit values and confidence intervals for the model parameters G_∞ and M_{crit}.

$$G_{best} = G_\infty + \frac{M_{crit}}{M} \tag{9}$$

The best fit was obtained using $G_\infty = 15.3$ and $M_{crit} = 1110$ by χ^2 minimization as described in Chapter 15 of [Press 1992c].

Figure 9



A graph of the approximate optimal value for G as a function of log M. The data are interpolated from fitting a quadratic to the observed minima of the computational effort graphs for each value of M. A two parameter best-fit to the optimal generation number is indicated, as determined by χ^2 minimization of the fitting curve. The presented fitting curve has $\chi^2 = 15.65$ with 13 d.f., p = 0.27. 95% confidence intervals on the parameter values were determined by examining the appropriate quantiles of the parameters after doing 10 000 resamples of the χ^2 minimization procedure.

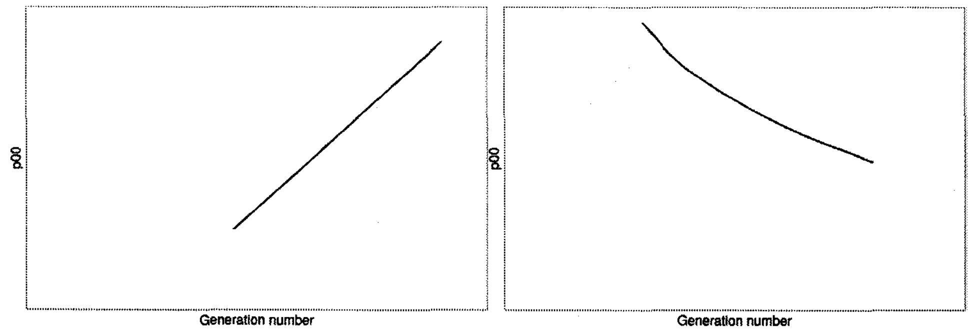
While the locations of the peaks G_{best} are somewhat uncertain in Fig. 7, they clearly become sharper as M increases. This can be seen to some degree in Fig. 9, by noting the relative widths of the confidence intervals of G_{best}. If the reader refers to Fig. 2, this effect is even more apparent: *more* runs were performed for the smaller population sizes. The height of the interval between the peak and the asymptote in $p_{00}(M,G)$ increases as M increases. This analysis has some implications for efficient evolution: when using a small population size, one doesn't need to be very precise in G; this is more important when using a large population size.

Other Possible p_{00} Curves

If we look at a range of evolutionary computation problems, we can expect to encounter a number of differently-shaped $p_{00}(M,G)$ curves. We believe that the majority will have the general character of Fig. 7, although we have no formal proof of this claim. The two curves shown in Fig. 10 are often encountered in practice. While at first, the shape of the curves in Fig. 10 seem different from that of Fig. 8, additional experimentation shows that there is an underlying similarity. Fig. 10a represents the ascent phase of Fig. 8, and occurs when we have not used a sufficiently large number of generations to achieve optimality. Fig. 10b is similar to the descent phase of Fig. 8, and occurs when a problem is best solved by performing runs at smaller generation numbers. The curve of Fig. 10b can begin at $G = 0$; in this case, random tree generation is more efficient than evolution for the given population size. If this shape obtains for all population sizes, the problem is GP-hard.

Figure 10

a) **b)**



A sketch of some alternative $p_{00}(M,G)$ curves that may be encountered. These curves are based on data from a sorting problem described in Chapter 7. Fig. 10a may be viewed as a zoomed portion of the ascent phase of Fig. 7. Fig. 10b may be viewed as the descent phase of Fig. 7. In both of these cases, the overall curve does have the shape of Fig. 7 when data is generated to cover a larger range of generations.

A Progressive Algorithm for Computational Evolution

Suppose that we wanted good performance from evolutionary computation in an environment where the optimal values of M and G are unknown. We can use some of the knowledge from Fig. 7 to discuss whether an adaptive algorithm could be fashioned that provides good performance. First, let us see what an optimal algorithm would do. Computational effort is minimized in a model with repeated trials by repeatedly performing runs with $M = M_{best}$ and $G = G_{best}$ until a solution is found; that is, the values of the two key parameters should not change on successive runs. This is clear from the independence of successive runs: if a better value of M_{best} and G_{best} could be found on iteration t, it would be better at every iteration. We would simply use these parameters instead of our supposedly "optimal" parameters. For a real-world problem, however, we want to find a success with the smallest computational effort in a situation where we know virtually nothing about the optimal parameter settings. In general, we immediately run afoul of the No Free Lunch theorem. For problems we are likely to encounter, the regularities in fitness space are vastly stronger than the randomness for which the No Free Lunch applies. Supposing that the qualitative model of Figs. 7 and 8 is a good one for the problem at hand, how might we rationally allocate runs in the parameter space?

One way that is guaranteed to work would be to allocate a schedule of runs with exponentially increasing M. Since the probability of success must increase with increasing M, this algorithm will eventually find a solution for any solvable problem. Specifically, we can define a schedule of run counts $\{M_1, M_2, \ldots, M_{success}\}$ as in (10), where the series continues until success is achieved. Here μ is a base parameter determining the exponential step size; we might choose $\mu = 2$ in practice.

$$M_i = \mu^i \tag{10}$$

The downside of this exponential schedule is that we have no rational way of allocating the generation number *G*. From Fig. 7, it is clear that an appropriate choice of *G* has a significant impact on the computational efficiency of a solution. For instance, at *M* = 1 000, the computational effort varies by a factor of 21 between *G* = 0 and *G* = 17. For other problems we might expect an even greater difference, since Santa Fe trail is effectively GP-hard to begin with.

One solution to the problem of allocating values for *G* would be to use the same kind of exponential scan across *G* in successive runs, just as we did in (10). This would make the schedule for *G* that of (11), where *j* ranges from 0 to *i*. When $j = i$, we would advance to the next step in *i*, as given by (10). The complete algorithm is given in SYSTEMATIC-EC-SOLVER.

$$G_j = \mu^j \tag{11}$$

Algorithm 1: SYSTEMATIC-EC-SOLVER

Input: tournament size *T*; a fitness operator $f : \mathbb{T} \to \mathbb{R}$ for which smaller values represent better solutions; $\min f$, which is the best possible value of f; and a base μ that controls the schedule of values of *M* and *G* used

Output: a tree $t \in \mathbb{T}$ which is a perfect solution to *f*

```
$i \leftarrow 0$
for ever
    $M \leftarrow \mu^i$
    for $j \leftarrow 0$ to $i$ do
        $G \leftarrow \mu^j$
        $hero \leftarrow$ EVOLVE-TREES $M, T, G, f(t)$
        if $f(hero) = f_{\min}$ then return $hero$
    end for
    $i \leftarrow i+1$
end for
```

SYSTEMATIC-EC-SOLVER will eventually return a solution to any posed problem, if a solution can be generated from the random initialization procedure. This is guaranteed since the population size M increases without bound, and $G = 0$ is visited once per iteration of the outside loop. A boundless number of random individuals will be generated. The odds of any event of finite probability occurring is 1 if an unlimited number of independent trials is performed. SYSTEMATIC-EC-SOLVER may, of course, perform rather poorly in practice, as it has no way of lingering on promising values of $\{M, G\}$.

We view SYSTEMATIC-EC-SOLVER as one instance of a family of related algorithms that are defined by performing serial evolutions on a problem of unknown difficulty. We term the family of such algorithms *progressive* EC algorithms. We can imagine several ways in which SYSTEMATIC-EC-SOLVER might be improved. One obvious step would be to use the fitness of the best individual created in a run to predict how well the EC system is performing. As explained in Figs. 15a and 15b of Chapter 3 and related discussion, we run the risk of incorrectly deciding that certain parameter settings are optimal by using mean best fitness as a proxy for success probability. However, such an approach is likely to be far better than doing nothing! Another possibility is to use the qualitative model of Fig. 7 to optimize the schedule of M and G. We can demonstrate the benefit of having a model by defining a hypothetical model to see what benefit might come from it.

Suppose that we knew that $M_{crit} \approx \sqrt{CE(0.99)_{\min}}$, $M_{optimal} \approx M_{crit}$, and $G_{\infty} \approx 4 \log_T M_{crit}$. These relations happen to be approximately true for the artificial ant on the Santa Fe trail problem. The relation for M_{crit} was chosen from happenstance, and the relation for G_∞ was chosen based on the expected takeover time for a successful individual as a function of population size. Under this model, using the relation of Fig. 9, we can compute that $G_{opt} \approx 4\log_T M_{crit} + 1$. We should do $r_{opt} \approx \frac{\sqrt{CE(0.99)_{\min}}}{2\log_T CE(0.99)_{\min} + 1}$ independent runs to find our solution with a 99% success probability. We could then choose an exponential schedule for the unknown best computational effort CE, as in (12).

$$CE_i = \kappa^i \tag{12}$$

The complete algorithm is given in MODEL-BASED-EC-SOLVER.

Algorithm 2: MODEL-BASED-EC-SOLVER

Input: tournament size *T*; a fitness operator $f : \mathbb{T} \to \mathbb{R}$ for which smaller values represent better solutions; $\min f$, which is the best possible value of f; and a base κ that controls the schedule of values of *M* and *G* used

Output: a tree $t \in \mathbb{T}$ which is a perfect solution to *f*

```
$i \leftarrow 0$
for ever
    $CE_{est} \leftarrow \kappa^i$
    $M \leftarrow \sqrt{CE_{est}}$
    $G \leftarrow 4\log_T M + 1$
    $runs \leftarrow \frac{CE_{est}}{M(G+1)}$
    for $j \leftarrow 1$ to $runs$ do
        $hero \leftarrow$ EVOLVE-TREES $M, T, G, f(t)$
        if $f(hero) = f_{\min}$ then return $hero$
    end for
    $i \leftarrow i + 1$
end for
```

This schedule will perform better than SYSTEMATIC-EC-SOLVER to the extent that the model given above is appropriate for the problem at hand. Suppose that this model happens to be correct, and $\exists i \mid CE(0.99)_{\min} = \kappa^i$. In this case, we will solve our problem with 99% probability by performing at most $\left(1+\frac{1}{\kappa-1}\right)CE(0.99)_{\min}$ fitness evaluations.

The argument is as follows. At $i=\log_{\kappa} CE(0.99)_{\min}$, we have the exact optimal parameters, and we will perform $CE(0.99)_{\min}$ fitness evaluations and obtain a 99% chance of success. We then simply need to count how many fitness evaluations we performed before we got to this value of *i*. This is a straightforward summation of a power series, which can be at most $\frac{1}{\kappa-1}CE(0.99)_{\min}$ if *i* is large.

It is more likely that there is no value of *i* for which the schedule of (12) exactly hits the 99% computational effort of the problem. Suppose that there is a value of *i* for which κ^i is slightly less than $CE(0.99)_{\min}$. We would then perform at least one extra step in *i*. If the optimal parameters computed for this extra step were close enough to optimal so that our performance did not degrade greatly, we would be very likely to exceed a formal 99% probability of success for the entire schedule. Accounting for this extra step, we achieve a 99% probability of success after performing $\left(\kappa+1+\frac{1}{\kappa-1}\right)CE(0.99)_{\min}$ fitness evaluations in total. This functional form can be minimized to solve for κ using (13).

$$\frac{d}{d\kappa}\left(\kappa+1+\frac{1}{\kappa-1}\right)=0 \qquad (13)$$

(13) has one feasible solution: $\kappa=2$. With this choice of κ, we have the final result of (14): we can solve an unknown problem with 99% probability by performing at most $4\,CE(0.99)_{\min}$ fitness evaluations.

$$CE_{systematic} \le 4\,CE(0.99)_{\min} \qquad (14)$$

Having a reasonable model of GP performance has enabled us to derive a useful upper bound on the amount of work that we must perform even when we do not know the optimal settings *a priori*. The bound thus derived is, at worst, a constant factor higher than the optimal value. We can therefore use comparisons between computational effort measured at the optimal parameter settings to infer relationships among the computational efforts on typical systems. This last statement is not only true for the model presented above; it will hold whenever (15) is true, where *k* is an arbitrary constant. This may enable efficient solution of a broader set of models than the hypothetical model presented above.

$$CE_{systematic} \leq k\ CE(0.99)_{\min} \qquad (15)$$

Determining whether this sort of model is appropriate for GP problems would involve considering the performance curves of many GP problems, and is beyond the scope of this work. We are very interested in what the GP theory community can add to this sort of modelling effort.

Automatic Parameter Setting for Population Size and Generation Number

One final note before we close this chapter. In our work with the scientist algorithm, we suffer from the common problem of not knowing what values of *M* and *G* are optimal. Here is a straightforward algorithm for determining these key parameter settings for a Genetic Programming system, CHOOSE-POPULATION-SIZE-AND-GENERATION-NUMBER. As determining good settings for the population size and generation number with any confidence requires many successful runs, this algorithm is only appropriate for benchmark problems for testing GP systems against one another. CHOOSE-POPULATION-SIZE-AND-GENERATION-NUMBER assumes that there is a single local minimum in the computational effort as a function of population size *M* and number of generations *G* and that there are no points of inflection in the computational effort graph. However, it does not assume that computational effort is linear in either parameter, which it generally is not.

There are a few tricks embedded in CHOOSE-POPULATION-SIZE-AND-GENERATION-NUMBER that we should explain briefly. The algorithm overall has three components. Throughout the algorithm, we track the number of runs performed and number of successes achieved for each combination of *M* and *G* evaluated.

First, we find a successful set of parameter settings *M* and *G* that can solve the problem subject to our success criterion using FIND-INITIAL-M-G. We iterate between four candidate models so as not to be too inefficient. These models represent good choices if the underlying problem is best solved by random search, by hill-climbing, by a Santa Fe ant-like model, or by a model that balances *M* and *G*. Otherwise FIND-INITIAL-M-G works like MODEL-BASED-EC-SOLVER, presented above. These parameter settings are then used to compute an initial estimate of the computational effort required to solve the problem, CE_{est}.

Second, we construct an exponentially distributed grid over *M* and *G* based on the estimated computational effort and user-supplied parameter, β. The values of this grid are chosen so that *M* is no smaller than 4 and *G* is no smaller than 1. Specifically, we can choose *M* using the schedule of (16), and *G* using (17).

$$M_i = \left\lfloor \beta^{-i} CE_{est} + 0.5 \right\rfloor \forall i = 0 \ldots \left\lfloor \log_\beta \frac{CE_{est}}{4} \right\rfloor \tag{16}$$

$$G_i = \left\lfloor \frac{CE_{est}}{M_i} + 0.5 \right\rfloor \forall i = 0 \ldots \left\lfloor \log_\beta \frac{CE_{est}}{4} \right\rfloor \tag{17}$$

Third, CHOOSE-POPULATION-SIZE-AND-GENERATION-NUMBER iteratively improves the error bounds at each point in the grid until a desired relative precision is achieved. It estimates upper and lower bounds on the computational effort at each point in the grid from the data observed so far, correcting for multiple comparison errors using the Bonferroni inequality. The algorithm them conducts experiments using this grid to improve the error bounds on the computational effort at each point. In each round, a fixed number of fitness evaluations roughly equal to CE_{est} is performed at each point along the grid. Because the $G = 1$ computational efforts are degenerate in M, we only perform experiments for the highest value of M. The grid's limiting generations are then refined if possible, by dropping all treatments that are no longer competitive with the best candidate. We determine the best candidate by computing the smallest upper bound for computational effort among the entire population of competing parameter settings, CE_{lub}. This is then compared against the estimated lower bounds on computational effort for each treatment to determine which treatments can be safely dropped. This focusses computer time on the best parameter settings as the algorithm progresses. By comparing confidence interval bounds rather than doing formal hypothesis testing, we gain an additional measure of conservatism in the algorithm. This is important because of the statistical issues with computational effort discussed in Chapter 4. More runs are performed until the target confidence interval width is achieved for the best parameter setting. Finally, we return the estimated confidence interval and the set of best candidate values for *M* and *G*.

Algorithm 3a: CHOOSE-POPULATION-SIZE-AND-GENERATION-NUMBER

Input: tournament size *T*; a fitness operator $f:\mathbb{T}\rightarrow\mathbb{R}$ for which smaller values represent better solutions; a success criteria $success:\mathbb{R}\rightarrow\mathbb{B}$ that returns 1 iff the given fitness value counts as a success; a base β that controls the schedule of values of *M* and *G* attempted; a desired success probability *z*, and a target relative error ε that represents the 1σ target relative error desired for the ultimate computational effort estimate

Output: an estimated computational effort CE_z and a set of statistically likely best candidates for *M* and *G*

```
$[CE_{est}, M] \leftarrow$ FIND-INITIAL-M-G $T,\ f(t), success,\ \beta$
$[\mathcal{M}, \mathcal{G}] \leftarrow$ MAKE-M-G-GRID $\beta, M,\ CE_{est}$
do
    for $i \leftarrow 1$ to $|\mathcal{M}|$
        $M \leftarrow \mathcal{M}_i$
        $G \leftarrow \mathcal{G}_i$
        $runs \leftarrow \begin{cases} \left\lceil \frac{CE_{est}}{GM} \right\rceil & \text{if } G>0 \\ 0 & \text{otherwise} \end{cases}$
        if $G \geq 2$ or $G = 1 \wedge runs \geq 2$ then
            $hero \leftarrow$ EVOLVE-TREES $M, T, G,\ f(t)$
            UPDATE-DATABASE-WITH-SUCCESSES $M, G, success$
        end if
    end for
    $\mathcal{G} \leftarrow$ UPDATE-GENERATION-LIMITS $\mathcal{M}$
until ESTIMATE-CE-RELATIVE-ERROR($\mathcal{M}$)$< \varepsilon$
return ESTIMATE-CE-AND-BEST-CANDIDATES($\mathcal{M}$)
```

Algorithm 3b: Find-Initial-M-G

Input: tournament size *T*; a fitness operator $f : \mathbb{T} \to \mathbb{R}$ for which smaller values represent better solutions; a success criteria $success : \mathbb{R} \to \mathbb{B}$ that returns 1 iff the given fitness value counts as a success; a base β that controls the schedule of values of *M* and *G* attempted

Output: initial parameter settings for *M* and a point estimate of the computational effort CE_{est} that can achieve success

```
$CE_{est} \leftarrow 1$
for ever
    $M \leftarrow CE_{est}$                – Model 1: GP-hard problems – 1 run, $G = 1$, $M = CE_{est}$
    $G \leftarrow 1$
    if SUCCESSFUL-EVOLUTION-POSSIBLE($M$, $T$, $G$, 1, $f(t)$, $success$) then exit for
    $M \leftarrow \left\lceil \sqrt{CE_{est}} \right\rceil$          – Model 2: Santa-Fe-like problems
    $G \leftarrow 4\log_T M + 1$
    $r \leftarrow \left\lceil \frac{CE_{est}}{MG} \right\rceil$
    if SUCCESSFUL-EVOLUTION-POSSIBLE($M$, $T$, $G$, $r$, $f(t)$, $success$) then exit for
    $M \leftarrow \left\lceil \sqrt{CE_{est}} \right\rceil$          – Model 3: balanced $M$, $G$ approach
    $G \leftarrow M$
    if SUCCESSFUL-EVOLUTION-POSSIBLE($M$, $T$, $G$, 1, $f(t)$, $success$) then exit for
    $M \leftarrow 7$                      – Model 4: small population size – GP hill-climbing
    $G \leftarrow \left\lceil \frac{CE_{est}}{M} \right\rceil$
    if SUCCESSFUL-EVOLUTION-POSSIBLE($M$, $T$, $G$, 1, $f(t)$, $success$) then exit for
    $CE_{est} \leftarrow \beta CE_{est}$
end for
return $[CE_{est}, M]$
```

Algorithm 3c: SUCCESSFUL-EVOLUTION-POSSIBLE

Input: population size M; number of generations G; tournament size T; number of runs r; a fitness operator $f : \mathbb{T} \to \mathbb{R}$ for which smaller values represent better solutions; a success criteria $success : \mathbb{R} \to \mathbb{B}$ that returns 1 iff the given fitness value counts as a success
Output: *true* if a successful evolution obtained, *false* otherwise

```
for $j \leftarrow 1$ to $r$ do
    $hero \leftarrow$ EVOLVE-TREES $M, T, G, f(t)$
    UPDATE-DATABASE-WITH-SUCCESSES $M, G, success$
    if $success(f(hero)) = 1$ then return $true$
end for
return $false$
```

Algorithm 3d: UPDATE-DATABASE-WITH-SUCCESSES

Input: population size M; number of generations performed G; success criteria $success : \mathbb{R} \to \mathbb{B}$ that returns 1 iff the given fitness value counts as a success, access to the best fitness values for each generation of the last run of EVOLVE-TREES
Globals: a sparse matrix $\mathbf{S}_{M,G}$ that tracks the number of successes and a sparse matrix $\mathbf{N}_{M,G}$ that tracks the number of runs performed
Output: none

```
Let $bestFitness(g)$ be the best fitness achieved at generation $g$ in the last run
for $g \leftarrow 0$ to $G - 1$ do
    $\mathbf{N}_{M,g} \leftarrow \mathbf{N}_{M,g} + 1$
    $\mathbf{S}_{M,g} \leftarrow \mathbf{S}_{M,g} + success(bestFitness(g))$
end for
```

Algorithm 3e: UPDATE-GENERATION-LIMITS

Input: a set of population sizes $\mathcal{M}$
Globals: a sparse matrix $\mathbf{S}_{M,G}$ that tracks the number of successes and a sparse matrix $\mathbf{N}_{M,G}$ that tracks the number of runs performed
Output: a set of limiting generation numbers $\mathcal{G}$

```
$[\mathbf{CE}^{LB}, \mathbf{CE}^{central}, \mathbf{CE}^{UB}] \leftarrow$ ESTIMATE-CONFIDENCE-INTERVALS $\mathcal{M}$
$CE_{\text{lub}} \leftarrow \min_{M,G} \mathbf{CE}^{UB}_{M,G}$
for $i \leftarrow 1$ to $|\mathcal{M}|$
    $M \leftarrow \mathcal{M}_i$
```

```
        $G_{\max} \leftarrow G \mid \mathbf{N}_{M,g} = 0 \; \forall g > G$
        $g \leftarrow G_{\max}$
        do
            $g \leftarrow g - 1$
        while $g > 0$ and $\mathbf{CE}^{LB}_{M,g} > CE_{\text{lub}}$
        $\mathcal{G}_i \leftarrow g$
    end for
    return $\mathcal{G}$
```

Algorithm 3f: ESTIMATE-CONFIDENCE-INTERVALS

Input: a set of population sizes $\mathcal{M}$

Globals: a sparse matrix $\mathbf{S}_{M,G}$ that tracks the number of successes; a sparse matrix $\mathbf{N}_{M,G}$ that tracks the number of runs performed; and a target upper bound on Type-I statistical error α

Output: a list of estimates for the lower bound, central value and upper bound on the computational effort for each value of *M* and *G*

```
    $treatments \leftarrow \sum_{M \in \mathcal{M}} \max_g \mathbf{N}_{M,g} > 0$
    $\alpha_{eff} \leftarrow \dfrac{\alpha}{treatments}$
    $z_{crit} \leftarrow -Z \left| \int_{z=-\infty}^{z=Z} \dfrac{e^{-\frac{x^2}{2}}}{\sqrt{2\pi}} dz = \alpha_{eff} \right.$
    for $i \leftarrow 1$ to $|\mathcal{M}|$
        $M \leftarrow \mathcal{M}_i$
        $G_{\max} \leftarrow G \mid \mathbf{N}_{M,g} = 0 \; \forall g > G$
        for $g \leftarrow 0$ to $G_{\max}$
            $\left[CE^{LB}, CE^{central}, CE^{UB}\right] \leftarrow$ ESTIMATE-COMPUTATIONAL-EFFORT $M, g, \mathbf{N}_{M,g}, \mathbf{S}_{M,g}, \alpha_{eff}, z_{crit}$
            $\mathbf{CE}^{LB}_{M,g} \leftarrow CE^{LB}$
            $\mathbf{CE}^{central}_{M,g} \leftarrow CE^{central}$
            $\mathbf{CE}^{UB}_{M,g} \leftarrow CE^{UB}$
        end for
    end for
    return $\left[\mathbf{CE}^{LB}, \mathbf{CE}^{central}, \mathbf{CE}^{UB}\right]$
```

Algorithm 3g: ESTIMATE-CE-RELATIVE-ERROR

Input: a set of population sizes $\mathcal{M}$

Globals: a sparse matrix $\mathbf{N}_{M,G}$ that tracks the number of runs performed and the target upper bound on Type-I statistical error α as used by ESTIMATE-CONFIDENCE-INTERVALS

Output: an estimate of the ratio of standard deviation of the best computational effort to the best computational effort

```
$treatments \leftarrow \sum_{M \in \mathcal{M}} \max_g \mathbf{N}_{M,g} > 0$
$z_{crit} \leftarrow -Z \left| \int_{z=-\infty}^{z=Z} \frac{e^{-\frac{x^2}{2}}}{\sqrt{2\pi}} dz = \frac{\alpha}{treatments} \right.$
$[\mathbf{CE}^{LB}, \mathbf{CE}^{central}, \mathbf{CE}^{UB}] \leftarrow$ ESTIMATE-CONFIDENCE-INTERVALS $\mathcal{M}$
$M_{best} \leftarrow \arg\min_M \left( \min_G \mathbf{CE}^{central}_{M,G} \right)$
$G_{best} \leftarrow \arg\min_G \left( \min_M \mathbf{CE}^{central}_{M,G} \right)$
if $\mathbf{CE}^{LB}_{M_{best},G_{best}} = 1$ or $\mathbf{CE}^{UB}_{M_{best},G_{best}} = \infty$ return $\infty$
return $\frac{1}{z_{crit}} \sqrt{\frac{\mathbf{CE}^{UB}_{M_{best},G_{best}}}{\mathbf{CE}^{LB}_{M_{best},G_{best}}}}$
```

Algorithm 3h: ESTIMATE-CE-AND-BEST-CANDIDATES

Input: a set of candidate population sizes $\mathcal{M}^{in}$

Globals: a sparse matrix $\mathbf{N}_{M,G}$ that tracks the number of runs performed and the target upper bound on Type-I statistical error α as used by ESTIMATE-CONFIDENCE-INTERVALS

Output: the list $[CE_{best}, \mathcal{M}, \mathcal{G}]$, where CE_{best} is an estimate of the best computational effort, and $\mathcal{M}$ and $\mathcal{G}$ are arrays of candidates for the best values of M and G

```
$[\mathbf{CE}^{LB}, \mathbf{CE}^{central}, \mathbf{CE}^{UB}] \leftarrow$ ESTIMATE-CONFIDENCE-INTERVALS $\mathcal{M}^{in}$
$index \leftarrow 1$
$CE_{\text{lub}} \leftarrow \min_{M,G} \mathbf{CE}^{UB}_{M,G}$
for $i \leftarrow 1$ to $|\mathcal{M}^{in}|$
    $M \leftarrow \mathcal{M}^{in}_i$
    $G_{\max} \leftarrow G \mid \mathbf{N}_{M,g} = 0 \; \forall g > G$
    for $g \leftarrow 0$ to $G_{\max}$
```

```
            if $\mathbf{CE}^{UB}_{M,g} < CE_{\text{lub}}$ then
                $\mathcal{M}_{index} \leftarrow M$
                $\mathcal{G}_{index} \leftarrow g$
                $index \leftarrow index + 1$
            end if
        end for
    end for
    return $\left[\min_{M,G} \mathbf{CE}^{central}_{M,G}, \mathcal{M}, \mathcal{G}\right]$
```

One component of this algorithm remains to be described. We need to estimate the computational effort for each value of *M*, *G*, number of successes *S*, and number of trials *N*. To do this effectively, we should really find exact α and $1-\alpha$ confidence intervals for the binomial probability. We begin with (18), the general formula for the probability of an unknown binomially distributed variable. The problem of confidence interval estimation is that of course, we observe values for *N* and *S* but do not have any knowledge of p. We adopt a model of uniform prior probability for the variable p, and get (19a) and (19b) for the upper and lower bounds.

$$B(N,S;p) = \binom{N}{S} p^S (1-p)^{N-S} \tag{18}$$

$$p^{LB} = \begin{cases} P \;\middle|\; \alpha = \dfrac{\int_{p=P}^{p=1}\left(\sum_{s=S}^{N} B(N,s;p)\right)dp}{\int_{p=0}^{p=1}\left(\sum_{s=S}^{N} B(N,s;p)\right)dp} & \text{if } S > 0 \\ 0 & \text{otherwise} \end{cases} \tag{19a}$$

$$p^{UB} = \begin{cases} P \;\middle|\; \alpha = \dfrac{\int_{p=0}^{p=P}\left(\sum_{s=0}^{S} B(N,s;p)\right)dp}{\int_{p=0}^{p=1}\left(\sum_{s=0}^{S} B(N,s;p)\right)dp} & \text{if } S < N \\ 1 & \text{otherwise} \end{cases} \tag{19b}$$

These integrals can be solved efficiently using the incomplete beta distribution [Press 1992d], but users may not have this special function implemented in their mathematics libraries. We will adopt a compromise position. For $S = 0$ and $S = N$, (19a) and (19b) have elegant solutions in the conventional transcendental functions for p, namely (20a) and (20b).

$$p^{LB} = \alpha^{1/N+1} \quad \text{if } S = N \tag{20a}$$

$$p^{UB} = 1 - \alpha^{1/N+1} \quad \text{if } S = 0 \tag{20b}$$

We use the conventional α and $1-\alpha$ confidence intervals for a binomially distributed variable for all other values of S. If these confidence intervals cover 0 or 1, we give up and revert to (20a) if $p < 0.5$ or (20b) if $p > 0.5$. After we have reliable confidence intervals for p, we can compute the computational effort using (1) as normal. The algorithm is given in ESTIMATE-COMPUTATION-EFFORT.

Algorithm 3i: ESTIMATE-COMPUTATIONAL-EFFORT

Input: population size *M*; generation number *G*; number of trials *N*, number of successes *S*, desired type I error probability α, and z_{crit}, the lower critical value of the standard normal distribution for α

Globals: the target probability of success for the computational effort, *z*

Output: a list of estimates for the lower bound, central value and upper bound on the computational effort

```
$\sigma_p \leftarrow \sqrt{\frac{p(1-p)}{N}}$
$B \leftarrow z\sigma_p$
if $p = 0$ or $p - B < 0$ or $p = 1$ or $p + B > 1$ then
    if $p < 0.5$ then
        $p^{LB} \leftarrow 0$
        $p^{UB} \leftarrow 1 - \alpha^{1/N+1}$
    else
        $p^{LB} \leftarrow \alpha^{1/N+1}$
        $p^{UB} \leftarrow 1$
    end if
else
    $p^{LB} \leftarrow p - B$
    $p^{UB} \leftarrow p + B$
end if
Let $CE(z,p)$ be $\begin{cases} \infty & p = 0 \\ 0 & p = 1 \\ M(G+1)\frac{\ln 1 - z}{\ln 1 - p} & \text{otherwise} \end{cases}$.
return $\left[CE(z,p^{LB}), CE(z,p), CE(z,p^{UB})\right]$
```

We can use a progressive algorithm like FIND-INITIAL-M-G to automatically make progress among a variety of possible scientist interventions as well. Suppose we have methods *a*, *b*, and *c* at our disposal for the scientist algorithm. We can alternate between all three of them to see which will be the most effective. An obvious approach would be to merely run all three methods in parallel, by time-slicing, and thereby incur at most a threefold penalty versus using the optimal approach. This is similar to FIND-INITIAL-M-G. A more sophisticated approach would be to try the three methods for a short while, and to discriminate between them based on how good the discovered solutions are. We could then allocate effort based on the results of experiments already performed. Of course we encounter the pitfalls of estimating success probability from best achieved fitness pointed out in Chapter 3. Better still would be to try out all three methods serially, and use good partial solutions achieved with one method to improve the performance of the others. This approach is the topic of the next chapter.

This page is intentionally left blank.

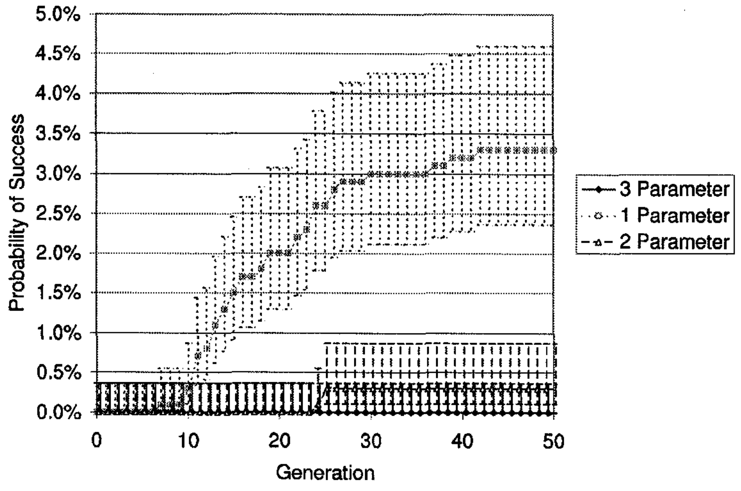
Chapter 7: Techniques That Can Be Used With the Scientist Algorithm

The scientist algorithm introduced in Chapter 2 presents a way to extend and use the functionality of genetic programming to make concrete progress on harder problems. We presented there the idea that we could envision a sort of toolkit of little algorithms and pieces that the scientist algorithm could use while working on a problem. In this chapter, we will present three "tools", or algorithms, which will be useful for the scientist algorithm. For some of the tools, we will also show experiments to demonstrate the utility of the tools on some aspects of hard problems. The three "tools" we will consider are the use of multiobjective optimization in genetic programing; principal components analysis to reduce the dimensionality of multiobjective programs; and a new idea we call incremental evolution.

On the Use of Multiobjective Methods in Genetic Programming

Multiobjective methods can be very useful in standard genetic programming. One problem in genetic programming is that of retaining good solutions to problems. Retaining good solutions is normally accomplished using elitism, which is the process of copying a few elite individuals over into a new population when it is formed. Elitism has the downside of reducing the diversity of the population and thereby forcing premature convergence, which can result in poorer solution quality. To see how multiobjective methods can be useful for genetic programming, we have engineered a simple example that illustrates some of the gains possible with a little manipulation of the standard GP paradigm.

Figure 1



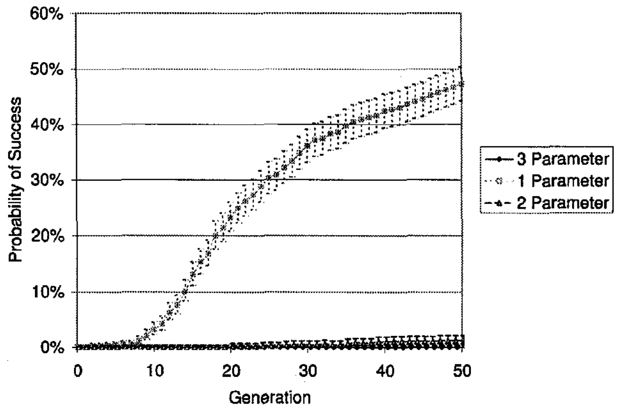
This diagram illustrates the observed difference in success probability when regressing a one-, two- and three-parameter Gaussian function. The one-parameter Gaussian fixes $\mu = 0$ and $\sigma = 1$, while the two-parameter Gaussian fixes only $\sigma = 1$. The data from 1000 trials of each treatment are shown; we observed 33 successes in the one-parameter case, 3 successes in the two-parameter case, and 0 successes in the three-parameter case. The same function and terminal sets were used in each case. The population size was held constant at 4000 individuals per generation, and no elitism was used.

Consider the generalization task for the Gaussian symbolic regression problem of Chapter 1. In Fig. 1, we see the probability of success for fitting the one-parameter (x varies), two-parameter (x and μ vary) and three-parameter (x, μ and σ vary) Gaussian functions. As more parameters are added for a given problem, we see a greatly reduced probability of success and hence greatly increased work-to-success.

The computed 99% success computational effort point statistics for the three settings are 18 million for the 1-D problem, 150 million for the 2-D problem, and infinity for the 3-D problem. Since the counting statistics are quite poor (we have 33, 3 and 0 successes, respectively), finding a valid confidence interval for the computational effort will be challenging. Instead, we take a hint from Koza [Koza 1992], and relax the success criterion somewhat.

Rather than defining success as exactly matching the target function for each of the 50 test cases, we can use an arbitrary "success" criterion for closeness. Fig. 2 shows the success probability of an approximate version of the task as a function of generation number. This easier task can give us a relative idea of how challenging the problem really is: we achieved much higher success rates for the 1-D problem, modestly higher for the 2-D problem, and still 0% for the 3-D problem.

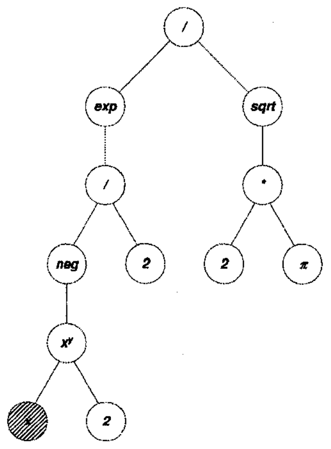
Figure 2



This diagram illustrates the observed difference in approximate "close hit" probabilities when regressing a one-, two- and three-parameter Gaussian function. We define a "close hit" as approximations with total error on 50 regression points less than 0.1. Notice the change in vertical scale as compared with Fig. 1. As in Fig. 1, the one-parameter Gaussian fixes $\mu = 0$ and $\sigma = 1$, while the two-parameter Gaussian fixes only $\sigma = 1$. The data from 1000 trials of each treatment are shown; we observed 473 successes in the one-parameter case, 12 successes in the two-parameter case, and no successes in the three-parameter case after 50 generations. The same function and terminal sets were used in each case. The population size was held constant at 4000 individuals per generation, and no elitism was used.

Reasonable confidence intervals for the computational effort of this easier problem are 1.27 million [1.14 million, 1.41 million] for the one-dimensional problem; 70 million [39 million, 125 million] for the two-dimensional problem; and ∞ [254 million, ∞] for the three-dimensional problem. We can see a real possibility to improve the mean performance by making progress on one parameter at a time. Suppose that we could solve the two-dimensional problem *using* the solution to the one-dimensional problem. If we manage the improvement using the same amount of work as it took to solve the 1-D problem in the first place, we would see an improvement of a factor of roughly 30 in effort. This process might then be performed again, to solve the previously insoluble three-dimensional problem.

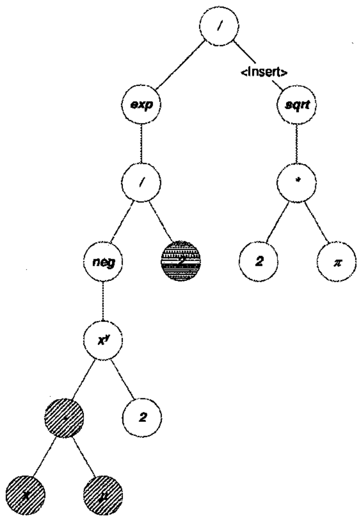
Figure 3



A solution to the one-dimensional Gaussian problem. When the hatched nodes are replaced as in Fig. 4, a solution to the two-dimensional Gaussian is produced.

The back-story for why we have chosen to look at the Gaussian problem can now be revealed: we can showa series of successful solutions to the 1-D, 2-D and 3-D problems. We can then infer what would be required to go from one solution to the next, by hand, to illustrate how the process might be automated. For instance, the human solution of the one-dimensional problem shown in Fig. 3 can be promoted to a solution of the two-dimensional by a single substitution of a 3-node subtree as in Fig. 4. This can then be transformed into a solution of the full three-dimensional problem by replacing a 1-node subtree with a 5-node subtree on the left branch, and by inserting a two-node inter-tree on the right branch as in Fig. 5. Since these subtrees are small, it seems likely that they will be available for GP to use as recombination targets. We can use the tree enumeration functions of Chapter 5 to show that there are 266 trees of size 3 or smaller, including the replacement tree of Fig. 4, and 19 635 trees of size 5 or smaller, such as the

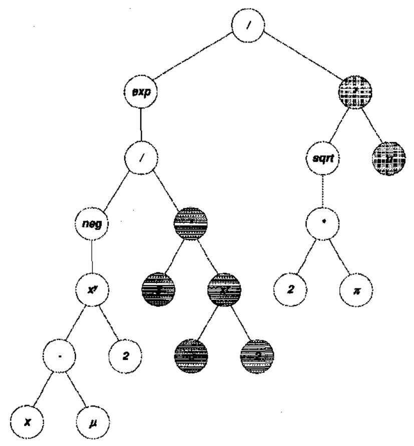
Figure 4



A solution to the two-dimensional Gaussian problem. This tree was produced by a substitution of the diagonally hatched nodes from the one-dimensional solution shown in Fig. 3. Further, a solution to the three-dimensional Gaussian problem can be achieved by replacing the shaded node and by inserting a subtree in the position marked "Insert", as in Fig. 5.

replacement tree on the left branch of Fig. 5. Of course, rotations and mirror image functions ensure that at least four functionally equivalent trees exist that compute the required subfunction in the second case. We can naively estimate the probability that a crossover from a random subtree will contribute a correct subtree as the inverse of number of small trees, multiplied by the number of target nodes available. A quick side computation produces the result $266 \cdot 12 \approx 3\,200$ in the first case, and $\frac{19\,635}{4} \cdot 12 \approx 70\,000$ evaluations in the second attempt. This compares favourably with the roughly 70 million and more than 250 million attempts that would be required in an *ab-initio* regression.

Figure 5



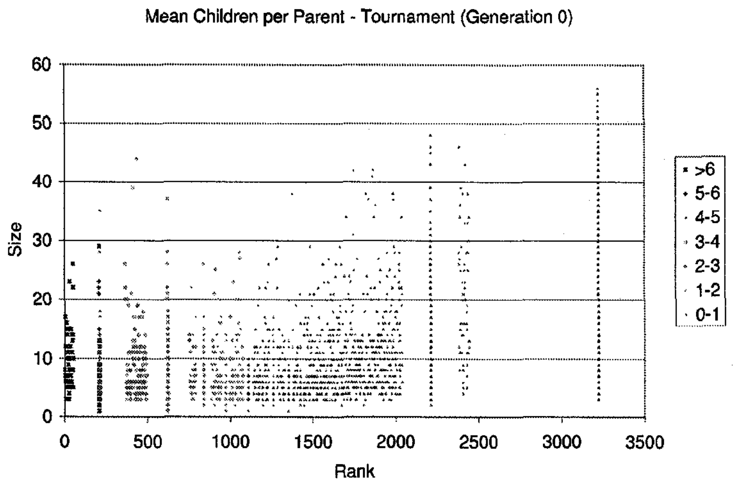
A solution to the three-dimensional Gaussian problem. This tree was produced by substituting the dense horizontally hatched nodes from the two-dimensional solution shown in Fig. 4, as well as inserting the shaded nodes in the tree on the right branch.

Now, the question becomes one of mechanics. Genetic programming has no foresight; thus all positive moves must be strictly local gains or they will not be retained. Notice that this is different from saying that only local improvements are pursued, such as in a greedy hill-climbing application. In a standard GP with elitism, the single best individual is retained, and is then available to beget child trees in the next generation. However, we now encounter a problem. Consider the case where we have a perfect solution to the one-variable problem, and we now want to generalize our solution to additionally solve the two-variable problem. For instance, we might add a few new test cases to our test set, drawn at random from the two-dimensional problem. However, we need to retain perfect solutions on the existing one-dimensional test cases, while simultaneously trying to optimize some two-dimensional test cases. A conventional "add up all the fitnesses" approach will probably not work, since we will likely lose the perfect solution that we attained by sacrificing some of the performance on those test cases for increased performance on the new test case. This process has much in common with multiobjective optimization, which brings us to the title of this chapter.

Multiobjective optimization, in the context of evolutionary computing, is normally concerned with satisficing. In satisficing, there are normally two or more desirable output dimensions, such as speed and program size. Tradeoffs intrinsic to the problem mean that improved values on one dimension result in reduced performance on another dimension. We can then define the Pareto frontier [Pareto 1906] as the set of discovered solutions where no point meet or exceed the solutions' performance on all available dimensions.

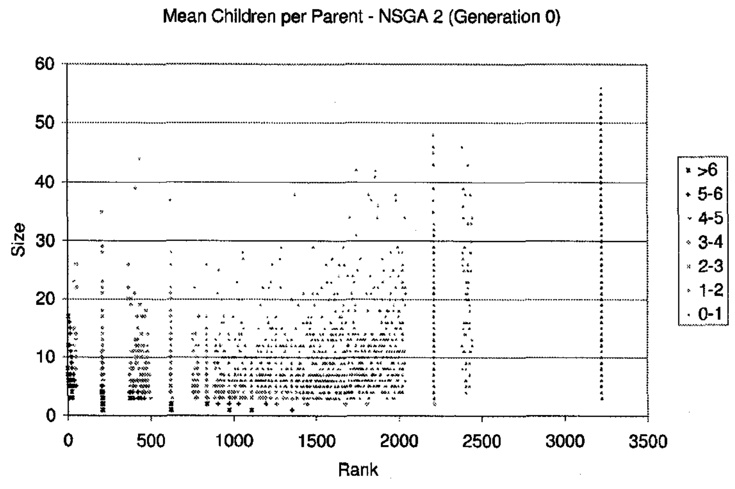
We will use the successful algorithm NSGA-2 [Deb 2001] as an example of a multiobjective optimization algorithm. Two major modifications to the standard genetic algorithm are required for multiobjective optimization (MOO for short). The first is in how fitness is allocated. The second is in how elitism is handled. First, fitness allocation. In NSGA-2, all points in the Pareto frontier are given first rank, and are sub-ranked by the distance to the neighbouring points on the frontier, such that more sparse points outrank denser points. All these points on the Pareto frontier are eliminated from the point set; and a new Pareto frontier is constructed from the remaining points. The process is repeated until all points are given a relative rank. Once the points are all ranked, tournament selection proceeds as normal.

Figure 6



A two-dimensional representation of the first generation of the 1-variable Gaussian problem. The raw fitness values are ranked and shown on the X-axis; the actual tree size is shown on the Y-axis. Also indicated is the mean number of children for each individual, assuming tournaments of size 7.

Figure 7



A two-dimensional representation of the first generation of the 1-variable Gaussian problem. The raw fitness values are ranked and shown on the X-axis; the actual tree size is shown on the Y-axis. Also indicated is the mean number of children for each individual, assuming tournaments of size 7. Here, the selection algorithm from NSGA-2 is used to allocate children.

We can illustrate this process with an example; in Fig. 6, we have diagrammed the initial generation of a one-variable Gaussian problem, as a two-dimensional graph of size and fitness. The points are shaded by their relative probability of selection. This is proportional to the mean number of children they will beget in the next generation.

This can be compared with Fig. 7, where we have used NSGA-2's algorithm for selection to determine the intensity of the points. Notice that now points with small size as well as points with high fitness are productive in the next generation.

The second change that a multiobjective algorithm requires over a standard one-dimensional optimization algorithm is in the behaviour of elitism. The goal of any standard multiobjective algorithm is to fill out, as much as possible, the uncovered Pareto frontier of the function. To accomplish this task efficiently, we must preserve discovered solutions that are at the Pareto frontier so that we don't lose good solutions. Multiobjective algorithms such as NSGA-2 typically maintain the Pareto frontier at each generation by copying individuals in the Pareto frontier directly into the next generation. There are other techniques used in different methods, but NSGA-2's will suffice for our purposes here.

Using the methods of multiobjective techniques such as NSGA-2 in our context can solve the problem we mentioned earlier. We are certain to retain our individuals that have solved the one-dimensional problem, even as we seek improved performance on secondary objectives.

There is a caveat, however, when dealing with multiobjective techniques. Due to geometrical considerations, the surface area-to-volume ratio of a high-dimensional shape becomes larger as the dimensionality increases. We can see this effect by considering the equations for the hyper-volume and hyper-surface area of a hypercube. The volume of a hypercube is trivially given by (1). A hypercube of dimension d has $2d$ faces, so we obtain (2) for the surface area of a hypercube.

$$V(x,d) = x^d \tag{1}$$

$$S(x,d) = 2dx^{d-1} \tag{2}$$

The surface area-to-volume ratio of a unit hypercube is therefore simply $2d$. The surface area-to-volume ratio for a unit sphere is d [Weisstein 2006]. For simple convex shapes generally, the ratio will be $O(d)$, and we find many more points on the surface of a high-dimensional manifold than in the interior. This has a notable consequence that multiobjective methods do not work as well in higher dimensions; essentially very many of the data points end up on the Pareto frontier, which is a surface of dimension $d-1$. With most of the population on the Pareto frontier, many of the points end up competing to be in the first rank due to a purely geometric effect. The selection pressure for high dimensions then quickly decays to nil. Indeed, the situation can often be worse than this, since in many variants we aim for a fixed number of points in the Pareto frontier for each dimension of the search less one. We then find $O(n^{d-1})$ points in the Pareto frontier, which quickly overwhelms the population size, typically on the order of a thousand points. We appear to be in real trouble. For instance, our symbolic regression problem has a fitness score of dimension 50, namely the absolute error at each point. It is for this reason that much of the research in multiobjective optimization takes place with 2- or 3-dimensional data.

We can propose a partial solution to this problem. Instead of performing direct multiobjective optimization with the entire vector of results, we can apply principal components analysis to the problem to lump the high-dimensional data down to a lower dimensionality. This is the subject of our next section.

Principal Components Analysis in Genetic Programming

Principal components analysis (PCA) is the process of determining how sets of data can be clustered by linear combinations of the axes to make a lower-dimensional subspace that holds most of the information of the original data [Hotelling 1953, PlanetMath 2006]. In our context, we will use PCA to reduce the dimensionality of a multiobjective optimization problem, so that two or three principal dimensions remain for us to optimize. PCA is described elsewhere; it consists of diagonalizing the correlation matrix between each pair of variables in a population. The eigenvalues of the diagonal matrix indicate how much of the variables' variability is accounted for by each axis. The eigenvectors corresponding to each eigenvalue are the weightings of the primitive vector that optimally cover the data set. PCA is a greedy algorithm; from an algorithmic point of view, one vector of norm 1 is chosen first, to account for as much of the variability among the data as possible. Then variability along this axis is factored out and another axis, orthogonal to the first, is chosen that covers as much of the remnant variability as possible. The process repeats until all the variability in the original data set is accounted for. In fact, the search is often not performed in a strictly greedy way, but all components are optimized simultaneously. We can, therefore, algorithmically determine the main components or factors that explain much of the data.

The magnitude of the eigenvalues reflects the number of degrees of freedom explained for which the corresponding eigenvector accounts. Of course, we will need to either determine beforehand how many eigenvectors to obtain, or use an adaptive scheme. One common stopping technique is to add new eigenvectors until a single eigenvector accounts for less than a degree of freedom worth of information. To avoid threshold effects, we stop discovering new factors when the magnitude of an eigenvector falls below 0.5 degrees of freedom.

PCA in a GP environment offers the possibility of automatically identifying “good directions” in which to look, since MOAs work best with just a few dimensions.

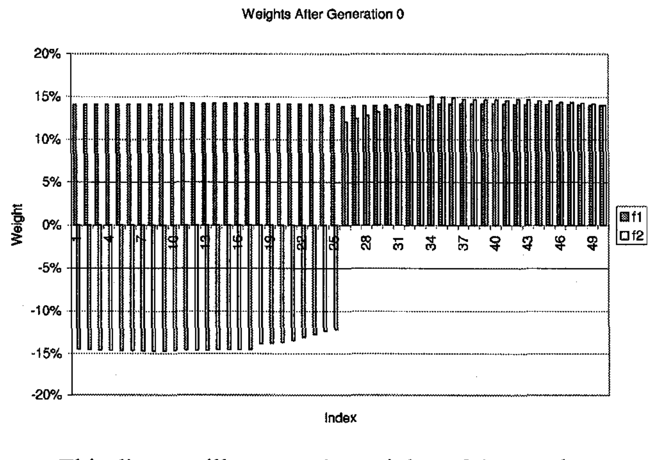
Chapter 7: Techniques That Can Be Used With the Scientist Algorithm

To illustrate the utility of PCA as an effective tool in directing the search process, we performed a principal components analysis of the 50-dimensional fitness values obtained after the initial generation of a GP run. To avoid infinities, we used the relative ranks of the fitness values instead of the raw fitness values as arguments to the PCA.
This gets around many problems such as fitness variables being of different units. Fitness outcomes are often measured on different scales – for instance, the number of faults performed by a program might be measured as a countable integer, while total error might be a real-valued outcome with a potentially infinite range. These fitness variables are measured with different units and their distributions may well be unrelated. Rather than run a PCA on the raw data, we can rank the observed values that we get back from GP's fitness evaluations among each dimension. Nonlinearity of the density of states in the fitness dimensions will also be addressed by taking the ranks in this way.

For these experiments, the demonstration problem is the one-variable Gaussian symbolic regression problem. Fig. 8 shows the weights of the first two factors, which are the only factors that account for more than half a degree of freedom of influence.

We can see some interesting structure in these weights already: the first factor, *f1*, reflects general fitness; it is essentially equivalent to the usual sum of fitness values. The

Figure 8

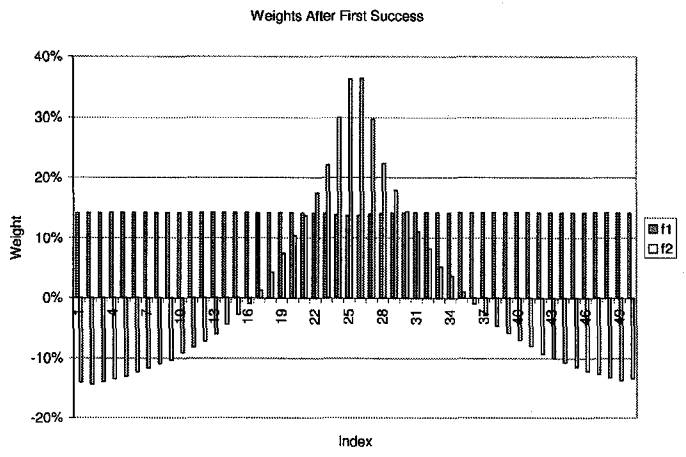


This diagram illustrates the weights of the two largest factors in the first generation of the 1-parameter Gaussian problem discovered by principal components analysis.

second factor reflects the fact that the argument of the one-dimensional Gaussian is negative for test cases 1 through 25, and is positive for test cases 26 through 50. Principal components analysis in this case is using variability in the fitness values of randomly-generated test functions to discover this second factor.

We performed the same procedure on an evolved population, here generation 42 of the same run shown in Fig. 8. Generation 42 is the first generation to show a perfectly successful individual on this problem, although by this time, much of the population is already highly fit. The factor weightings are shown in Fig. 9; again, only two eigenvectors explained most of the data.

Figure 9

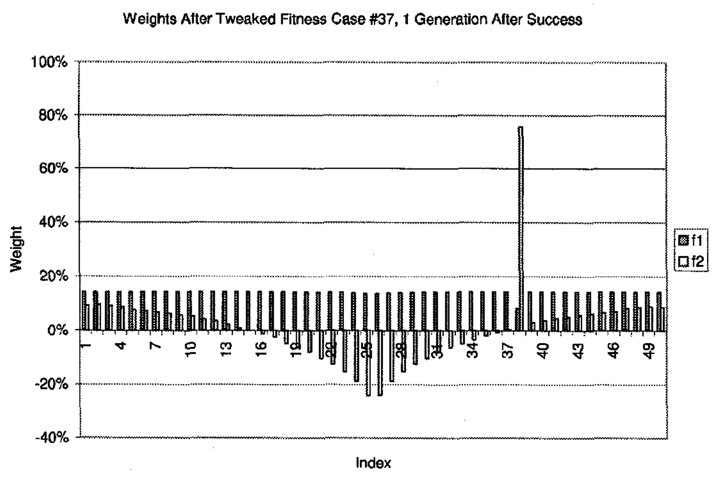


This diagram illustrates the weights of the two largest factors in the 42nd generation of a particular run of the 1-parameter Gaussian problem, discovered by principal components analysis. The 42nd generation is the first generation that discovered a perfectly successful individual in this run.

We see that, again, the first factor corresponds to general fitness. However, we can see that the majority of evolved organisms are discriminating based on whether they perform well in central regions of the Gaussian, or in the tails. *f2* shows that a new axis with large positive weights in the central region and modest negative weights in the tails of the distribution maximally discriminates between the fitness values attained in this generation. If you look closely at the graph, you can see a minor artifact of the PCA: since the weightings for factor *f2* are so large in the middle, the weightings for factor *f1* decrease slightly there to ensure that the total weighting for each variable, scaled by its eigenvalue, adds to one.

Finally, we wanted to test the hypothesis that PCA could be useful in selecting factors for simplifying multidimensional optimization. We performed an interesting experiment. After generation 42 of this run, we swapped one of the test cases out for a single test case from a related problem – we used a test case from a two-parameter Gaussian problem instead. The intent behind this experiment is that if PCA is useful in identifying dimensions that are "interesting" to an evolved population, we should see the replaced test case as a clear signal in the column weights. Fig. 10 confirms this suspicion. Notice that in this figure we can see the effects of our "natural" factor *f2* of Fig. 9 superimposed on the strong effect of our experimental modification.

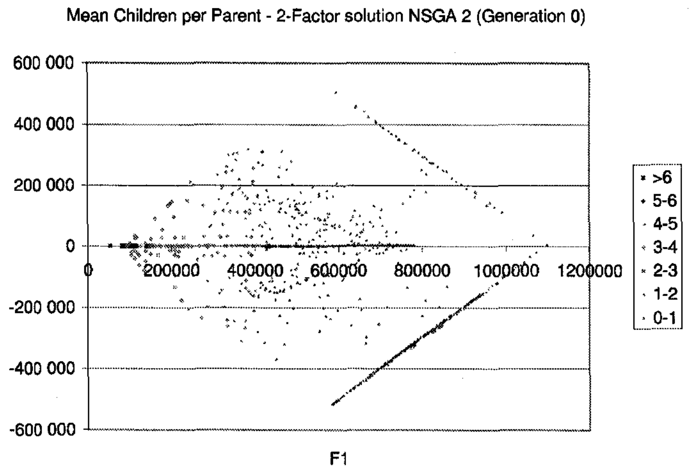
Figure 10



This diagram illustrates the weights of the two largest factors in generation 43 of a particular run, discovered by principal components analysis. This 43rd generation is the generation immediately after the successful generation, but we have replaced the 38th data point of the 1-parameter Gaussian with a data point chosen at random from the 2-parameter Gaussian. Notice the strength of this parameter as identified by the principal components analysis technique.

Now we are in a position to identify the utility of PCA in multiobjective optimization for genetic programming. Our thought is that PCA can be used automatically by the scientist framework to identify a few most "important" or "interesting" dimensions on which to progress. Since multiobjective methods retain elites along the Pareto frontier, we will not throw away individuals that perform the best on either the first factor *f1*, or on any stationary auxiliary factors. However, there is one detail specific to multiobjective optimization that we should point out. In Fig. 11, we have shown the first generation of the one-variable Gaussian problem, evaluated on the new axes *f1* and *f2*.

Figure 11



A two-dimensional representation of the first generation of the 1-variable Gaussian problem. The data of factor *f1* are ranked and shown on the X-axis; the *f2* factor is shown on the Y-axis. Also indicated through colouring is the mean number of children for each individual, assuming tournaments of size 7. Here, the selection algorithm from NSGA-2 is used to allocate children.

Since the diagonalization process allocates columns with negative weights, we find that much of the population lies below zero in terms of performance. This is undesired from the viewpoint of multiobjective optimization, which assumes that smaller values in each dimension are to be preferred. Here, we are trying to minimize the effects of a second dimension, so having a very large negative value of *f2* is not useful. Taking the absolute value would seem to be an obvious remedy, but this has an unintended consequence – the fitness value 0 on *f2* is readily achievable, as seen in Fig. 11. Therefore, this putative solution will tend to eliminate diversity from the population, which is hardly our intent! The goal of moving to a multidimensional representation is to enable the search to explore and populate different regions of the search space – that is, to choose simpler subproblems when progress on the main problem is hard to come by. If the auxiliary dimensions are to be considered as "interesting" properties of the search space, then we should populate them with candidate solutions as well, just to see what we get. Therefore, small absolute values of *f2* are interesting, as are large absolute values of *f2*. Once we have an idea of a figure-of-merit, it is straightforward to make a conventional multiobjective fitness dimension. We might consider small values of *f2* and large values of *f2* worth exploring, and so we can define the auxiliary variables *aux2* and *aux3* from *f2* as in (3). We would then introduce *aux2* and *aux3* as new objectives and remove *f2*.

$$aux2 = -abs(f2) \tag{3a}$$
$$aux3 = abs(f2) \tag{3b}$$

Now we have two additional objectives, which can be treated together or separately. We would expect slower progress towards the main goal, *f1*, since multiobjective methods do evolve more slowly. This new definition of auxiliary variables that are automatically determined can allow for innovative approaches. It also accords well with the scientist framework in general, since we may want to perform little experiments to see which of the new dimension *aux2* and *aux3* are best suited to overcoming fitness plateaux and providing an extra push to GP when it gets stuck.

Consider the fitness weightings shown in Fig. 10. Here, the second factor *f2* measures the degree of interest in finding good solutions to the newly introduced task, that of performing well with a changed mean μ. This suggests a new line of attack: use multiobjectivity to introduce new problem types or subtypes to existing solutions.

Adding Additional Tasks as New Objectives for GP

Recall from our discussion of fitness records that one of the insights from this thesis is that we should break down problems into the most basic cases possible. We then are left with the problem of composing solutions. It may be the case that composing solutions can be performed automatically by GP, without any explicit coding necessary on our part. For instance, suppose that a problem *P* breaks down naturally into two simple cases, *P1* and *P2*. If we can solve *P1* and *P2* separately by genetic programming, we can get solutions to each subcase in a restricted environment where problem solution is quite feasible. We can then have GP automatically combine the solutions without human intervention using COMPOSITION-ALGORITHM.

Algorithm 1: COMPOSITION-ALGORITHM

Input: 2 trees t_1 and t_2 that successfully solve two different fitness objectives, as defined by the fitness objectives $f_1(t)$ and $f_2(t)$.

Output: a valid tree *t* that may solve both problems to completion

```
$trees_i \leftarrow t_i \; \forall i = 1..2$
**Population Initialization**: generate $M-2$ random trees and add them in
for $j \leftarrow 3$ to $M$ do
    $trees_j \leftarrow$ RANDOM-TREE
end for
$trees \leftarrow$ MULTIOBJECTIVE-EVOLVE-STARTING-WITH $trees$, $f_1$, $f_2$, $M$, $G$
return CHOOSE-BEST-SATISFIER $trees$, $f_1$, $f_2$, $M$
```

Since multiobjective algorithms such as NSGP-2 never lose the most highly fit individuals on any axis, COMPOSITION-ALGORITHM cannot ever do worse than reproduce t_1 and t_2. Fig. 12 gives an indication of what may happen when COMPOSITION-ALGORITHM is used to combine fit subsolutions. Of interest here is the adaptation of the

Figure 12

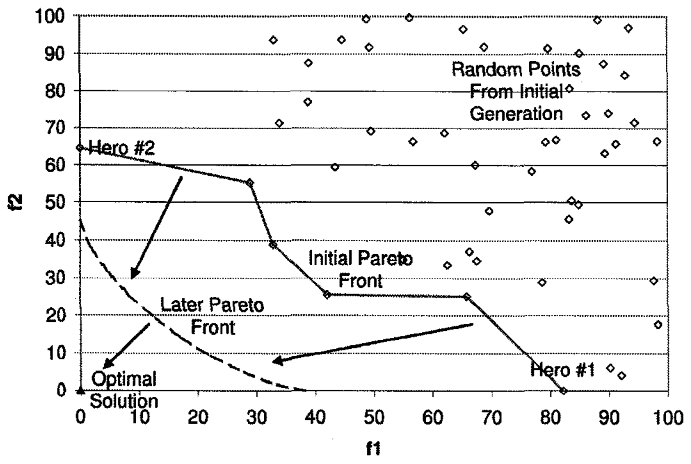


Diagram of the intended dynamical behaviour of COMPOSITION-ALGORITHM. The initial bi-objective population is seeded with two heroes, which represent successful solutions to the two subproblems with fitness values indicated by f_1 and f_2. Random points are generated to fill the population, giving the indicated Pareto frontier of successful solutions. A multiobjective algorithm such as NSGA-2 is then used to evolve the population towards the ultimate objective. This data and graph are schematic only, and do not represent observed data.

NSGA-2 algorithm [Deb 2001] to genetic programming, and of the fact that the true Pareto frontier is the single point in the bottom-left corner of Fig. 12. In multiobjective terms, we would say that the Pareto frontier is degenerate.

While we have not performed these sorts of experiments with COMPOSITION-ALGORITHM, we have performed a related experiment very often with regular and repeated success, which we expect that the reader will find convincing. For the particular case where we have a primary and an anciliary goal, there is an embedding which has the same effect as COMPOSITION-ALGORITHM but that does not require going to a full multidimensional optimization technique. Suppose that we have two fitness objectives $f_1(t)$ and $f_2(t)$. We distinguish this case by preferring progress on $f_1(t)$ over progress on $f_2(t)$. If $f_1(t)$ and $f_2(t)$ are integral-valued and $f_2(t)$ has a maximum achievable value $(f_2)_{\max}$, we might combine the two objectives into a single fitness function, $f(t) = f_1(t) + \frac{f_2(t)}{(f_2)_{\max} + 1}$. This will modify the behaviour of the evolution of the GP, since now GP will prefer individuals by f_2 during the evolution; essentially breaking ties in f_1 by f_2. There is another way to adapt a single fitness function that is a much closer analogy to COMPOSITION-ALGORITHM. Suppose that we use f_2 to break ties *only* when $f_1(t) = \min f_1$. In this case, we are essentially beginning a new phase of evolution once f_1 is satisfied, namely "work on f_2 but not at the expense of your perfect solution to f_1." The new fitness function *f* is shown in (4). This new algorithm is given below as Alg. 2, EVOLVE-TREES-PREFERRED-FITNESS.

$$f(t) = \begin{cases} f_1(t) & \text{if } f_1(t) > \min f_1 \\ f_1(t) + \dfrac{f_2(t)}{(f_2)_{\max} + 1} & \text{otherwise} \end{cases} \tag{4}$$

We can also adapt (4) to work with non-integral values, as in (5). If f_2 has no largest possible value, we can use an arctangent transform to allow our method to work anyway, as in (6). This tactic is generally only possible for those problems for which f_1 is a success-based fitness measure by GP: no successes on f_1 will result in simple conventional one-dimensional evolution on f_1. Since (6) is the most general case, we have used it for EVOLVE-TREES-PREFERRED-FITNESS.

$$f(t) = \begin{cases} f_1(t) & \text{if } f_1(t) > \min f_1 \\ f_1(t) - (f_2)_{\max} + f_2(t) & \text{otherwise} \end{cases} \tag{5}$$

$$f(t) = \begin{cases} f_1(t) & \text{if } f_1(t) > \min f_1 \\ f_1(t) + \arctan f_2(t) - \dfrac{\pi}{2} & \text{otherwise} \end{cases} \tag{6}$$

Algorithm 2: EVOLVE-TREES-PREFERRED-FITNESS

```
Input:  population size $M$, number of generations $G$, tournament size $T$;
        a primary fitness operator $f_1 : \mathbb{T} \rightarrow \mathbb{R}$ for which smaller values represent better
        solutions; $\min f_1$, which is the best possible value of $f_1$; an ancillary fitness
        operator $f_2 : \mathbb{T} \rightarrow \mathbb{R}$ which is to evaluate progress once $f_1$ is satisfied
Output: a tree $t \in \mathbb{T}$ which has a small fitness on $f_1$ and, if $f_1$ is satisfied, has a small
        fitness on $f_2$.

        Let $f(t) = \begin{cases} f_1(t) & \text{if } f_1(t) > \min f_1 \\ f_1(t) + \arctan f_2(t) - \frac{\pi}{2} & \text{otherwise} \end{cases}$
        return EVOLVE-TREES $M$, $T$, $G$, $f(t)$
```

We mentioned above that we have used EVOLVE-TREES-PREFERRED-FITNESS extensively in our research in this thesis. We have used it primarily for parsimony pressure. This technique nearly always reduces functions to nearly optimal tree sizes; indeed, it has often produced shorter trees than we were able to write by hand. We have used it both with an existing population, and as a follow-on step to make parsimonious trees when using the scientist algorithm. The later version is given as an explicit algorithm in EVOLVE-TINY-TREES, which can be used verbatim in SCIENTIST-ALGORITHM. Here we have used values of G, T, and M, indicated by the subscript *shrink*, that are optimized for what we call "shrinkwrapping", the process that EVOLVE-TINY-TREES adds to conventional optimization.

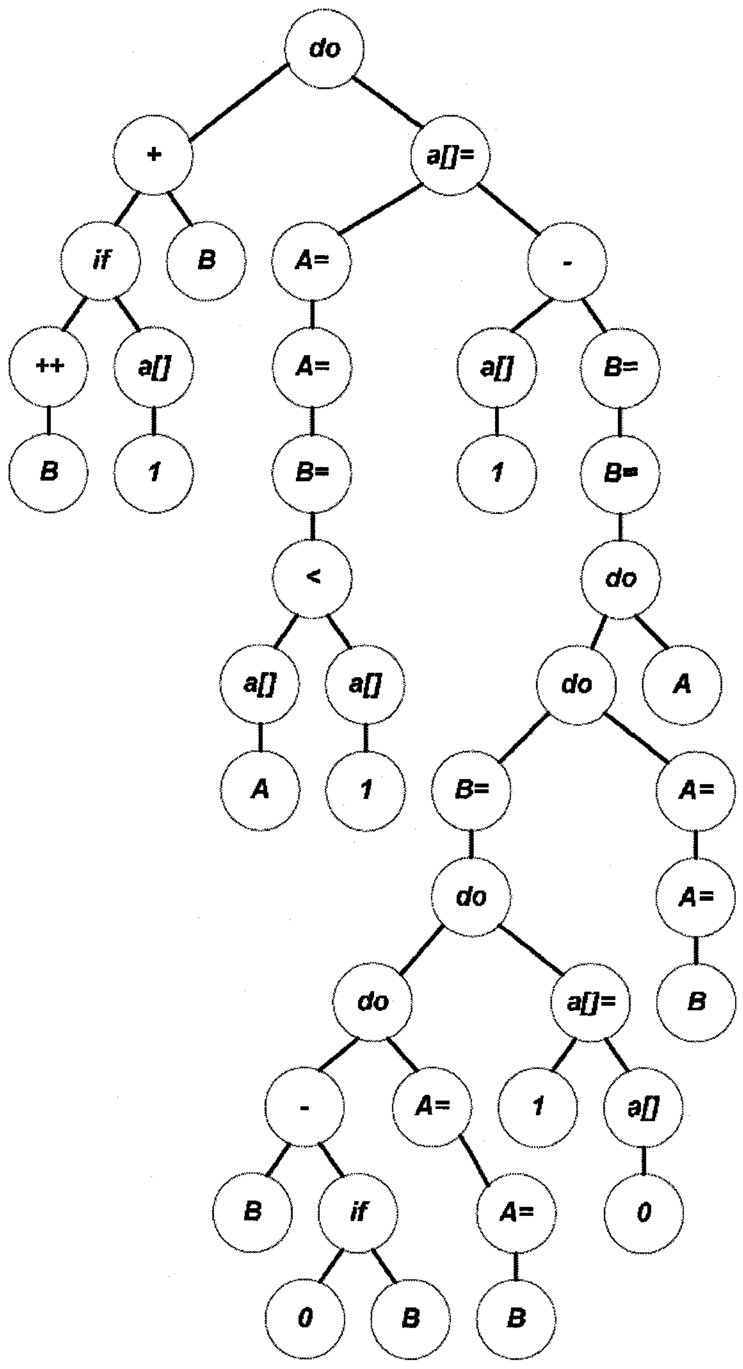
Algorithm 3: EVOLVE-TINY-TREES

Input: population size M, number of generations G, tournament size T;
a fitness operator $f : \mathbb{T} \to \mathbb{R}$ for which smaller values represent better solutions; $\min f$, which is the best possible value of f
Output: a tree $t \in \mathbb{T}$ which has a small fitness on f and, if f is satisfied, is very small.

```
hero ← EVOLVE-TREES M, T, G, $f(t)$
Re-evolve: prepare to evolve again seeding hero in a new population
$trees_1 \leftarrow hero$
Population Initialization: generate $M-1$ random trees and add them in
for $j \leftarrow 2$ to $M$ do
    $trees_j \leftarrow$ RANDOM-TREE
end for
Shrinkwrapping: now evolve as usual with the modified fitness function $f_{new}$
Let $f_{new}(t) = \begin{cases} f(t) & \text{if } f(t) > \min f \\ f(t) + \arctan size(t) - \frac{\pi}{2} & \text{otherwise} \end{cases}$
return EVOLVE-TREES-STARTING-WITH $trees, M_{shrink}, T_{shrink}, G_{shrink}, f_{new}$
```

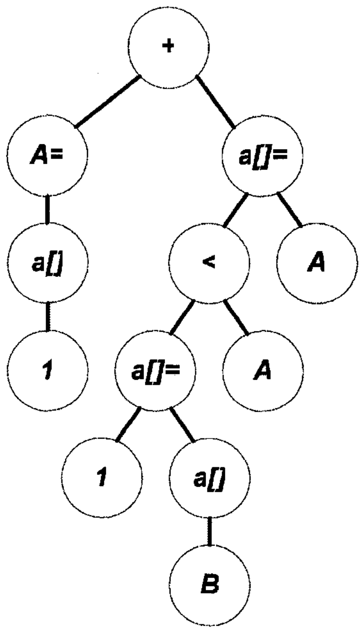
Figs. 13 and 14 show one example using EVOLVE-TINY-TREES to shrink a very fit tree. Fig. 13 shows a typical tree that is highly fit evolved by GP for a problem of sorting an array of integers of size 2. It has 43 nodes. It is reduced by shrinkwrapping into the solution of Fig. 14, which has 12 nodes. It is clear that this algorithm greatly increases the readability and parseability of the code.

Figure 13



A highly fit tree generated by standard genetic programming. This program correctly solves the problem of sorting an array of two integers. A and B are variables, given the values 0 and 1 at evaluate-time.

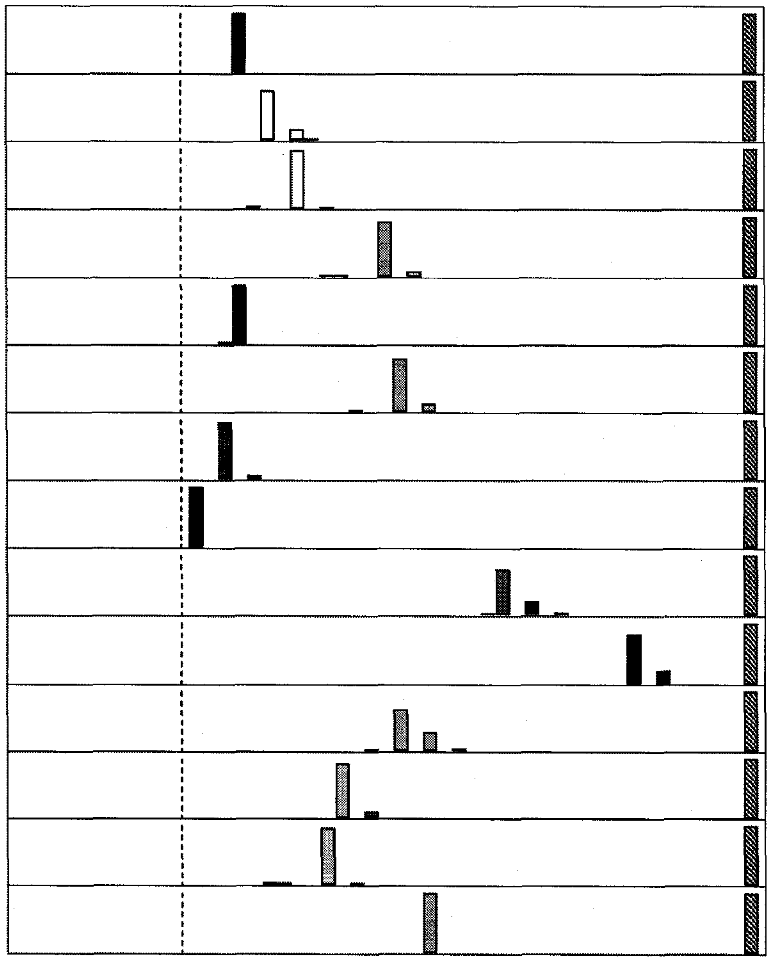
Figure 14



A typical result of shrinkwrapping the highly fit tree of Fig. 13. This program correctly solves the problem of sorting two slots in an array whose addresses are indicated by A and B.

To demonstrate the usefulness of this technique, we performed a simple experiment. We generated, using conventional genetic programming, 274 correct solutions to the artificial ant on the Santa Fe trail with 600 time steps allowed. Of these 274 successful solutions, we chose the 14 that were of the largest size allowed in our system, 50 nodes. For each of these 14 runs, we performed 100 runs of the re-evolve section of EVOLVE-TINY-TREES. We used a population size $M_{shrink} = 7$, tournament size $T_{shrink} = 7$, and $G_{shrink} = 320$ generations. These choices were made from work described later on in this section. In Fig. 15, we show the distribution of 100 independent runs starting from each of the 14 maximum-size correct trees. Fig. 15 also indicates the smallest possible size achievable for the Santa Fe ant problem, as discovered by the enumeration procedure of Chapter 5. Clearly, the algorithms generated by GP have a certain minimum complexity associated with them; however, it is interesting to note that one of our 14 tests shrank down to one of the 48 second-smallest

Figure 15


Distribution of the sizes of successful Santa Fe ant solutions before and after calling EVOLVE-TINY-TREES. The x-axis measures tree size in nodes; the scale is 0 to 50. The hashed bar on the far right is the initial distribution at size 50. The histograms are the distributions of outcomes after 100 runs using the settings described in the main text. The dotted line on the left is the smallest possible tree, of size 11, that solves the artificial ant on the Santa Fe trail problem.

trees achievable. Of course, in each of the 14 cases, the correctness of the fit solution was preserved.

It is also possible to use this trick to optimize for other secondary properties other than the size of a successful tree. Much human programming tends to use the more common symbols and references, such as 0, rather than using unusual symbols and expressions, such as $x - x$. In some of our work, we wanted to generate trees that were both small and of low entropy. Parsimony pressure might elide these strange artifacts away, but we were looking for a more direct way to simplify programming code. We can coerce GP into generating a simple, parseable version of the code by rewarding trees that have a small diversity of node types. To do this, we walk the tree and accumulate the number of times each node is used in the source tree. We can use the tree manipulation nomenclature from Chapter 5 to define this vector of node type counts rigorously. We previously defined three vectors of use in completely describing a given function tree. The arity vector a of a problem stores the number of nodes types of each arity. The geometry vector g of a tree stores the vector of node arities in a pre-order traversal of a tree. We also defined the labelling vector l of a tree as the vector of node alternatives chosen for each tree element, numbered consecutively from 0 to $a_i - 1$. We will need a way to uniquely number each kind of node, regardless of its arity. This is readily achieved by the following procedure. We give all the arity-0 nodes their own labelling numbers, 0 to $a_0 - 1$. Arity-one nodes then add the count of arity-0 nodes to their labelling number, giving numbers in the range a_0 to $a_0 + a_1 - 1$. The process continues likewise until all distinct node types have been labelled. Mathematically, (7) defines a label mapping function *m* that takes an arity vector *a*, an arity of the node *arity*, and a node label *label*. *m* then provides a unique number for each node type in a tree.

$$m(a, arity, label) = label + \sum_{i=1}^{arity} a_i \tag{7}$$

We can then compute the number of occurences of each node type in a tree using (8), where s is the size of the tree.

$$v_i = \sum_{j=1}^{s} \begin{cases} 1 & \text{if } m(a, c_j, l_j) = i \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

This vector v of node weights is then used to determine the entropy of the new tree. Let $a_{\max}$ be $\sum_{i=0}^{|a|} a_i$. We define the normalized size-weighted entropy e as (9). The normalization ensures that $e = s$ when the entropy is maximized; that is, when each node type appears the same number of times.

$$e = -s \frac{\sum_{i=0}^{a_{\max}} \frac{v_i}{s} \ln \frac{v_i}{s}}{\ln s} \tag{9}$$

An advantage of using a size-weighted entropy measure like this is that we can weight the node types to include a preference or aversion for certain nodes. This is readily accomplished by modifying (9) to include a "weight" w_i for each node type. This gives us (10), where we have performed a weighted normalization. This weighted version of the equation will be useful to us in the next section.

$$e = -s \frac{\sum_{i=0}^{a_{\max}} w_i \frac{v_i}{s} \ln \frac{v_i}{s}}{\sum_{i=0}^{a_{\max}} \frac{w_i}{s} \ln \frac{1}{s}} \tag{10}$$

We have performed some experiments to determine good settings for the variables M_{shrink} and G_{shrink}, using the unweighted size-weighted entropy measure of (9). Using a constant value of $T_{shrink} = 7$, we looked at two different shrinkwrapping problems. The first was an "easy" task where a tree of size 49 was shrunk to minimal entropy and tree size using (9). The final tree is 9 nodes in size. It was called "easy" because the task was relatively easy, and so fitness-preserving transformations weren't difficult to come by. The second task was a "hard" task where a semantically different tree of about the same size was shrunk to an optimal size-weighted entropy. The fitness function in the second case was more challenging, requiring on average 9 times more computational effort to find a solution than the "easy" problem. The shrinkwrapping operation might be expected to be more difficult. The final tree in this case is 10 nodes in size. In both cases, EVOLVE-TINY-TREES was used, save that only a single run of the initial evolution was performed for all subsequent shrinking operations. For the "easy" problem, we required $1\,270 \pm 30\%$ fitness evaluations to shrink the successful program to optimality, 50% of the time. For the "hard" problem, we needed $1\,900 \pm 35\%$ fitness evaluations to shrink the successful program to optimality, 50% of the time. To get the conventional 99% success probabilities, multiply these scores by $\frac{\ln 0.01}{\ln 0.5} = 6.64$. Our best settings for the two problems were determined by a procedure similar to that performed in Chapter 6 – that is, we varied the population size across an exponentially distributed set of values around what we found to be a good parameter setting. Unlike Chapter 6, we performed independent runs in the number of generations, since so little time was required for each run. The best settings for 99% success of the "easy" problem were found by performing

4 independent runs of $M = 5$ individuals for 320 generations. The best settings for 99% success of the "hard" problem were found by performing 8 independent runs of $M = 7$ individuals for 220 generations. Perhaps predictably, in each case the behaviour around these optimal settings is fairly tolerant of small variations. We can therefore recommend some intermediate settings such as $M = 7$, $G = 320$, and 11 independent runs. This requires twice as many fitness evaluations as the optimal choice for the "hard" problem above, and aims for a 99% success rate overall. It is interesting to note that for the shrinkwrapping problem, optimal performance for GP is achieved by a kind of hill-climbing operation. We can tell this by noting that the optimal success probability is achieved when the population size is small and many generations are performed. This suggests that many relatively easy incremental steps are best for GP's success. It is likely that the evolution here proceeds stepwise through many small modifications, in a kind of hill-climbing.

One final note on multiobjective optimization with respect to program evolution. It seems likely that the combination of principal components analysis with multiobjective optimization will prove to be a valuable addition to the toolkit of available interventions, particularly when we adopt a fitness record approach to fitness measurement, as described in Chapter 2.

Scalable Difficulty, Incremental Evolution, and The Inductive Method for Genetic Programming

We will discuss another important way in which the difficulty of a genetic programming problem may be lowered or raised, by evolving a partial solution to a hard problem on an easy version of the problem. We then can try to extend our easy solution into the more challenging complete domain. For instance, O'Reilly showed that sorting an array is easy if the swap primitive is included in the function set; and difficult otherwise [O'Reilly 1995]. Other work on evolving sorting in genetic programming was done by Kinnear [Kinnear Jr. 1993]. One sort of progressive-challenge protocol that the scientist algorithm can use is to first solve the sort problem for an array of size 2. The solution to this problem is simply conditional-swap; swap the two elements if and only if they are out-of-order. The scientist algorithm might then generalize this successful solution to a subroutine taking two parameters, namely the indices of the two data elements used in the swap. If correctly generalized, this primitive would then be highly productive in evolving a general sorting algorithm from scratch. Notice that this progressive-challenge framework is useful for many problems of interest to computer scientists; indeed, it is the principal technique by which advances in computer science are made.

To determine whether these operations can actually work in practice, we implemented the previous example and performed experiments to quantify exactly how beneficial the progressive-challenge approach is on this problem, in terms of computational effort. First, let us describe the problem.

The Integer-Sorting Problem

The integer-sorting problem is defined using a problem-generation oracle of the type introduced in Chapter 2. We define the oracle *O*(*n*) as an algorithm that, when provided with a scalar *n*, returns an array of *n* unique integers chosen randomly on the interval $[-2^{31}, 2^{31}-1]$. The genetic programming system is allowed to use the language primitives of Fig. 16 when finding solutions, with some small variations for different tests. As for the most common case in genetic programming, all nodes return a value, and there is exactly one data type used for all function inputs and outputs. For our problem, integers are a natural choice for this universal return type.

Figure 16

Terminal	*Arity*	*Meaning*
T0; T1; etc.	terminal	temporary variable accessors
A	terminal	algorithm-assigned constant
B	terminal	algorithm-assigned constant
N	terminal	array length
Zero; One; Two	terminal	constant values
GetAt *x*	1	array accessor
SetT0 *x*; SetT1 *x*	1	temporary variable assignment
Negate *x*	1	arithmetic negation
Add *x*, *y*	2	addition
Sub *x*, *y*	2	subtraction
SetAt *x*, *y*	2	array element assignment
If>0 *condition*, *clause*	2	conditional evaluation
Simple-For *variable*, *start*, *end*, *clause*	4	looping primitive

Functions and terminals for the integer-sorting problem. Multiple terminals and functions of the same kind are indicated on the same row. All functions and terminals return an integer; exact semantics are described in the main text.

A few of the functions indicated in Fig. 16 deserve explanation. The terminals T0, T1 access per-run variables in the function tree, and are initialized with 0 on evaluation state. A and B will be explained in the context of evaluation. N is a read-only constant storing the size of the current array, and the named integers Zero, One and Two are constant literals. The temporary variable assignment operators SetT0(*x*) and SetT1(*x*) return the value assigned to the new variable, as does the array assignment operator SetAt(*x*, *y*). The array accessor function GetAt(*x*) returns the value of the array at address *x*. This can force an out of range error, which is tracked separately in a per-evaluation data structure. In the case of an error, the index *x* is returned. The arithmetic operations Negate, Add, and Sub have their normal behaviour on signed 32-bit integers. Since there is no explicit sequencing operator, Add and Sub are often used by GP for their side-effect of evaluating the two operands in sequential order. The conditional operator If>0(*condition*, *clause*) invokes special evaluation rules to behave correctly. The condition is evaluated first, and if the result is strictly greater than 0, the dependent clause is evaluated and its result returned. If the dependent clause is not evaluated, the condition is returned instead. The Simple-For operator has 4 parts. The first part, the *variable* identifier, is evaluated for the side effect of touching a variable. Whatever variable is last accessed or named in this subtree is tracked, and is used as the loop iterator. If no assignable variable is touched in this phase – for instance, if the constant Zero is used as a loop variable – an illegal loop variable error is noted, and the function returns the loop subtree value. The initializer is then evaluated, and its return value is used to initialize the loop variable. When the loop executes, the loop maximum subtree is reevaluated at each iteration and the loop exits whenever the loop variable exceeds the loop maximum. If the loop does not exit, the dependent clause is evaluated, and then the loop iterator is incremented. Since we can now easily write an infinite loop, we must be able to arrest slow programs in a timely manner. We define a maximum number of evaluations for which we will allow the program to run, and if this limit is exceeded, we interrupt the evaluation, return the last partially-evaluated result, and note an "evaluations-exceeded" error for the program.

Successful evolution of fit genetic programs lives or dies on the fitness operator. We have tried a fitness-record approach as described in Chapter 2 for this problem. We define four different outcome variables for this problem. If any data position in the array has been modified, we set a fitness flag *isInitialPosition* indicating that we are no longer in the initial position. We then count the number of array positions that are filled with values that were not in the provided array, and store it in a fitness variable *numberWrong*. Let $p(x)$ be the position of the array element $x = a_i$ in the correctly sorted array. Finally, we define a fitness variable *distanceScore* as the sum of absolute distances between the elements of the array and their target position using (11).

$$distanceScore = \sum_{i=0}^{n-1} \left| i - p(a_i) \right| \tag{11}$$

To test using conventional genetic programming to solve this problem, we combined these four fitness elements together to get a conventional fitness function. Let ε be *numberWrong*, the number of unexpected values introduced during fitness evaluation. We then define the fitness function *f* using (12).

$$f(a) = \begin{cases} n^2 & \text{if } \mathit{isInitialPosition} \text{ and } \varepsilon + distanceScore > 0 \\ n\varepsilon + distanceScore & \text{otherwise} \end{cases} \tag{12}$$

This is a fairly conventional definition; the largest possible contribution to the distance score is achieved when data that should be in the first position appears in the last position, for a contribution to *distanceScore* of $n - 1$. Accordingly, we score a fitness value of *n* for each unexpected value that occurs. If no variable is touched, we score a fitness value equivalent to discovering that *each* value in the array is unexpected, unless the array happens to be perfectly sorted to begin with. In this case, we should leave the array alone, and so we assign a fitness score of 0 if the program correctly does nothing. With respect to the environment of randomly generated test cases, we can choose the number of test cases over which to evaluate the data dynamically. We will use the variable t to indicate the number of test cases to consider when evaluating candidate individuals. For the single scalar fitness value case, we simply added the fitness scores achieved on each test case together to get a composite score.

Chapter 7: Techniques That Can Be Used With the Scientist Algorithm

Incremental Evolution

To see if the integer sorting problem is soluble using standard GP, we first tried solving it directly. No successes were found in 1000 runs for an array of size $n = 10$ using 20 randomly-generated test cases. We then considered evolving solutions to smaller problems first. Since problems of size $n = 1$ are trivially solved by any program that does not touch the accessed array, we considered the $n = 2$ problem next. Using 6 randomly-generated test cases, we were able to obtain a considerable number of successes from some random genetic programming runs. For testing purposes, we dropped the Simple-For operator from the function set, since no looping was expected to be necessary. The solutions discovered seemed to cluster into three distinct algorithms (after simplification), which are shown in Fig. 17. Of interest is that genetic programming automatically discovered two ways to sort two distinct numbers that do not involve the use of a temporary variable. For the second method of Fig. 17, the stack-store swap, the processor stack is used as a temporary variable.

Figure 17

Method	***Evolved code***	***Usual C form***	***Algorithm***
1	(If>0 (- (getAt 0) (t0= (getAt 1))) (- (setAt 1 (getAt 0)) (setAt 0 t0)))	`t = a[1]; if (a[0] > t) { a[1] = a[0]; a[0] = t; }`	conditional auxiliary variable swap
2	(If>0 (- (getAt 0) (getAt 1)) (setAt 1 (+ (getAt 0) (If>0 (setAt 0 (getAt 1)) 0))))	`if (a[0] > a[1]) { a[1] = a[0] + (a[0] = a[1]) > 0 ? 0 : a[0]); }`	conditional stack-store swap on positive a[]
3	(If>0 (- (getAt 0) (getAt 1)) (setAt 0 (- (getAt 0) (+ (neg (getAt 1)) (setAt 1 (getAt 0))))))	`if (a[0] > a[1]) { t = -a[1] + (a[1] = a[0]); a[0] = a[0] - t; }`	conditional arithmetic swap

Three different automatically evolved algorithms for sorting two integers correctly. The evolved code has been shortened and simplified using EVOLVE-TINY-TREES. The algorithms have been classified into the three categories listed here based on their functional characteristics.

The specific version of the algorithm appearing in Fig. 17 requires that the array elements be positive to work correctly, as they were when this algorithm was evolved. For the third method, the arithmetic swap, the sum $a[0]-a[1]$ is computed as an intermediate variable; the algorithm then recovers $a[1]=a[0]-\left(a[0]-a[1]\right)$ to complete the swap. It is perhaps interesting that while we had heard of the arithmetic swap trick, the stack-store swap was unknown to the author. This is another case where genetic programming discovered a solution unknown to the implementor.

Having determined that genetic programming can automatically evolve solutions to what we might call the sort-2 problem, we determined the computational efficiency and optimal settings for this problem using the exponential parameter setting procedure of Chapter 6. Approximately optimal values were attained using $M = 2\ 500$, $G = 20$, and 81 runs to get 99% confidence intervals. The expected number of runs to get a 50% success rate with these settings is therefore 640 000 fitness evaluations. We can conclude that for this problem and representation, even sorting 2 array elements correctly from scratch is non-trivial.

We were then curious as to see whether we could reduce the computational effort required to solve this problem still more. As it happens, for arrays of size 2, there are exactly two kinds of test case behaviour: either the data arrive in-order, or they arrive out-of-order. If the data are in-order, the correct behaviour is to do nothing. If the data are out-of-order, the correct behaviour is to swap the data. Our insight here is that the fitness contrast is sufficiently obvious that an automatic technique such as factor analysis could readily identify the difference between these test cases. The algorithm CLUSTER-

BY-FITNESS evaluates a number of randomly-generated function trees on a set of test cases to determine which test cases would be candidates for grouping together.

Algorithm 4: CLUSTER-BY-FITNESS

Input: population size M, number of generations G, tournament size T;
a vector of n test cases $\mathcal{T} = [t_1, t_2, ..., t_n]$ upon which to make progress, a fitness operator that maps performance by a tree $t \in \mathbb{T}$ on a test case $t \in \mathcal{T}$ to a fitness value $f : \mathbb{T}, \mathcal{T} \to \mathbb{R}$ for which smaller values are better, and an external clustering algorithm CLUSTER that clusters an n x M matrix into some natural clusters

Output: a partitioning $p : \mathcal{T} \to \{\mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_n\}$.

```
**Population Initialization**: generate $M$ random trees
for $j \leftarrow 1$ to $M$ do
    $trees_j \leftarrow$ RANDOM-TREE
end for
**Evaluation**: evaluate the $M$ random trees on each of the $n$ test cases
for $j \leftarrow 1$ to $M$ do
    for $i \leftarrow 1$ to $n$ do
        $fitness_{i,j} \leftarrow f(trees_j, t_i)$
    end for
end for
return CLUSTER $fitness_{i,j}$
```

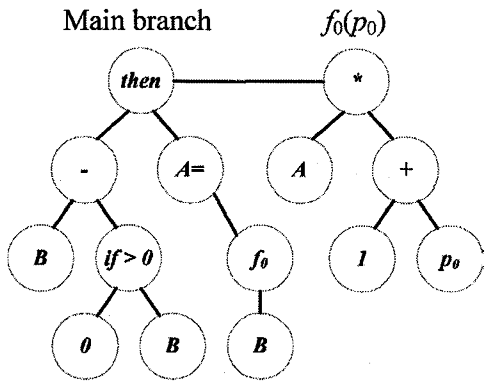
We have ourselves used k-MEANS and ADAPTIVE-k-MEANS clustering algorithms with CLUSTER-BY-FITNESS, although any clustering algorithm would likely be appropriate for this task. After we have clustered the test cases in this way, it may be easier to evolve solutions to the subsets of fitness cases than to evolve solutions to all the test cases at once. This is clearly true for clustering data on the sort-2 problem, since fully half of the test cases can be trivially solved by simply ignoring the given array and returning. This results in many perfect solutions at population initialization, including all the functions of size 1. What is perhaps not obvious is that solving the "hard" population *and* generalizing the solutions is easier than solving the entire problem at once.

Suppose that we begin a new genetic programming run with the task of solving sort-2 where all of the test cases require an inversion. We can name this new problem sort-2-(b, a), since we are sorting all the cases where the larger case occurs first in the array. We use the same setup as before; that is, we leave out the For operator but otherwise perform a full evolution. This simpler problem is optimized using population size $M = 2\ 000$, $G = 10$, where 22 independent runs should be used to solve the problem with 99% probability. This gives a computational effort of 68 000 fitness evaluations to solve the problem 50% of the time, a factor of 9.5 times less than a solution of the complete problem. We can make a profit in computational effort if we can find out how to automatically use the solution of sort-2-(b, a) to induce the solution to sort-2. To perform this task, we will introduce a new automatic subroutine-generation feature to the genetic programming algorithm, the *automatically defined library function* or ADLF for short.

Automatically Defined Library Functions

In his second book on genetic programming [Koza 1994], Koza introduced *automatically defined functions* or ADFs as a way for genetic programming to improve performance on difficult problems. To implement this feature, one or more auxiliary trees are added to each individual in the genetic programming population. These additional trees are viewed as functions that can be called by what is now renamed the "main branch" of the genetic programming forest. The overall layout and behaviour of a genetic programming tree with ADFs is shown in Fig. 18. The main branch of a genetic programming tree with automatically defined functions may call new functions corresponding to each of the ADFs available. For instance, in Fig. 18, one ADF with a

Figure 18



An example of a function with a single automatically defined function with one parameter. The tree on the left is the main branch of the function, where evaluation begins. When the special node f_0 is encountered, execution passes to the first automatically-defined function, on the right. Any subtrees are evaluated and substituted in as formal parameters to the ADF. The return value from the ADF is then returned to be used in computing the main branch.

single parameter is available for the main branch to use. When the main branch encounters an ADF node, any subtrees of the ADF node are evaluated and passed in to the ADF as formal parameters. Execution passes to the appropriate numbered ADF, and a return value is computed. This return value is used as the result of the ADF node in the main branch. In the most usual case, each individual in a genetic programming population has its own set of ADF subtrees, which evolve along with the main branch. We will use a variant of this ADF to make a useful subroutine out of a proven successful solution.

We define an automatically defined library function, or ADLF, in parallel to Koza's ADFs. Evaluation proceeds exactly as in the ADF case. Unlike in the ADF case, however, library functions are stored in a globally accessible archive, and are available for use by all members of an evolving population. The main branch can then use the ADLFs as additional functions, but the ADLFs are not generally evolved along with the main code: they are automatically manufactured from successful individuals by the algorithm MAKE-ADLF. The scientist algorithm can then try a new run with a candidate ADLF and see if it does, in fact, improve performance. If so, the new ADLF can be used in subsequent evolution; if not, the scientist algorithm can give up on the new ADLF and go back to the drawing board. MAKE-ADLF converts a successful solution of a simpler subproblem to an ADLF to be used in future evolutions. MAKE-ADLF uses the candidate parameters A, B, and so on that are chosen and allocated by the related algorithm EVOLVE-ADLF-CANDIDATE. We will describe the operation of EVOLVE-ADLF-CANDIDATE first.

When writing a subroutine, a human programmer has a sense of which information might make good formal parameters. An automatic process has no means of choosing, so we must have the algorithm do something else. One early solution that we tried was to choose formal parameters randomly, by substituting leaf nodes in a correct solution with formal parameters in an ADLF. This worked well enough in specific instances, but was computationally inefficient due to the exponential number of ways that l terminal nodes could be allocated to p formal parameters in an ADLF. After some analysis and testing, we arrived at a better solution. When evolving a function tree that we hope to convert into an ADLF, we augment the terminal set with what we call "candidate formal parameters." We name the candidate formal parameters A, B, and so on. When the correct solution is turned into an ADLF by MAKE-ADLF, these candidate formal parameters become the new ADLF's formal parameters. These candidate formal parameters may be constant, or may covary along with test cases in a way that is described in the next section. For the process of converting the solution of sort-2-(b, a) into a useful ADLF for evolving sort-2, for example, we used constant values of A = 0 and B = 1. In the definition of EVOLVE-ADLF-CANDIDATE, we use the easily enumerable symbols A_1, A_2, and so on to mean the variables A, B, etc. EVOLVE-ADLF-CANDIDATE uses the size-weighted variant of EVOLVE-TINY-TREES using (10) in place of *size*(t) to ensure that future subroutine calls to the ADLF are efficient and use the formal parameters where appropriate. We weight the terminal nodes so that the candidate formal parameters A_i are preferred over regular terminals; that way, the weighted shrinking operation will use them. Specifically, we have had some success using the weights as defined in (13) with EVOLVE-TINY-TREES in EVOLVE-ADLF-CANDIDATE.

$$w_i = \begin{cases} 0.5 & \text{if } i \text{ refers to a member of } \{A_j\} \\ 1 & \text{otherwise} \end{cases} \tag{13}$$

Algorithm 5: EVOLVE-ADLF-CANDIDATE

```
Input:  population size $M$; number of generations $G$; tournament size $T$;
        a vector of $n$ test cases $\mathcal{T} = [t_1, t_2, ..., t_n]$ upon which to make progress; a
        fitness operator that maps performance by a tree $t \in \mathbb{T}$ on a test case $t \in \mathcal{T}$ to a
        fitness value $f : \mathbb{T}, \mathcal{T} \to \mathbb{R}$ for which smaller values are better; $\min f$, which is
        the best possible value of $f$, and a vector-valued function $formals : \mathcal{T} \to \mathbb{R}^d$ that
        assigns each test case a vector of candidate formal values.
Output:  an optimized candidate ADLF hero

    Evolve a successful individual:
    Augment the terminal set of the problem by $d$ additional terminals $\{A_1, A_2, ..., A_d\}$
    In the following evolution, assign $A_i = (formals(t_j))_i \ \forall i = 1...d \forall j = 1...n$ and use
            the weighted variant equation (10) in place of size(t)
        hero ← EVOLVE-TINY-TREES $M, G, T, \sum_{i=1}^{n} f(t, t_i), nf_{\min}$
    end in
    return hero
```

Once we have a candidate library function *hero*, we will make an ADLF from the function using MAKE-ADLF. We can then use the new ADLF to try to evolve a successful individual for the encompassing problem.

Algorithm 6: MAKE-ADLF

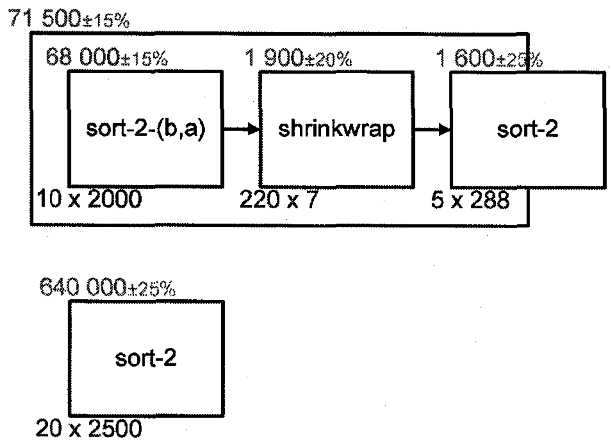
Input: a candidate ADLF function tree *hero*.
Output: an automatically defined library function *adlf*.

```
adlf ← hero
For each node node in adlf do
    If $node \in \{A_1, A_2, ..., A_d\} \wedge node = A_i$ then
        $node \leftarrow p_i$
    end if
end for
return adlf
```

Returning now to the problem of evolving sort-2 from a successful solution to sort-2-(b, a), we performed some tests to determine whether this technique can provide a savings in computational effort. Fig. 19 shows the results of these tests. We can see that performing the evolution in two steps results in a ninefold decrease in net computational effort.

As is typical in evolutionary systems, what we think may be a correct ADLF may not, in fact, work correctly. Ideally, we would like to test an ADLF by using it in a context similar to that in which it will ultimately be used. For the sort-2 problem, no specific after-evolution testing was performed; after shrinking

Figure 19



Approximate computational effort and optimal parameter settings required for each of the three steps of a serial evolution using the ADLF-making functions of this section. CE for a direct evolution of sort-2 is shown as well. The numbers on the top of each box are computational effort and estimated 95% relative confidence intervals; the numbers underneath each box are the optimal main parameters for the evolution, in the form G x M. Performing an evolution in two stages is found in this problem instance to require about 9 times fewer fitness evaluations.

and simplification, the program could be verified by inspection, as it was easily human-readable. This will not be possible in general for automatic systems or for more complex solutions, so another solution will be required.

A good way would be to test the new ADLF by using it as a subroutine on a different problem that is known or suspected to be more challenging than the problem that generated the ADLF. Such a problem might be one that solves a larger number of test cases, as sort-2 is to sort-2-(b, a); or one that is more complex, such as sort-3 is to sort-2. The algorithm TEST-ADLF-CANDIDATE tests a candidate ADLF, and accepts it only if it improves the typical performance of new function trees on a harder problem. Determining how much effort to put into testing a candidate ADLF is a little dicey; we could use a progressive algorithm like that described in Chapter 6, or run the evolution with the new ADLF head-to-head against the same system without the new ADLF. In practice, we have simply chosen "reasonably good" parameters for subsequent evolutions and used TEST-ADLF-CANDIDATE as is.

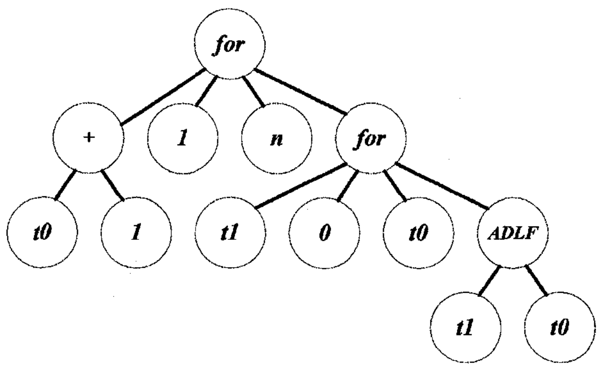
Algorithm 7: TEST-ADLF-CANDIDATE

Input: population size *M*; number of generations *G*; tournament size *T*;
a candidate ADLF *adlf* to test, a fitness operator $f : \mathbb{T} \to \mathbb{R}$ for which smaller values represent better solutions for a problem that is more challenging than the problem on which *adlf* was evolved
Output: *true* if the ADLF is acceptable, false otherwise

```
$sum_{default} \leftarrow sum_{adlf} \leftarrow 0$
for $i \leftarrow 1$ to 3 do
    $sum_{default} \leftarrow sum_{default} + f($ EVOLVE-TREES $M, G, T, f(t), f_{\min})$
    Augment the function set of the problem by adlf
    $sum_{adlf} \leftarrow sum_{adlf} + f($ EVOLVE-TREES $M, G, T, f(t), f_{\min})$
return $sum_{adlf} < sum_{default}$
```

Finally, we would like to evolve a correct solution to the general case. For this ultimate test, we skip directly from size 3 to a much larger size which we take as representative of the general case. We used size 10 as effectively meaning "infinite size". Since the number of possible orderings of *n* elements is *n*!, a successful solution is unlikely to exploit a particular configuration at size 10. This choice is also justified by the fact that generating solutions for sort-3, sort-4, and sort-5 in turn without the benefit of our sort-2-based ADLF becomes exponentially more difficult, with no observed successes in 1000 trials for sort-5. Unsurprisingly, we also observed no successes in 1000 trials for a direct evolution to sort-10. Fitting an exponential regression to the observed computational efforts of the direct evolutions which succeeded gives a 99% computational effort of very roughly $CE \approx 400 \cdot 10^{12}$. This is likely to be an underestimate since the curve appears to be quadratic in *n*, much like Fig. 4 of Chapter 2. Direct evolution of a solution to sort-10 is difficult even when an ADLF from sort-2 is available; however, we able to get reliable successes with a 99% computational effort of roughly $50\,000\,000$ fitness evaluations. We then performed a comparatively inexpensive shrinkwrap operation to get the representative tree shown in Fig. 20.

Figure 20

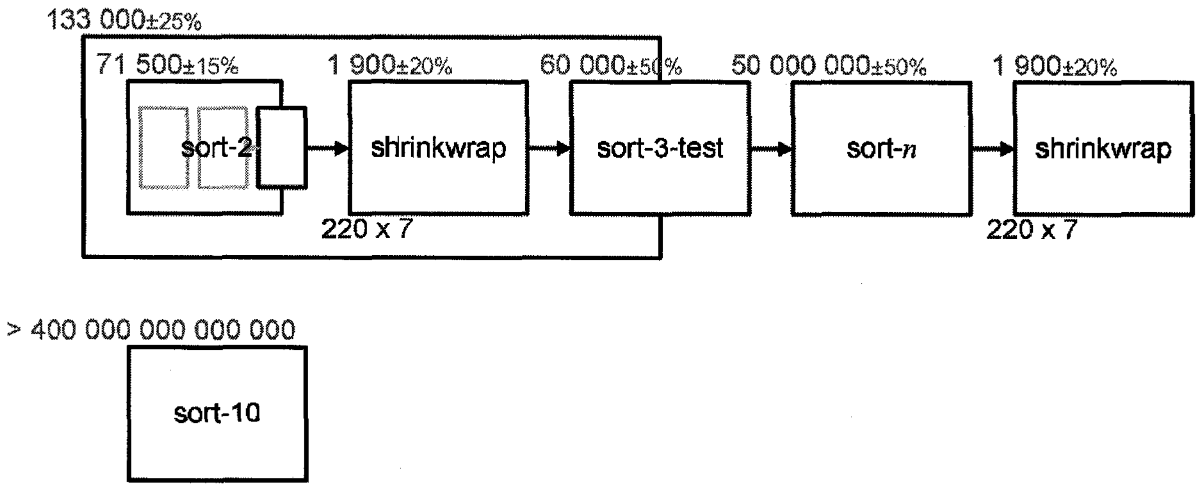


An example of an evolved solution to the sort-10 problem, after shrinking. The leftmost subtree of height 2 is an alias of the optimal subtree *t0*; otherwise this evolved tree is of minimum size for solving the sorting problem.

The overall operation of the scientist algorithm on this problem is sketched in Fig. 21, along with estimated computational efforts for each step. With this last step, we have accomplished an implementation of the initial goal of this thesis – scalable genetic programming.

To demonstrate that this process can be fully automated without any intervention, we implemented the evolution chain of Fig. 21 as a set of perl scripts using Sean Luke's ECJ [Luke 2001] to perform the evolutions. This implementation of the scientist algorithm was then run until a successful sorting algorithm was evolved.

Figure 21



Approximate computational effort and optimal parameter settings required for each of five steps of a serial evolution using the ADLF-making functions of this section, demonstrating scalable genetic programming. The numbers on the top of each box are computational effort and estimated 95% relative confidence intervals; the numbers underneath each box are the optimal main parameters for the evolution, in the form *G* x *M*. The computational effort for sort-10 is estimated based on logarithmic regression from observed success rates on smaller problems, and is likely conservative.

Chapter 8: Conclusion

Thesis Synthesis

One of the overarching goals of artificial intelligence is to duplicate human-level performance. The Turing test is by far the most famous operationalization of this idea [Turing 1950]. We have contributed to work on an empirical problem domain-by-problem domain categorization of the Turing test, called the Turing ratio [Masum 2002]. In the context of scalable genetic programming, the most salient issue from this paper is that of intelligence amplification. Intelligence amplification considers how the algorithm achieves its performance level to measure the autonomy of an AI technique. At one extreme, a program has a full list of answers given to it; at the other extreme it starts with zero domain knowledge and succeeds in learning good solutions. In between, there are various degrees of "cooking": exhaustive enumeration of cases, hand-designed primitive functions, intermediate reinforcement mechanisms, externally supplied fitness functions, and so forth. This distinction is analogous to that between rote and deep learning: in the former, all knowledge is given by the teacher; while in the latter, the teacher suggests insights and poses problems but the student does most of the problem-solving. In terms of artificial intelligence, a contrast could be made between the checkers program Chinook [Schaeffer 1996], which uses opening book, end-game and pure horsepower to play checkers; and Blondie 24 [Fogel 2002], which evolved a neural-net to evaluate boards by playing humans online.

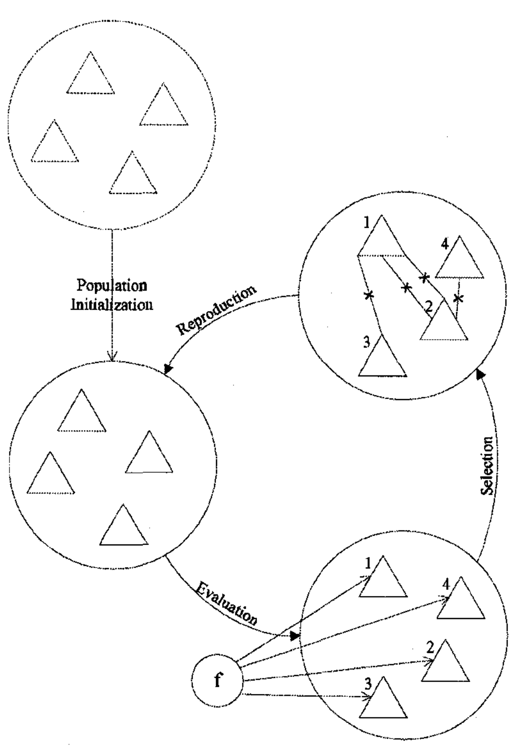
We can draw an analogy between teaching style and algorithm design. A general-purpose algorithm separates a general problem into a set of base cases and a set of inductive cases, solves each separately, and derives a correct and hopefully efficient solution for any input. This can be distinguished from a lookup-table approach, where solutions for simple cases are precomputed and enumerated. Both of these techniques are combined in practical problem solving, with the more general approach being preferred, but more difficult and expensive to achieve.

Genetic programming has traditionally been located near the autonomous end of the intelligence amplification spectrum, in that a relatively simple algorithm generates innovation. The problem-specific smarts is encoded largely through the fitness function; and indirectly through the set of functions and terminals available to the evolutionary system. We seek herein to extend this intelligence amplification from solving a particular well-defined problem to being able to solve a problem with a scalable difficulty parameter. In this thesis, we push the frontier of automatic problem solving a few small steps towards human performance. We did so at the cost of adding some complexity in terms of expanding the genetic programming problem-specification and -validation paradigm, as well as using genetic programming as a subroutine. However, as these additional tasks require only constant additional effort, we see that the intelligence amplification of scalable genetic programming remains high.

The scientist algorithm presented in Chapter 2 and elaborated on in Chapters 5, 6, and 7 represents an attempt to encode some of the smarts of human algorithm design as a programming toolkit. The overall idea is to use genetic programming as an innovation subroutine while making progress in a reliable way towards an ultimate goal. Fig. 1 shows the overall operation of the genetic programming algorithm.

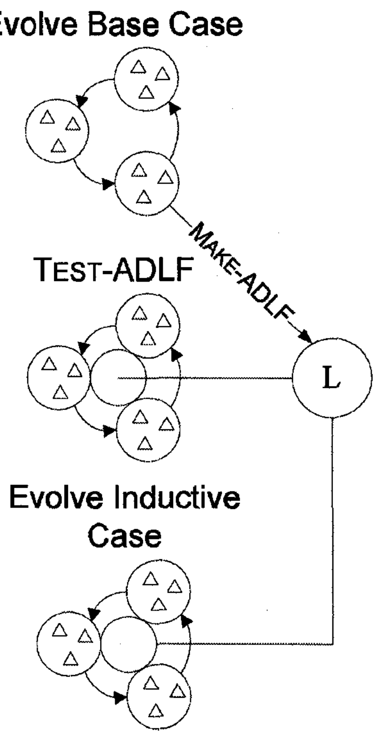
The SCIENTIST-ALGORITHM of Chapter 2 is more like a scaffold than a complete algorithmic description of the scientist framework. We can get a better sense of how the scientist algorithm works by considering the sequence of steps performed at the end of Chapter 7. Fig. 2 illustrates this discovery process which led to a demonstration of scalable genetic programming.

Figure 1



Canonical operation of the genetic programming algorithm. Function trees are represented by triangles in this diagram. The circle labelled "f" represents the fitness function. Fitness values are represented by their ranks.

Figure 2



Operation of the scientist algorithm in Chapter 7. The canonical GP algorithm of Fig. 1 has been condensed to three circles feeding into one another. The circle labelled "L" represents the library of automatically defined library functions available for use in further evaluations. This clearly shows the use of genetic programming as a subroutine in the scientist algorithm.

The labels of Fig. 2 clearly indicate the analogy to algorithm design. The familiar divide-and-conquer technique of algorithm design begins with one or a few base cases and moves to an inductive case. We implemented an analogous process for genetic programming, to make successful progress on a previously insoluble problem. While this straightforward approach will work for some problems, it will not work for them all. Other problems may require more trial-and-error than this; for instance, the careful characterization of optimal values for *M* and *G* of CHOOSE-POPULATION-SIZE-AND-GENERATION-NUMBER of Chapter 6 may prove useful in finding good parameter settings. Certainly the statistical approach embodied in UPDATE-GENERATION-LIMITS, also from Chapter 6, is a reliable way to determine whether a given experimental intervention is productive or not. This algorithm draws on the hard-won insights of Chapters 3 and 4 as to how best to perform hypothesis testing for evolutionary computation. This careful statistical measurement can be applied as well to scientist interventions, as a means of validating the work performed by GP.

In this thesis, we felt it was more important to introduce and prove productive ideas than to exhaustively test them. To make concrete progress on scalable genetic programming, we hoped to illustrate the problem and highlight the major innovations that will be required to achieve this goal reliably and automatically. By advancing along the variability levels of Fig. 1 of Chapter 2, we can move towards human performance in algorithm generation, and hence improve automatic problem solving.

Contributions to the Field

While the work performed on scalable genetic programming is the most exciting part of this dissertation, it is probably the contributions towards evaluating and comparing stochastic search algorithms that will be the most useful. While we suspect that the overwhelming majority of published conclusions about evolutionary computation performance are correct, it is essential to have and use reliable yardsticks if serious progress is to be made. Chapters 3 and 4 go a long way towards achieving this end. It is our hope that these results will not only gain acceptance in the evolutionary computation community, but be more widely accepted in the broader field of artificial intelligence. Indeed, the experimental parameter-setting approach embodied in CHOOSE-POPULATION-SIZE-AND-GENERATION-NUMBER is already a highly useful subroutine for the common problem of setting parameters in stochastic search systems.

Another significant contribute of this thesis is in using genetic programming to automatically discover parsimonious solutions using EVOLVE-TINY-TREES. Parsimonious programs are required to obtain one of the advantages of genetic programming over competing techniques, in that human-readable and -comprehensible solutions can be achieved automatically. Much has been written about code bloat in genetic programming, culminating in Terrance Soule's Ph.D. thesis in [Soule 1998]. While mitigating bloat during evolution for the purposes of for computational efficiency remains valuable, EVOLVE-TINY-TREES enables optimal and near-optimal code size for success-based problems. The related algorithm EVOLVE-TREES-PREFERRED-FITNESS is useful in other contexts as well; for instance, we have used it to reduce the number of time steps required for the artificial ant on the Santa Fe trail. We suspect that many auxiliary variables can be satisfied in this way, through serial evolving.

Adopting a top-level algorithm that conducts experiments to make progress will be useful in many more contexts than the algorithms and proofs-of-concept provided herein. Consider the algorithm sketch of Fig. 2. Genetic programming is a fantastic source of innovation and problem-solving ability. Great gains can be achieved by using genetic programming as a subroutine. We have demonstrated that useful subroutine discovery can be automatically and reliably achieved by some naïve algorithms combined with some simple experimentation. The systematic experimentation is the key component that allows clear and unambiguous progress to be made when validating subroutines. While there is a strong case to be made that genetic programming continuously experiments to make progress, the ability to identify useful subroutines, prove them, and protect them from the destructive manipulations of genetic programming is new. The automatic gathering and use of statistical information to make informed decisions about which avenues and manipulations are productive has wider applicability than CHOOSE-POPULATION-SIZE-AND-GENERATION-NUMBER. Incremental evolution and statistical analysis adds a kind of ratcheting mechanism to the relative chaos of genetic programming, leading it progressively upwards toward ultimate problem solution. This, in a way, combines "good old-fashioned artificial intelligence" with the robustness of evolutionary techniques. This kind of synthetic approach is a valuable contribution of this thesis.

We would be remiss if we did not draw attention to one of the major surprises of this work – namely that breaking problems down to the simplest non-trivial components can have major benefits in terms of problem-solving ability. The experiments on sorting of Chapter 8 demonstrate that even solving the nearly trivial problem of sorting a reversed array of size 2 has an order-of-magnitude benefit on solving the sorting problem of size 2. The exciting thing about this observation is that we can automatically determine these easy cases through an algorithm like CLUSTER-BY-FITNESS of Chapter 8. The combination of some experimentation, some automatic analysis through clustering, and some subroutine induction is a powerful technique that will have wider utility than just the sorting example introduced here.

Genetic programming has achieved many impressive successes, as demonstrated by Fig. 1 of Chapter 1. However, most of these sucesses are outside of the nominal target domain of programming. Increasing the generalizability and power of genetic programming on traditional programming tasks is a significant advance, even if only early steps have been demonstrated here.

Recommendations and Advice

By way of summary, we would like to highlight some of the recommendations and advice for practitioners uncovered through the experiments performed in this thesis. Our first concrete advice comes from the work of Chapters 3 and 4: use effective success probability or computational effort to compare *EC techniques* where success is an achievable outcome. Use these same measures to compare the *difficulty* of problems for automatic programming. For problems where continuous improvement is the goal, effective mean best fitness and the *y*-test are the best techniques to use, depending on whether one wants to do parametric or non-parametric testing. For all of these methods, biases in the underlying statistics require a significant number of experiments: hundreds of experiments rather than dozens. All these measures, but especially computational effort estimates, should always be presented the number of runs performed. The best generation results from Chapter 4 are salient as well, suggesting that we should also report the number of competing treatments investigated. This last value can be readily estimated by comparing confidence intervals; any treatment that may be the best treatment should be added to the count of competing treatments.

Experimentation on genetic programming has shown a surprising result: evolutionary computation problems are often close to GP-hard. For difficult problems, we would be well-advised to try evolving solutions with small trees, since the density of solutions is highest there. The work of Chapter 5 suggests that systematically considering all small trees and making subroutines from these small trees can be a productive approach, even when a problem is nearly GP-hard. The modelling work and experiments of Chapter 6 suggest that with small population sizes, we need not be too precise in the number of generations performed. For large populations, however, we can waste a great deal of work if we do not choose the number of generations carefully. The results there point to a fairly broad minimum in the computational effort, which suggests that choosing population sizes to within a factor of two or three is probably sufficient for many problems.

Future Work

This dissertation opens many doors. This has the necessary consequence that much experimentation and validation remains to be done. The statistical work of Chapter 3 reveals that problems can be empirically categorized into two different kinds: success-based and improvement-based problems. These kinds of problems require different comparison methodologies. Computational effort is an excellent tool to use for success-based problems; however, Chapter 4 shows that there are some pitfalls to using this technique to compare algorithms. A version of the computational effort algorithm that corrects for the biases of Chapter 4 is still lacking, although the work presented here should enable a robust bias-estimation procedure.

Early tests on CHOOSE-POPULATION-SIZE-AND-GENERATION-NUMBER seem promising; however, we would like to test this algorithm on a wider set of test problems. It would seem to be a worthwhile effort to simply rerun CHOOSE-POPULATION-SIZE-AND-GENERATION-NUMBER on the set of test problems introduced in [Koza 1992] and [Koza 1994]. Reliable numbers for computational effort, best population size and best generation number would provide a solid foundation for informing parameter choice on other problems. Another advantage of a broad investigation of performance would be in providing data to inform candidate GP theories of success.

The modelling approach of Chapter 6 seems to us very promising. After considering enough problems, it seems likely that a semi-empirical model of EC performance could be fashioned. The data of Koza's Genetic Programming series [Koza 1992, Koza 1994, Koza 1999, Koza 2004] would be an excellent starting place for this effort. Regenerating the qualitative model of Chapter 6 for each instance of a broad problem suite would go a long way towards quantifying the benefits of different genetic programming interventions such as automatically defined functions. It is possible that a parameterized model of success probability could result from such an effort; this would allow a refinement of CHOOSE-POPULATION-SIZE-AND-GENERATION-NUMBER that needs only to estimate a few model parameters. This would be far more efficient than the current algorithm, which must compete hundreds or thousands of treatments against one another to ascertain best parameter settings.

The four models that we cycled among in FIND-INITIAL-M-G represent more a proof-of-concept than serious work. Using additional best-parameter data from a broader set of problems would enable more efficient behaviour in this algorithm. While it is perhaps obvious, it is worth mentioning that an informed selection of good parameter settings will have a direct payoff in the number and kind of human-competitive problems that GP can solve. This is equally true for the subroutine-manufacturing code of SYSTEMATIC-SUBROUTINE-GENERALIZATION and MAKE-ADLF. These two subroutine-generating algorithms are proved on specific problems; testing them over a broader problem suite would be a significant advance.

On the topic of subroutine-generating functions, testing the scientist algorithm further on other scalable problems is an obvious next step. Fortunately, most of the problems in [Koza 1994] are virtually designed for scalability, so coming up with test problems should not pose a major obstacle. Many problems in first- and second-year computer science courses are also scalable problems that should be amenable to the scalable genetic programming technique. Inducing subroutines on a broader set of problems should prove very interesting, both in terms of the successes achieved and the identity of the subroutines generated. One exciting possibility bears special consideration. We might present a slightly beefed-up version of the scientist algorithm with a portfolio of problems to solve. If these problems all share and can contribute to a common automatically defined function library, we may find the equivalent of human-designed programming library functions being evolved. For instance, sorting is a useful library function for a broad set of algorithms; providing a sorting operator as an evolved primitive operation will likely enable the efficient solution of some sorts of otherwise-insoluble problems. Serious progress along these lines will require the adaptation of evolved data structures to the genetic programming paradigm as well. Some work has already been done along these lines, particularly in [Langdon 1998b]. A companion effort to the present work on scalable genetic programming that elucidates scalable data structures would be a very productive endeavour. With evolved data structures, automated and scalable programs, subroutine generation and a co-evolved validation mechanism, we can imagine a very powerful automated problem-solving system could be fashioned. Using the scientific method, embodied through experimentation and hypothesis testing, will be key to the success of such a programming system.

This page is intentionally left blank.

Appendix 1: How Does Genetic Programming Work?

In this appendix, we will illustrate the conventional problem that genetic programming is expected to solve in human terms, so that the reader can appreciate how difficult the task is. We will make use of an thought experiment similar to Searle's "Chinese Room" allegory [Searle 1980, Searle 1984] to clearly illustrate the problem. This appendix is written as a narrative, unlike the remainder of this thesis.

Genetic Programming Room Allegory

Suppose that you are ushered into a closed room with two tables. One of the tables has a number of piles of different tokens, each with a number inscribed on it. The other table is empty. Each token looks something like a tinker toy disc: it is flat, round, has a number inscribed on one side of it, and has one or more short stems of an elastic material sticking out from the perimeter with a little plastic knob at the end. Some of the tokens have no knobs. There are several different kinds of token and many copies of each kind of token on the table. After some investigation, it seems that there are seven kinds of token in all. Picking up one of the tokens, you discover that it has a slot at the top; the slot looks like the knobs could probably fit into it. A moment's notice shows that each token has the same kind of slot at the top. Since the stems are flexible, you can probably recombine any tokens with any other tokens easily - they fit snugly once plugged together. Since there is ony one slot per token, you are constrained to produce trees with the token set. The stems are flexible enough that you can avoid geometrical problems with the tokens intersecting one another.

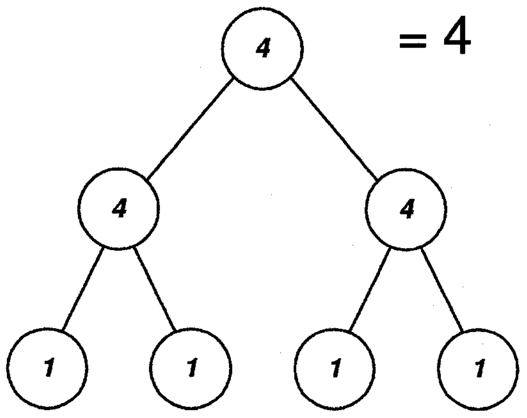
The room has a rotating change door along one wall, like those at the ticket counter at a movie theatre. The change door has a large flat surface that rotates around. There is a button labelled "submit" beside the change door. You figure that a completed tree of tokens will fit nicely into it. There is a numeric readout on the same wall as the change door, which looks like an old-style plasma number display. It looks like the display can also spell the words "Error" and "Success" - you can see the letters lightly traced over the presently darkened display. A poster on the wall to the left of the number display has the text "Score:" written on it, like a label.

There is a second much larger display, maybe the size of a blackboard, on the wall beside the first display. It is currently black. You aren't sure exactly what this one does. Having nothing else to do, you examine the tokens more closely. The tokens labelled 0, 1 and 2 have only a slot at the top of the token, and no stems coming out of them. The other four tokens, labelled 3, 4, 5, and 6, have two stems each at the bottom, and a slot at the top. All of a sudden, a person comes in the door. She says, “you can't go for lunch until the word "Success" lights up on the display.” "Oh, yeah, one more thing: the best that you can score is eight," and she closes the door. You hear a distinct locking sound.

Since you have some time to kill, you try putting some tokens together. They stick together nicely. You walk over to the change door and put the tree in the compartment. You rotate the change door, and push the button. Almost instantly, the word "Error" appears on the score display. The big blackboard display draws a cartoon picture of your tree, with the word "Error" beside it. The change door rotates on its own, and gives you your tree back.

Appendix 1: How Does Genetic Programming Work?

Figure 1



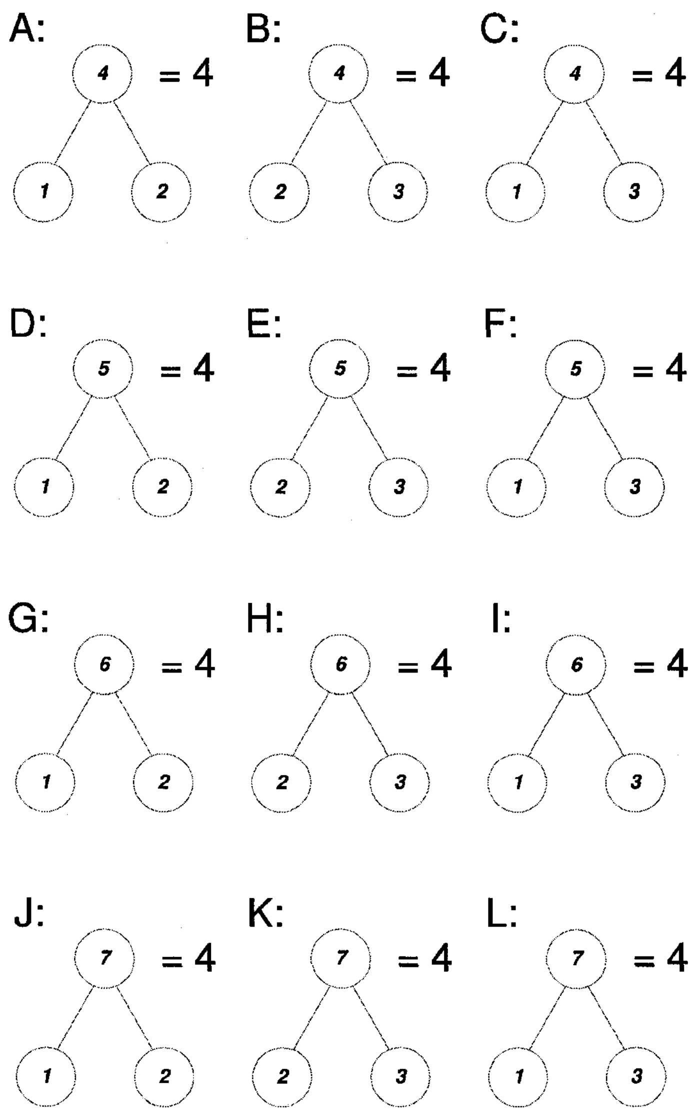
The first successful tree attempt for the GP room.

Your second and third attempts give "Error" as well, and you start to lose faith that *anything* will work. It looks like it could be a long time until lunch. For your fourth attempt, you try plugging in 1-tokens along the bottom edge of your tree, giving the picture in Fig. 1. Once again, the change door rotates, and you expect another "Error".

However, this time you are rewarded for your persistence, since the display lights up "4". A little trial and error later, you figure out that you always need to terminate the tree by making sure all the slots at the bottom end of the tree are attached to 0, 1, or 2 tokens, or else you will get an error. As you submit each successive tree, the big display updates to keep all the trees you've been working on and the score they achieved in view.

After a few more random trials, you decide that perhaps a more systematic approach might be in order. You decide to try out all the three-node trees. After a little experimentation, you discover that there are twelve of them in total. They give the scores shown in Fig. 2, shown on the next page. If there is one thing to notice, it is that the results are consistent: all of our attempts gave scores of four! You begin to wonder if the machine might be broken, and visions of an quick exit from the room begins to recede. You are about to call for a technician, when it occurs to you - maybe you need three layers of nodes to make a successful tree. Back to the drawing board. Continuing with the systematic theme, you decide to try all combinations that can be made with two "4" tokens. If these all return a score of four points as well, you're going to give up and try to rouse somebody.

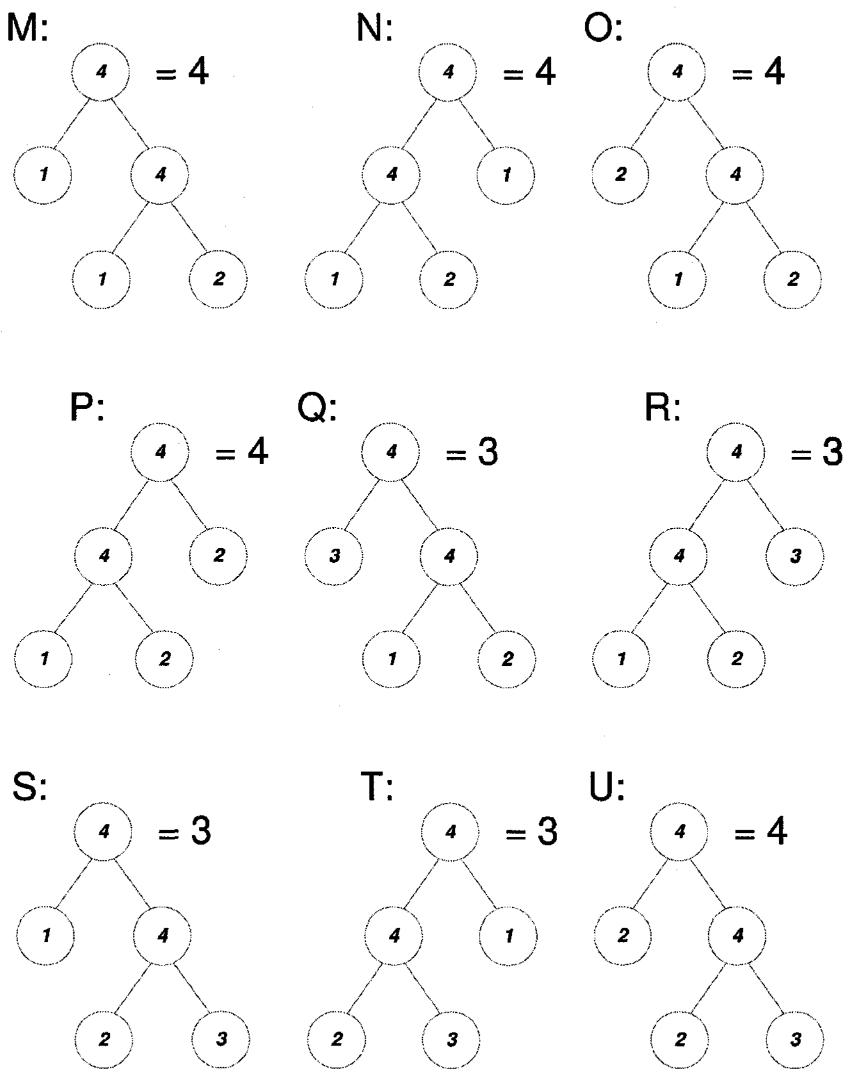
Figure 2



All possible 3-node trees for the tree creation problem, and their scores.

The first nine such trees are shown in Fig. 3, and the second set of nine in Fig. 4.

Figure 3

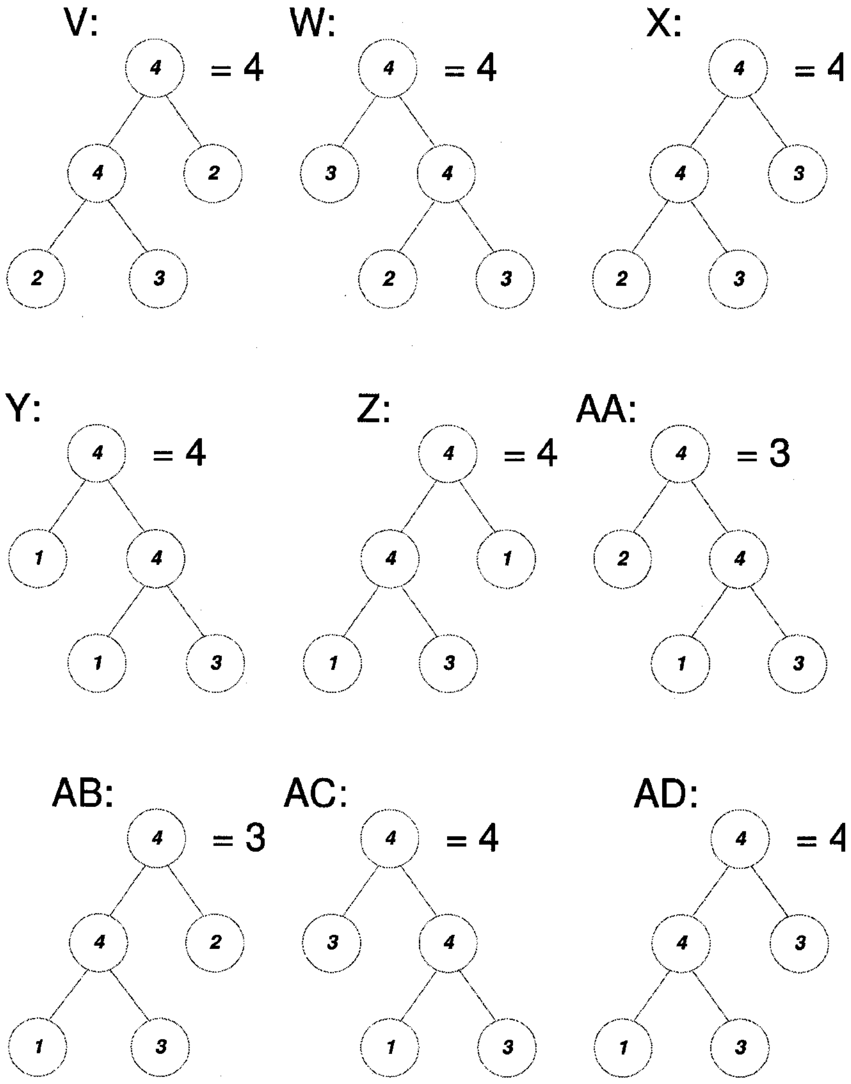


First set of 9 combinations achievable using two "4" tokens.

The good news is that we've managed to get a score other than four. The bad news is that our new high scores is still four, as all the new scores are worse than four! Another thing that you notice is that when the leaf nodes hold a "1, 2, 3" combination in any order, the score is three. Interesting.

Appendix 1: How Does Genetic Programming Work?

Figure 4

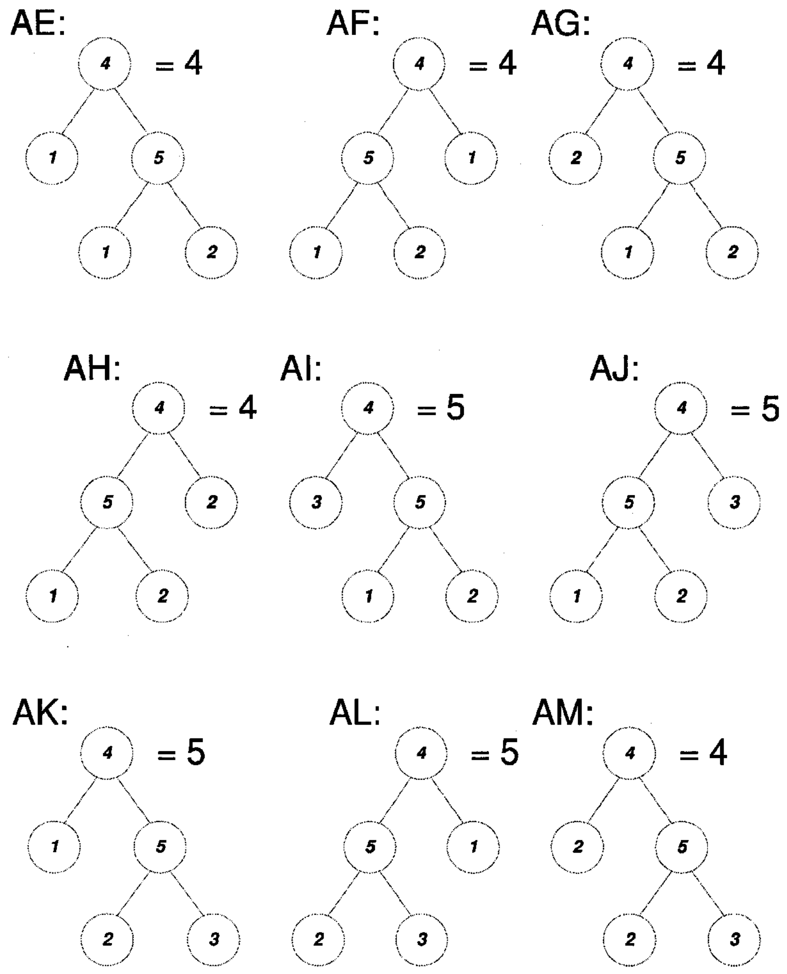


Second set of 9 combinations achievable using two "4" tokens.

Well, if a pair of 4 nodes at the top doesn't work, perhaps a 4 and a 5 will. You resolve to systematically try all of these trees. The results are heartening - your first tree with the "1, 2, 3" configuration, labelled "AI" in Fig. 5, scores you a whopping five points! Maybe there's hope of finishing this off in time for lunch after all.

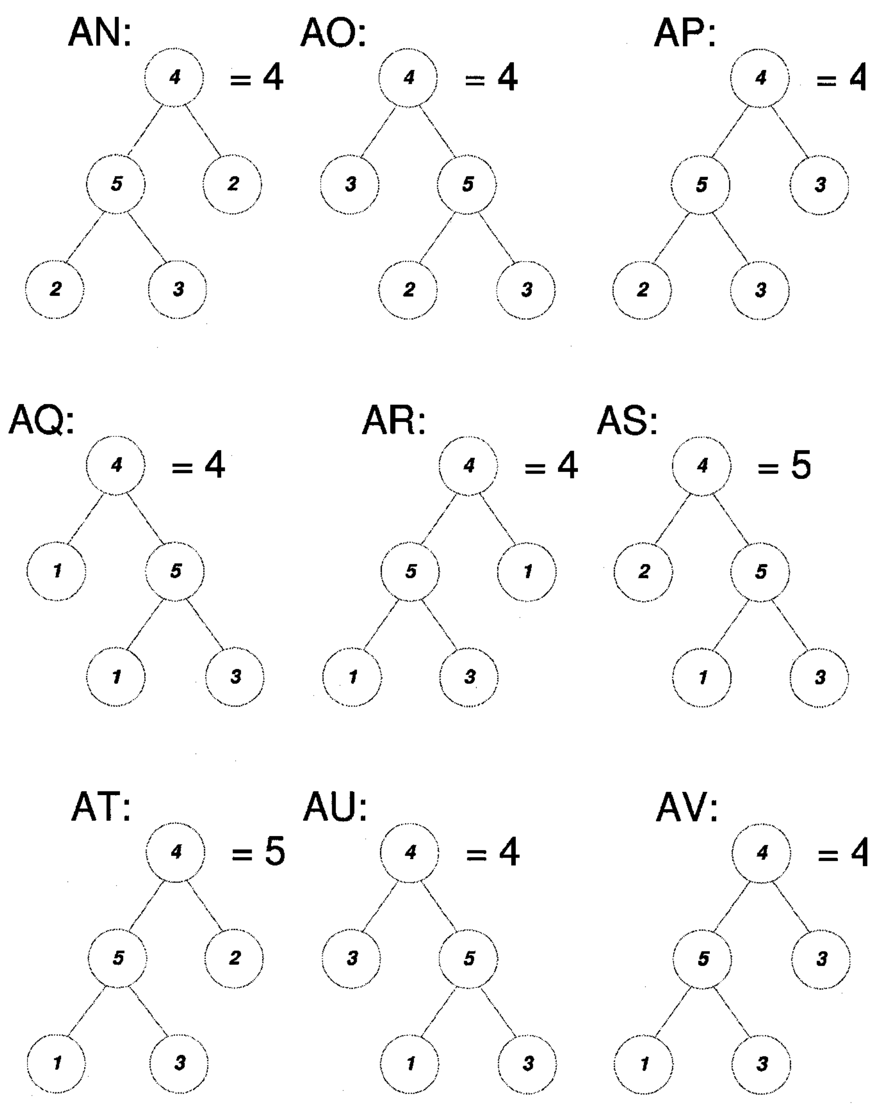
There are eighteen such trees in all; the other nine are shown in Fig. 6, offering few surprises. It appears that any permutation of "1, 2, 3" in the leaf nodes gives us a superior score, while variants with the doublets "1, 1, 2" and the like give a score of only four.

Figure 5



First 9 combinations achievable using a "4" and a "5" token at the tree root.

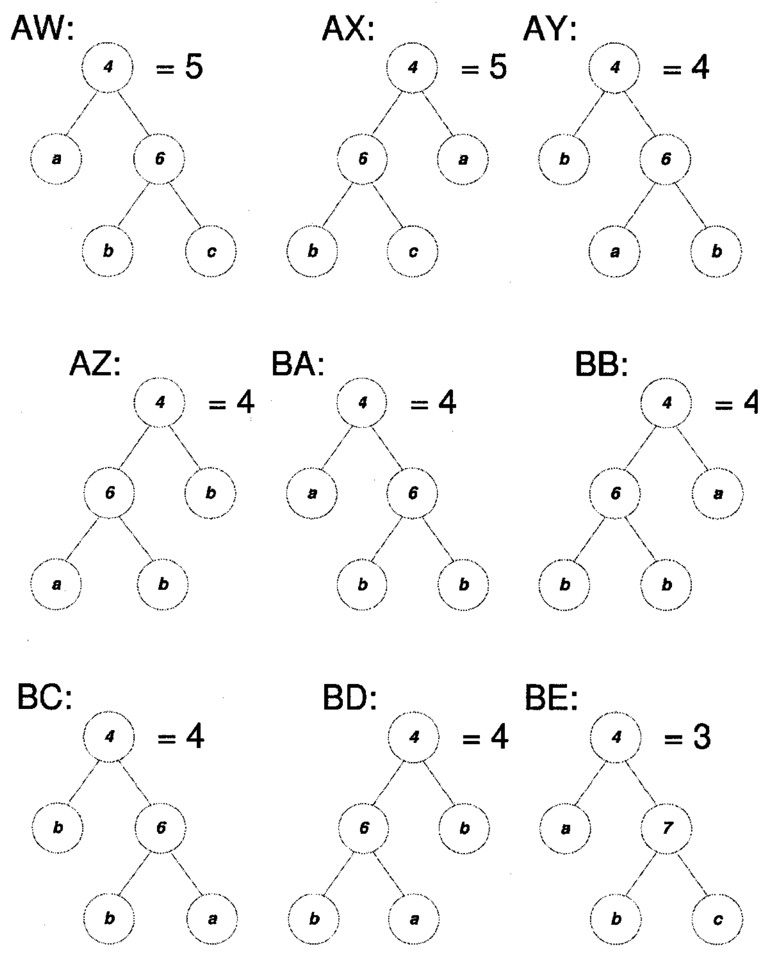
Figure 6



Second 9 combinations achievable using a "4" and a "5" token at the tree root

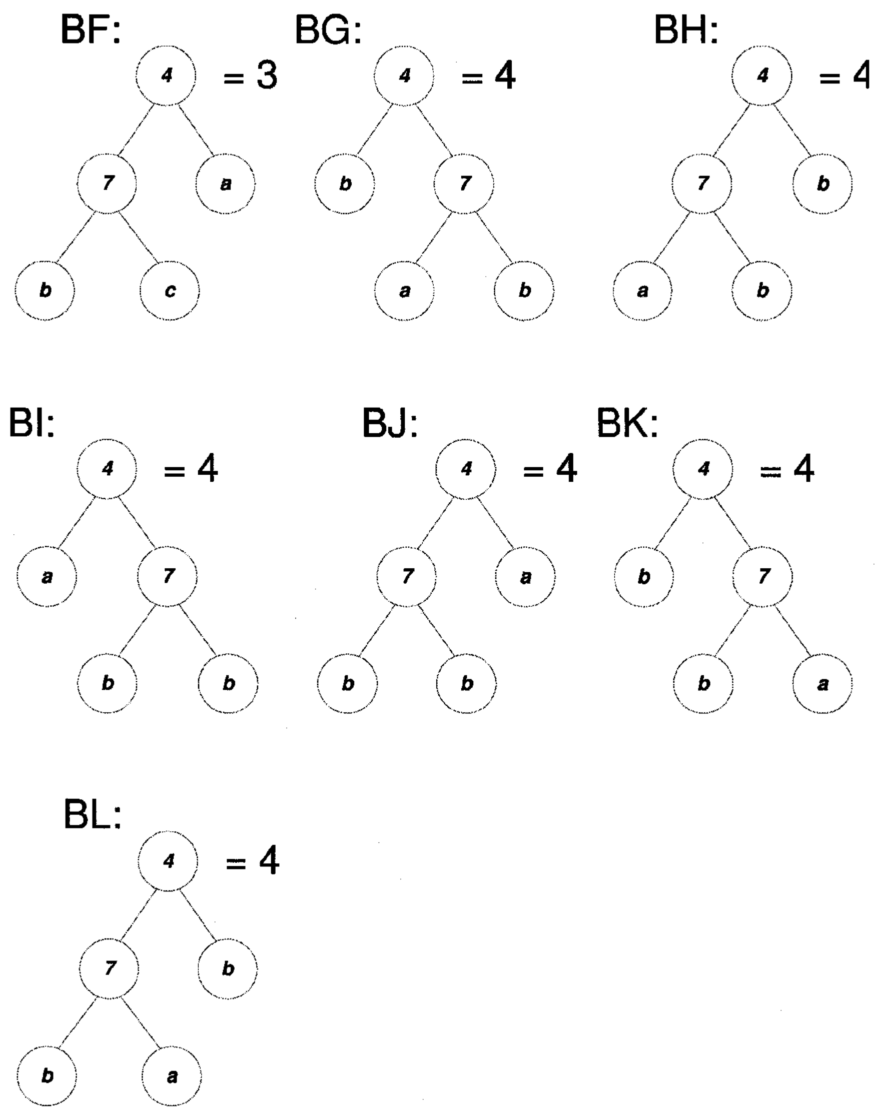
Continuing in the systematic vein, you try some trees involving the binary nodes 4 and 6. All of the various configurations are summarized in Fig. 7. Of note here is that 4 and 6 at the root of the tree seem to behave like 4 and 5 do - in fact, there appears to be no significant difference between the two. A quick check of 4 and 7 just to be sure is in order. Wait a minute! The first tree you try, tree "BE", seems to contradict the trend. It gives a score of only three, just like the 4-4 trees did! Carefully checking your work, you try out all the other configurations, as shown in Fig. 8. Sure enough, all the other configurations return a score of either four in the "*a*, *a*, *b*" case or three in the "*a*, *b*, *c*" case.

Figure 7



Combinations achievable using a "4" and a "6" token. The symbols ***a***, ***b***, and ***c*** may be assigned to the tokens "1", "2" and "3" as differing variables, giving the indicated scores. One example using a "4" and a "7" token is indicated (BE).

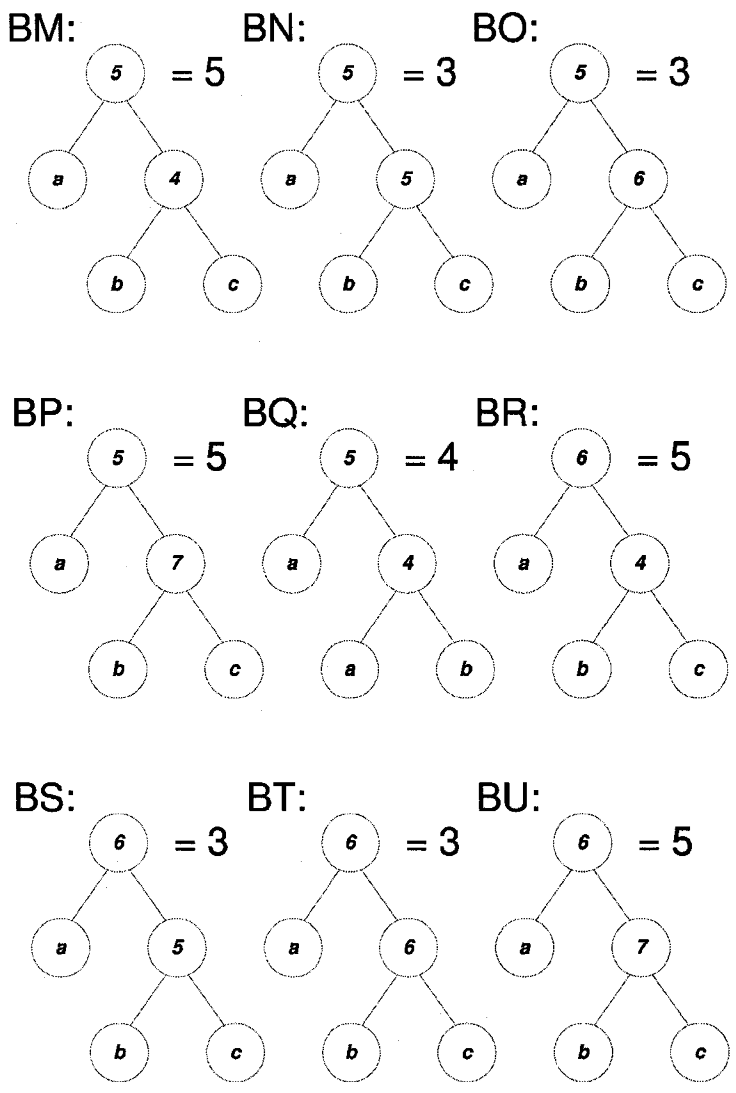
Figure 8



Combinations achievable using a "4" and a "7" token. The symbols ***a***, ***b***, and ***c*** may be assigned to the tokens "1", "2" and "3" as differing variables, giving the indicated scores.

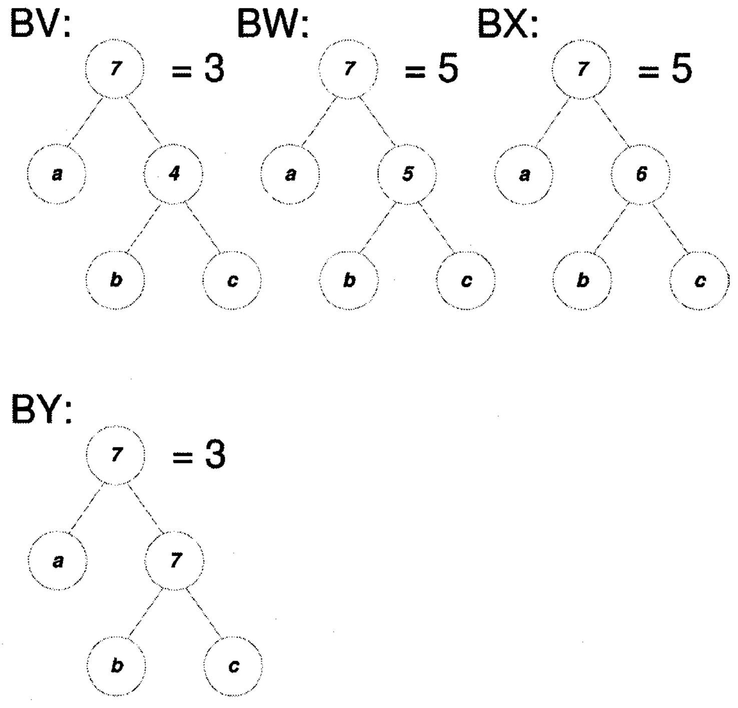
Having tried all the combinations with a 4 in the topmost position, you figure that perhaps it would be useful to try 5, 6, and 7 there. In the interests of time, perhaps it might make sense to only try a representative element from each of the $4^2 - 4$ remaining combinations. You resolve to try the "1, 2, 3" configuration for each pair of top nodes, and you get the results shown in Figs. 9 and 10.

Figure 9



Combinations achievable using a "5" or a "6" token at the root. The symbols ***a***, ***b***, and ***c*** may be assigned to the tokens "1", "2" and "3" as differing variables, giving the indicated scores.

Figure 10

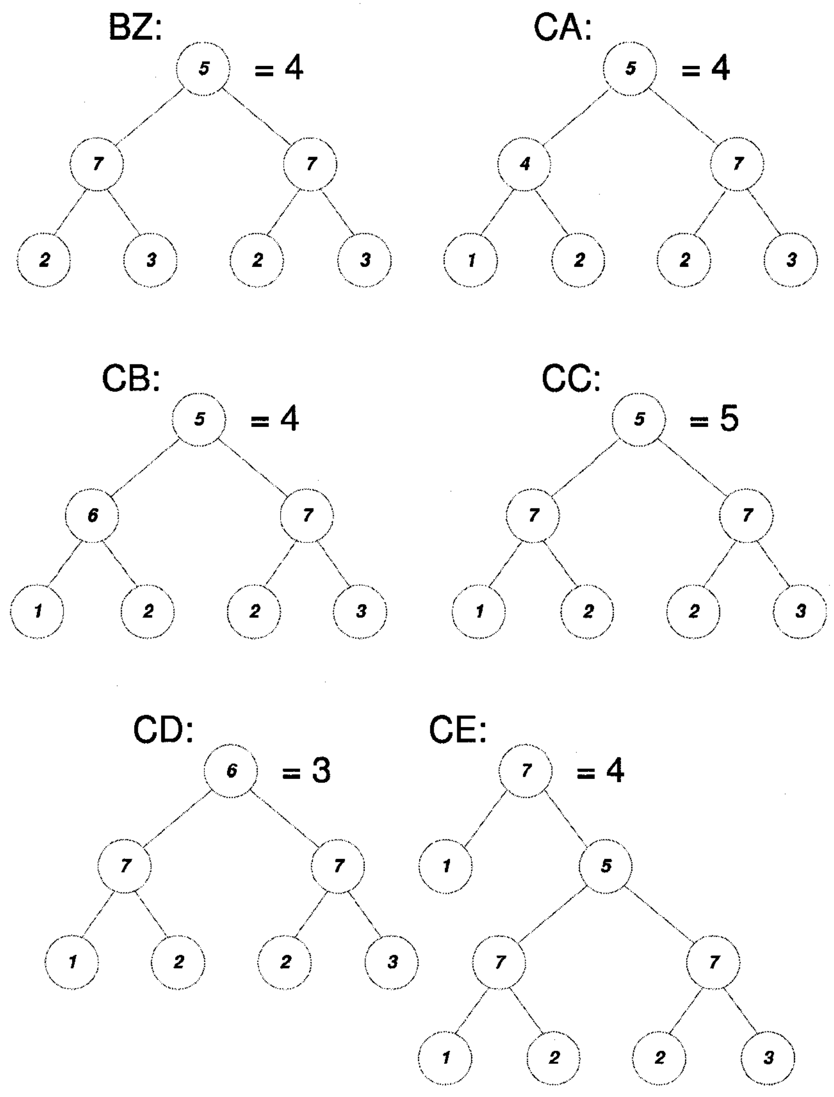


Combinations achievable using a "7" token at the root. The symbols ***a***, ***b***, and ***c*** may be assigned to the tokens "1", "2" and "3" as differing variables, giving the indicated scores.

It would appear that some of the two-node combinations have an affinity for one another. For instance, the 5-4, 5-7, 6-4, 6-7, 7-5 and 7-6 pairs all produce scores of five, while the opposite pairs give scores of three. Adding in the results of the previously encountered 4-5 and 4-6 pairs suggests that these pairs seem to reinforce one another.

What to try next? Obviously, we'll need to try a larger tree, one with 4 layers, perhaps. A quick calculation shows that our systematic approach will become quite arduous at this point, since there are $4^3 = 64$ ways to lay out just the 3 topmost nodes! Let's try using 5 as the topmost node, 7 as the right internode, and variously 4, 5, 6 and 7 as the left internodes. A few minutes later, the trees labelled "BZ" through "CC" appear on the big display, with scores as shown in Fig. 11. This gives us a score of four, which is unsurprising, given that we have only used the terminal nodes 2 and 3 here.

Figure 11



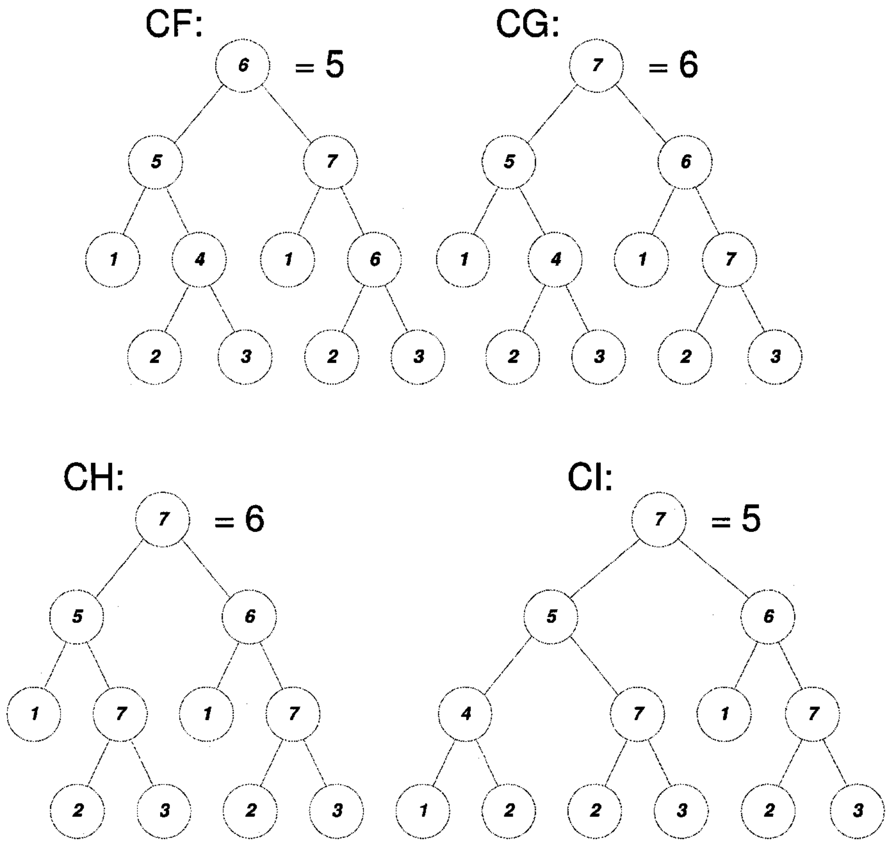
Some balanced full trees of height 3 and their respective scores. The tree labelled CE is the same as CC, but grafted as a right child onto a small tree of height 2.

Perhaps a variant using 4, 6, or 7 for the bottom-leftmost node with the leaf nodes "1, 2, 2, 3" might be in order. They also give scores of four, except for the variant using

7, which has a score of five. At this point, maybe we should revisit the decision of using a 5 as the root node; swapping in a 6 node gives us the tree labelled "CD", which has the disappointingly poor score of three. Perhaps moving to a larger tree might be in order? Starting at tree "CC", we extend it by one node in the upwards direction, as shown in tree "CE". This gives us the default score of four, so let's try something else.

Perhaps a better strategy would be to duplicate the whole subtree – so that we have a single root, and then two geometrically similar subtrees lying underneath that. A first attempt along this line gives us the relatively successful trials shown as trees "CF" through "CH" in Fig. 12. Now we're really getting somewhere – notice that we have found two different ways of achieving the impressive score of six! Perhaps if we replace the lone 1 nodes hanging off the leftmost children of

Figure 12

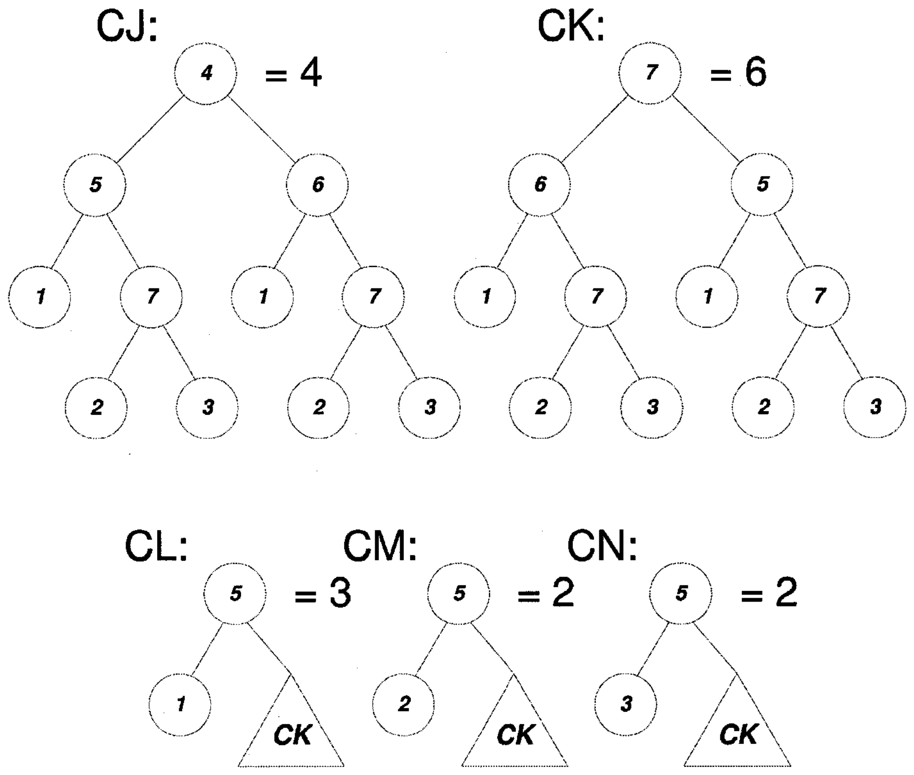


Some combinations of achievable trees with height 4. These are chosen so as to have geometrically similar subtrees under the root, in this case "4, 2, 3", "6, 2, 3", and "7, 2, 3".

the subtrees under the root with full subtrees, we might be able to continue on and score even higher than our new best of six! Our first attempt along these lines, tree "CI" of Fig. 12, isn't that impressive - a single point degradation from our new best trees. Going back a step, we can rotate in other successful combinations of nodes, as in trees "CJ" and "CK" of Fig. 13.

Maybe increased height is the answer again! We can designate the successful tree "CK" as a subtree, and then try hanging tree "CK" off various subtrees. Three quick attempts, shown by trees "CL" through "CN" in Fig. 13, give positively awful results, scoring three, two, and two respectively.

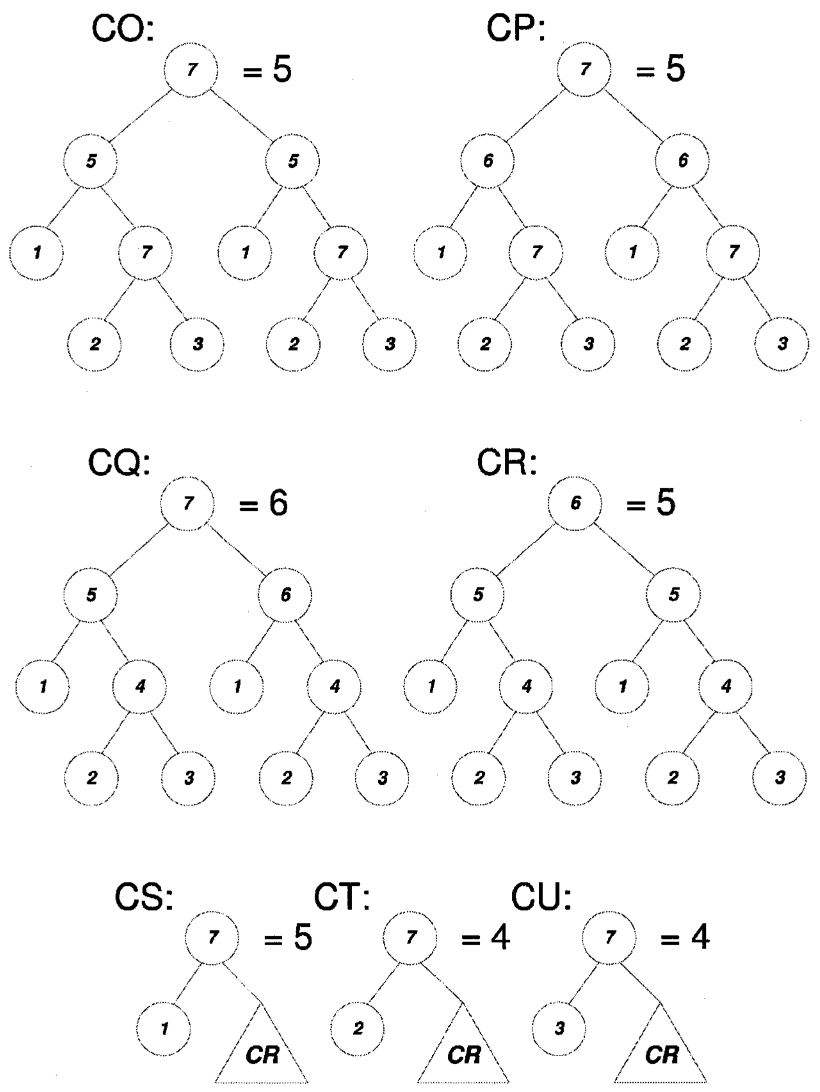
Figure 13



The trees "CJ" and "CK" are combinations achievable of height four with geometrically similar subtrees under the root. Trees "CL", "CM", and "CN" use the symbol "CK" to indicate the entire tree labelled "CK", which is attached as a subtree under the root node.

Let's go back and try these same tricks again, but using matching nodes under the root this time. We get the trees "CO" through "CU", as shown in Fig. 14. These duplicate but do not exceed prior successes. It does feel nice to have several sixes available to us. Getting frustrated, and hungry, we try a single larger tree with nice symmetry properties.

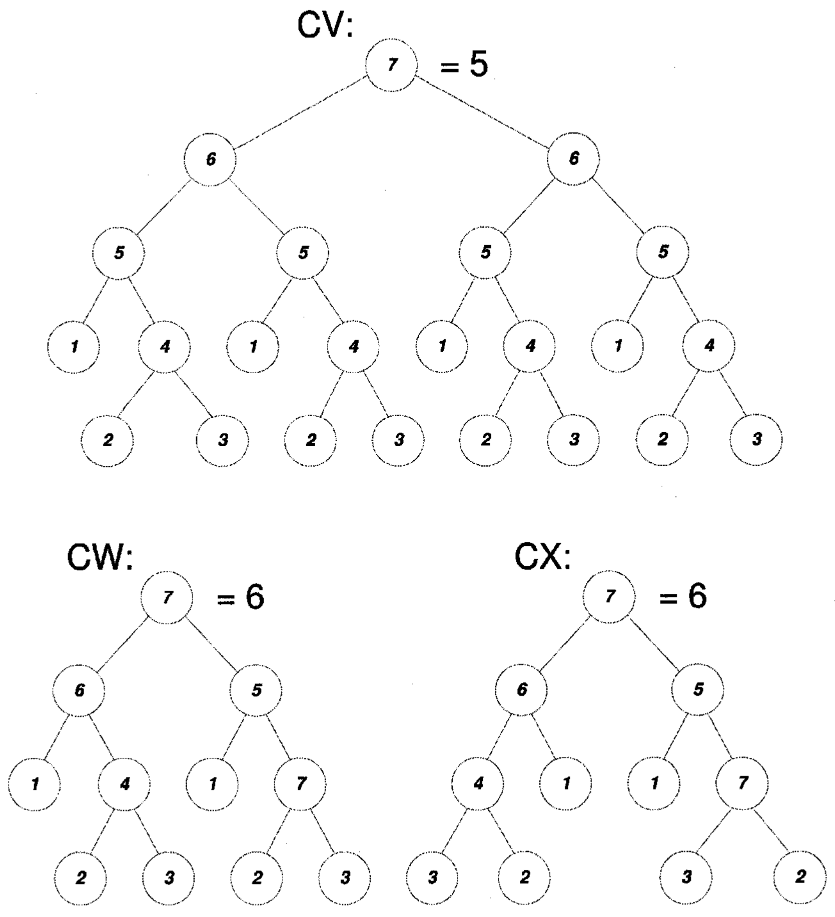
Figure 14



The trees "CQ" and "CR" are combinations achievable of height four with geometrically similar subtrees under the root. Trees "CS", "CT", and "CU" use the symbol "CR" to indicate the entire tree labelled "CR", which is attached as a subtree under the root node.

The height five tree "CV" shown in Fig. 15 has nice, regular numbering but only scores four from the screen. An idea strikes: perhaps a subtree rotation will matter to the problem. A little experiment with the rotation pairs "CW" and "CX", also shown in Fig. 15, give the same score of six.

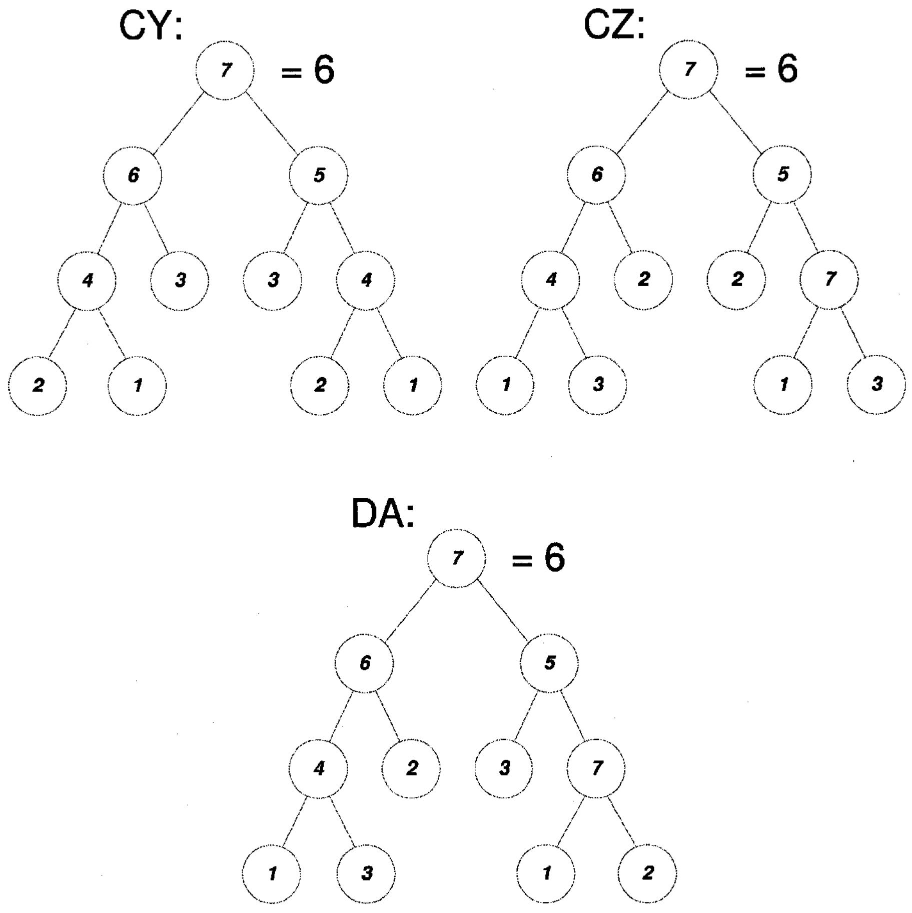
Figure 15



A large tree, another tree of similar geometry to previous high-scoring trees, and a rotational variant of this tree.

Some more tests with rotations of previously successful trees give the same results, shown in Fig. 16. It would appear that the scoring function is rotation invariant. At this point, the door opens, and the attendant invites us out for lunch, and for a discussion of our 79 attempts at this problem.

Figure 16



Rotational variants of previously successful trees, designed to test rotation invariance of the scoring function.

Appendix 1: How Does Genetic Programming Work?

Analysis of the Genetic Programming Room Allegory

As you can see from the above story, several interesting things were happening. The most apparent thing is that it is not at all obvious what kind of problem we are solving when the context and labels are removed. A second thing is that a scoring function that returns only a single integer makes it very difficult to measure progress. Did we move from a score of 5 to 6 because of the height of the tree? Was it due to the shape of the tree? Did the specific nodes that we chose make a difference? Does the "fitness function" like repetitive subunits? When placed in such a context, the number of possible hypotheses multiply almost without bound. Lacking auxiliary or additional information, we can't even tentatively determine what problem we were attacking. Much like Searle's original Chinese Room thought experiment, we are somehow ignorant of the information processing task that we are accomplishing. This is true even though we clearly *made progress* on the problem at hand without having any particular knowledge of the identity of the problem. Indeed, with a little more time and a few more trials, we might have been able to come up with a perfect solution without knowing the identity of the problem at all! It is this last point that gives us some confidence that genetic programming can make progress without explicit modelling of the problem at hand – after all, we managed to.

A little analysis of the problem-solving techniques used by the participant in the above allegory suggests some common threads that we might use to improve genetic programming. Some heuristics appear to have been used by the participant in attempting to solve the genetic programming problem. Firstly, try the simplest trees first. The searcher used approximately a breadth-first search through the space of all possible trees, at least until this became prohibitive. Secondly, try to infer relationships among entities. For instance, several attempts were made to determine the associativity of the operators 4, 5, 6, and 7. At the end, a tentative decision was made about rotations not being significant. Thirdly, go from the best. At each step, there is a best tree in mind, which is used as a template to make new variants. Finally, try little experiments to see if a particular intervention improves or degrades performance.

Of these four heuristics, genetic programming in its present form uses only the third, in the form of differential selection when choosing parents and in elitism. That alone can get you pretty far, but we can implement some of the other tricks as well. In this thesis, we discuss how trying the simplest trees first can be a very productive strategy in Chapter 5. We make an attempt at determining the relationships among entities herein, though it may well be a productive idea as well. Trying out little experiments is at the core of the scientist algorithm, which is introduced in Chapter 2 and expanded upon in the latter half of the thesis.

We hope that this small *gedankenexperiment* has been interesting and thought-provoking for the reader. One more thing. The problem in question in the GP room allegory is "give the solution to even-3-parity". That is, the task is to make a program that will answer TRUE if and only if zero or two of the inputs are TRUE. The three terminal nodes, "1", "2" and "3" are the input values of the problem, the three inputs that will be combined. The four connecting nodes, "4" through "7", represent the boolean

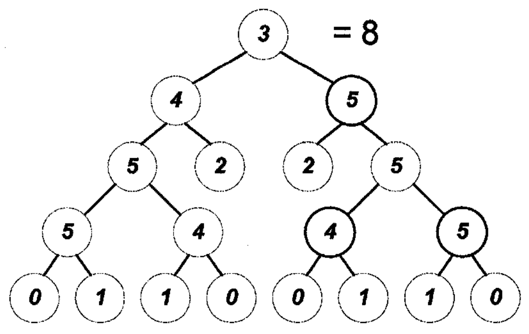
Figure 17

	Computational Effort		log Comp. Eff.	
k	*GP*	*GP+ADF*	*GP*	*GP+ADF*
3	96 000	64 000	5.0	4.8
4	384 000	176 000	5.6	5.2
5	6 528 000	464 000	6.8	5.7
6	70 176 000	1 344 000	7.8	6.1

Approximate computational effort to solve even-*k*-parity with 99% success rate as a function of the number of Boolean predicates considered, *k*. Computational effort numbers are accurate to roughly a factor of two, given the number of runs performed in these experiments. Trials are performed both with Automatically Defined Functions, marked as $GP + ADF$, and without, marked as GP. The data given for $k = 6$ for the GP treatment are extrapolated from the earlier data, and are consistent with the absence of any successes in 19 trials. All data are taken from [Koza, 2000].

functions AND, OR, negated-AND and negated-OR, respectively. The score presented to the subject is the number of rows of the truth table of the program which match the truth table row of the target problem. The best possible score is indeed eight, representing a perfectly correct solution. As shown in Fig. 17, genetic programming routinely solves this problem and related problems of larger size without any explicit knowledge whatsoever of the problem. The smallest tree that we could find that solves the problem is of size 19, although we did not try all the $1\,891\,919\,462\,400$ trees of sizes 13, 15, and 17 to be sure. One particularly elegant perfect solution of size 19, found using the size-weighted entropy code described in Chapter 7 is shown in Fig. 18.

Figure 18



A function tree generated by genetic programming and optimized by breaking ties by size-weighted entropy that perfectly solves the Even-3-Parity problem of this Chapter. The nodes "0", "1" and "2" represent the three inputs of the problem. "3" is the boolean function AND, "4" is the function OR, and "5" is the function Negated-AND. Three subtree rotations have been performed on the nodes marked with a heavy outline to increase the symmetry of the solution.

This page is intentionally left blank.

Appendix 1: How Does Genetic Programming Work?

References

[Alonzo 1995] L. Alonzo and R. Schott (1995). *Random Generation of Trees*, Kluwer.

[Barnum 2000] H. Barnum, H.J. Bernstein, and L. Spector (2000). "Quantum circuits for OR and AND of ORs." *Journal of Physics A: Mathematical and General* **33** (45): 8047-8057.

[Bonferroni 1935] C.E. Bonferroni (1935). "Il calcolo delle assicurazioni su gruppi di teste." *Studi in Onore del Professore Salvatore Ortu Carboni*, Rome, Italy: 13-60.

[Bryant 2001] C. Bryant, S. Muggleton, S. Oliver, D. Kell, P. Reiser, R. King (2001). "Combining Inductive Logic Programming, Active Learning, and Robotics to Discover the Function of Genes." *Linköping Electronic Articles in Computer and Information Science* **6**: no. 12. http://www.ep.liu.se/ea/cis/2001/012/.

[Chellapilla 1997] K. Chellapilla (1997). "Evolutionary Programming with tree mutations: Evolving computer programs without crossover." *Genetic Programming 1997: Proceedings of the Second Annual Conference*, Stanford University, USA, Morgan Kaufmann: 431-438.

[Christensen 2001] S. Christensen and F. Oppacher (2001). "What can we learn from No Free Lunch? A First Attempt to Characterize the Concept of a Searchable Function." *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2001*, San Francisco, USA, Morgan Kaufmann Publishers: 1219-1226.

[Christensen 2002] S. Christensen and F. Oppacher (2002). "An analysis of Koza's computational effort statistic for genetic programming." *EuroGP 2002*, LNCS 2278, Heidelberg, Germany, Springer-Verlag: 182-191.

[Christensen 2006a] S. Christensen (as viewed November 2006). "A spreadsheet for calculating the y-test". *http://www.scs.carleton.ca/~schriste/ytest.*

[Christensen 2006b] S. Christensen (as viewed November 2006). "A C++ library for calculating the y-test". *http://www.scs.carleton.ca/~schriste/ytestc.*

[Cohen 1988] Cohen (1988). *Statistical Power Analysis for the Behavioural Sciences*, 2nd ed., Hillsdale, USA, Lawrence Earlbaum Associates.

[Deb 2001] K. Deb (2001). *Multiobjective Optimization using Evolutionary Algorithms*, USA, John Wiley and Sons, Ltd.

[Dennett 1991] D. Dennett (1991). *Consciousness Explained*, London, UK, The Penguin Press.

[Fogel 2002] D. Fogel (2002). *Blondie 24: Playing at the Edge of AI*, San Diego, USA, Academic Press.

[Gibbons-Jean 1991] D. Gibbons-Jean and S. Chakraborti (Spring 1991). "Comparisons of the Mann-Whitney, Student's t, and alternate t tests for means of normal distributions." *Journal of Experimental Education* **59** (3): 258-267.

[Gilmour 1939] J.S.L. Gilmour and J.W. Gregor (1939). "Demes: A Suggested New Terminology." *Nature* **144**: 133.

[Holland 1975] J.H. Holland (1975). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*, Ann Arbor, USA, University of Michigan Press.

[Hotelling 1953] Hotelling (1953). "New light on the correlation coefficient and its transform." *Journal of the Royal Statistical Society, Series B* **15**.

[IBM 2006] IBM (as viewed November 2006). "Kasparov vs. Deep Blue the rematch". *http://www.research.ibm.com/deepblue.*

[Keijzer 2001] M. Keijzer, V. Babovic, C. Ryan, M. O'Neill, M. Cattolico (2001). "Adaptive Logic Programming." *Proceedings of the 2001 Genetic and Evolutionary Computation Conference*, San Francisco, USA, Morgan Kaufmann.

[King 2001] R. King, K. Whelan, F. Jones, P. Reiser, C. Bryant, S. Muggleton, D. Kell, S. Oliver (2001). "Functional genomic hypothesis generation and experimentation by a robot scientist." *Nature* **427**: 247-252.

[Kinnear Jr. 1993] K.E. Kinnear, Jr. (1993). "Evolving a sort: Lessons in genetic programming." *Proceedings of the 1993 International Conference on Neural Networks* **2**: 881-888.

[Koza 1992] J.R. Koza. (1992). *Genetic Programming: on the Programming of Computers by Means of Natural Selection*, Cambridge, USA, MIT Press.

[Koza 1992b] J.R. Koza. (1992). "Artificial Ant on the Santa Fe Trail." *Genetic Programming: on the Programming of Computers by Means of Natural Selection*, Cambridge, USA, MIT Press: 147-155.

[Koza 1992c] J.R. Koza. (1992). *Genetic Programming: on the Programming of Computers by Means of Natural Selection*, Cambridge, USA, MIT Press: 113.

[Koza 1992d] J.R. Koza. (1992). "Appendix B: Problem-Specific Part of Simple LISP Code, and Appendix C: Kernel of the Simple LISP Code." *Genetic Programming: on the Programming of Computers by Means of Natural Selection*, Cambridge, USA, MIT Press: 705-755.

[Koza 1994] J.R. Koza. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*, Cambridge, USA, MIT Press.

[Koza 1999] J.R. Koza, F.H.Bennett III, D. Andre, and M.A. Keane (1999). *Genetic Programming III: Darwinian Invention and Problem Solving*, San Francisco, USA, Morgan Kaufmann.

[Koza 2004] J.R. Koza, M.A. Keane, M.J.Streeter, W. Mydlowec, J. Yu, and G. Lanza (2004). *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*, Kluwer Academic Publishers.

[Koza 2004b] J.R. Koza, M.A. Keane, M.J.Streeter, W. Mydlowec, J. Yu, and G. Lanza (2004). "DVD Video included with Book." *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers.

[Koza 2006] J.R. Koza (as viewed November 2006). *http://www.genetic-programming.com/humancompetitive.html.*

[Lanczos 1964] C. Lanczos (1964). *SIAM Journal on Numerical Analysis ser. B* **1**: 86-96.

[Langdon 1998] W.B. Langdon and R. Poli (1998). "Why Ants are Hard." *Genetic Programming 1998: Proceedings of the Third Annual Conference*, Madison, USA, Morgan Kaufmann: 193-201.

[Langdon 1998b] W.B. Langdon and J.R. Koza (1998). *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!*, Norwell, USA, Kluwer Academic Publishers.

[Li 1997] M. Li and P. Vitanyi (1997). *An Introduction to Kolmogorov Complexity and Its Applications, 2nd Ed.*, Heidelberg, Germany, Springer Verlag.

[Luke 2001] S. Luke and L. Panait (2001). "A Survey and Comparison of Tree Generation Algorithms." *Proceedings of the 2001 Genetic and Evolutionary Computation Conference*, San Francisco, USA, Morgan Kaufmann.

[Mann 1947] H.B. Mann, and D.R. Whitney (1947). "On a test of whether one of 2 random variables is stochastically larger than the other." *Annals of Mathematical Statistics* **18**: 50-60.

[Maple 1996] Waterloo Maple Inc. (1981-1996). *Maple V Release 4.00a.*

[Masum 2002] H. Masum, S. Christensen, and F. Oppacher (2002). "The Turing Ratio: Metrics for Open-Ended Tasks". *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, New York, USA, Morgan Kaufmann.

[Moore 1998] F.W.Moore and O.N.Garcia (1998). "A Genetic Programming Methodology for Missile Countermeasures Optimization Under Uncertainty". Evolutionary Programming VII, 7th International Conference, EP98 LNCS 1447.

[O'Reilly 1995] U.-M. O'Reilly (1995). *An Analysis of Genetic Programming*, Ph.D. thesis, Ottawa, Canada, Ottawa-Carleton Institute for Computer Science.

[Pareto 1906] V. Pareto (1906). *Manuale di economia politica*, Italy.

[Poli 2000a] R. Poli (2000). "Exact Schema Theorem and Effective Fitness for GP with One-Point Crossover." *Proceedings of the Genetic and Evolutionary Computation Conference*, Stanford University, California, Morgan Kaufmann.

[Poli 2000b] R. Poli (2000). "Why the Schema Theorem is Correct also in the Presence of Stochastic Effects." *Proceedings of the Congress on Evolutionary Computation (CEC 2000)*, San Diego, USA: 487-492.

[Poli 2001a] R. Poli (2001). "General schema theory for genetic programming with subtee-swapping crossover." *Genetic Programming, Proceedings of EuroGP 2001*, LNCS, Milan, Italy, Springer-Verlag.

[Poli 2001b] R. Poli, J.E.Rowe, and N.F.McPhee (2001). "Markov chain models for GP and variable-length GAs with homologous crossover." *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2001)*, San Francisco, USA, Morgan Kaufmann.

[PlanetMath 2006] PlanetMath (as viewed on November 2006). "Principal Components Analysis." *http://planetmath.org/encyclopedia/HotellingTransform.html.*

[Press 1992a] W. Press, S. Teukolsky, W. Vetterling and B. Flannery (1992). "Chapter 9: Root Finding and Nonlinear Set of Equations." *Numerical Recipes in C, The Art of Scientific Computing, 2nd Ed.*, New York, USA, Cambridge University Press: 347 - 393.

[Press 1992c] W. Press, S. Teukolsky, W. Vetterling and B. Flannery (1992). "Chi-Square Fitting." *Numerical Recipes in C, The Art of Scientific Computing, 2nd Ed.*, New York, USA, Cambridge University Press: 659-661.

[Press 1992d] W. Press, S. Teukolsky, W. Vetterling and B. Flannery (1992). "Section 6.4: Incomplete Beta Function, Student's Distribution, F-Distribution, Cumulative Binomial Distribution." *Numerical Recipes in C, The Art of Scientific Computing, 2nd Ed.*, New York, USA, Cambridge University Press: 226-229.

[Schaeffer 1996] J. Schaeffer (1996). *One Jump Ahead: Challenging Human Supremacy in Checkers.*

[Schmidhuber 2004] Schmidhuber (2004). "Optimal Ordered Problem Solver." *Machine Learning* 54.

[Schöneburg 1994] E. Schöneburg, F. Heinzmann, and S. Feddersen (1994). *Genetische Algorithmen und Evolutionsstrategien: Eine Einführung in Theorie und Praxis der simulierten Evolution.*

[Schumacher 2001] C. Schumacher, M.D. Vose and L.D. Whitley (2001). "The No Free Lunch and Problem Description Length." *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2001*, San Francisco, USA, Morgan Kaufmann Publishers: 565-570.

[Searle 1980] Searle (1980). "Minds, Brains and Programs". *Behavioral and Brain Sciences* **3**: 417-424.

[Searle 1984] Searle (1984). *Minds, Brains, and Science*, Cambridge, USA, Harvard University Press.

[Soule 1998] T. Soule (1998). *Code Growth in Genetic Programming*, Ph.D. thesis, Moscow, USA, University of Idaho.

[Spector 1998] L. Spector, H. Barnum, and H.J. Bernstein (1998). "Genetic Programming for Quantum Computers." *Genetic Programming 1998: Proceedings of the Third Annual Conference*, Madison, USA, Morgan Kaufmann: 365-373.

[Spector 2003] L. Spector and H.J. Bernstein (2003). "Communication capacities of some quantum gates, discovered in part through genetic programming." *Proceedings of the Sixth International Conference on Quantum Communication, Measurement, and Computing*, Princeton, USA, Rinton Press: 500-503.

[Turing 1936] A.M.Turing (1936). "On computable numbers, with an application to the Entscheidungsproblem." *Proceedings of the London Mathematical Society, Series 2* **42**: 230-265.

[Turing 1950] A.M.Turing (1950). "Computing Machinery and Intelligence." *Mind* **59**: 433-460.

[von Dyck 1882] W.F.A. von Dyck (1882). "Gruppentheoretische Studien." *Mathematische Annalen* **20**: 16438.

[Weisstein 2006] E.W. Weisstein (as viewed November 2006). "Hypersphere." *http://mathworld.wolfram.com/Hypersphere.html*, MathWorld – A Wolfram Web Resource.

[Whitley 2006] D. Whitley, M. Richards, R. Beveridge, and A. da Motta Salles Barreto (2006). "Alternate Evolutionary Algorithms for Evolving Programs: Evolution Strategies and Steady State GP." *Genetic and Evolutionary Computational Conference (GECCO 2006)*, ACM Press: 919-926.

[Wilcoxon 1945] F. Wilcoxon (1945). "Individual comparisons by ranking methods." *Biometrics Bulletin* **1**: 80-83.

[Wineberg 2000] M. Wineberg (2000). "The Deme Approach in the GA Literature as Inspired by the Shifting Balance Theory." *Improving the Behaviour of the Genetic Algorithm in a Dynamic Environment*, Ph.D. Thesis, Ottawa, Canada, Ottawa-Carleton Institute for Computer Science: 13-17.

[Wolpert 1995] D.H. Wolpert and W.G. Macready (1995). "No Free Lunch Theorems for Search." *Technical report SFI-TR-95-010*, Santa Fe Institute.

[Wolpert 1996] D.H. Wolpert and W.G.Macready (1996). "No Free Lunch Theorems for Optimization". *IEEE Transactions on Evolutionary Computation* **1**: 67-82.

[Zimmerman 1989] D.W. Zimmerman and B.D. Zumbo (1989). "A note on rank transformations and comparative power of the student t-test and Wilcoxon-Mann-Whitney Test." *Perceptual & Motor Skills* **68** (3, pt.2): 1139-1146.

[Zimmerman 1992] D.W. Zimmerman (1992). "An extension of the rank transformation concept." *Journal of Experimental Education* **61** (1): 73-80.