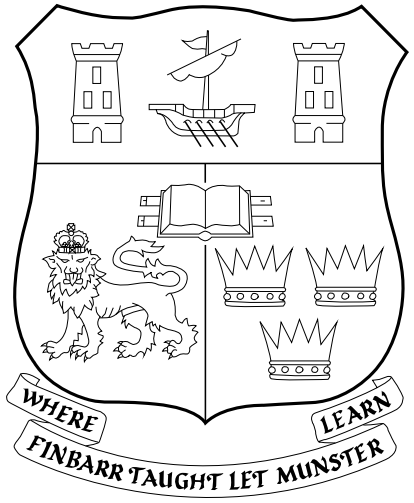
Computer Science Department

University College, Cork



Reducing Premature Convergence in Evolutionary Algorithms

by

Conor Ryan

B.A.

A thesis presented for

the PhD Degree of the

National University of Ireland

July 2, 1996

To my family, who have helped me through the last few years in more ways than they know.

Acknowledgements

I must first thank my thesis advisor **Dr. Gordon Oulsnam**. I owe an enormous debt of gratitude to **Dr. Peter Jones** of the Plant Science Department in UCC. Peter spent many hours answering all my biology questions, and made countless suggestions to me, many of which ended up in the thesis. I would also like to thank my father, **Dermot Ryan**, who made sure that the thesis is readable.

The head of the Computer Science Department, **Prof. Patrick O' Regan**, was extremely supportive, and made sure that I was able to make a living for the past three years. **Dr. Brendan Murphy** and **Seamus O'Shea**, of Cork Regional Technical College and Tralee Regional Technical College respectively, were both also very supportive and were extremely flexible and understanding with hours.

Several of the staff members of UCC also helped out in the past few years, **John Morrison** was often on hand for a discussion, while **Dr. Simon Foley** was always ready for an argument about proving the correctness of programs. **Humphery Sorenson** took it upon himself to provide me with transport, for which I am very grateful, if still a little mystified.

Several of my fellow postgraduate students contributed in some way to the final product. **Colm Dineen** helped me out with the parallel code and PVM, particularly the programming in Chapter 7, while **Colin McCormack** was always around to discuss any problem that came up. **Chris Higgins** set up my machine and often took time to criticise my code, while **Dave O'Byrne** kept the machine running, all the while trying to persuade me to change it. **Paul Walsh** co-wrote the papers with me that eventually became Chapter 5, as well as giving many useful suggestions on my thesis. **Maura O'Halloran** read much of this work and taught me how to spell Portuguese.

I have had much contact with other members of the EA community through e-mail and the Genetic Programming discussion list, and have had many useful discussions. **Kim Kinnear** was the first to explain GP to me, and was extremely encouraging right at the very start, while **Eric Siegel** has been in constant communication, and we will be writing a paper together "real soon now". **Dr. John Koza**, the inventor of GP, was also very

supportive, and really made me feel part of the GP community when I met him.

Dr. Zbignew Michalwicz very kindly presented my EP95 paper for me, despite us not having even met yet, and **Dr. David Fogel** was very helpful when I was preparing that paper.

This work was supported in part by **Memorex Telex Ireland Ltd.**

Abstract

We define *Evolutionary Algorithms* to be those algorithms which employ or model natural evolution. Generally, when an Evolutionary Algorithm fails to produce a satisfactory solution to a problem, it is because the population has *prematurely converged* to a suboptimal solution.

This thesis seeks to improve the performance of Evolutionary Algorithms by reducing the occurrence of premature convergence. All the extensions presented in this thesis are either naturally occurring phenomena, or are methods employed by biologists and / or plant and animal breeders. In all the cases examined in this thesis, it is shown that the less human control there is with evolution, the better a population will perform.

A number of standard benchmark problems are examined, and new, biologically-inspired approaches are presented. A new selection scheme involving multiple fitness functions is introduced. This scheme is applied to the optimisation of multi-objective functions and multi-modal functions.

Genetic Programming is applied to a new problem area, the autoparallelisation of serial programs, through the use of techniques developed in this thesis.

The notion of *addditive diploidy*, a type of diploidy that occurs naturally in biology, is introduced and applied to Genetic Algorithms. Additive diploidy is shown to outperform traditional, dominance-oriented, diploidy on a difficult test problem.

A new benchmark problem for Genetic Programming is introduced. This competition-oriented benchmark permits the direct comparison of two or more possible solutions. In producing individuals for this benchmark, Genetic Programming is also shown to be suitable for the evolution of event driven programs.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

This thesis seeks to improve the performance of Evolutionary Algorithms by taking examples from nature. Evolutionary Algorithms are computational models which use artificial evolution to solve problems. When using evolutionary algorithms to solve a particular problem, one must be careful to strike a balance between *selection pressure*, the means by which a population steadily advances towards a solution, and *diversity*, the reservoir of genetic material necessary to enable selection pressure produce a useful solution. Too much selection pressure produces a non-optimal solution, while insufficient selection pressure doesn't force a population to evolve.

The quality of solution (and time taken to produce it) generated by an evolutionary algorithm also depends on the training set it is given. The training set is the set of testcases that an individual is applied to when calculating its fitness. Too much training on a particular set can lead to loss of diversity, while training on an excessively large set can lead to prohibitively long training times.

Evolutionary algorithms are subject to many parameters, and the incorrect choice of these parameters can lead to the sub-optimal performance of a population. There are often no fixed rules which one can follow when selecting various parameters, so one is reduced to a certain degree of trial-and-error.

It is suggested that these problems, and the performance of evolutionary algorithms in general, can be tackled by looking to natural biology. Few species of plant or animal

evolve in total isolation in a stable environment, yet this is how most evolutionary algorithms function. It is shown that for the cases considered in this thesis, the less human control or interference there is with evolution, the better it performs. Virtually every part of the plant or animal world is under genetic control, and it is suggested that the entrusting of parameter control to evolution may lead to better performance than could be hoped for if it were under human control. It is also found that allowing several species of specialist individuals to develop, who can then co-operate with each other, either by mating or by working in teams, gave an excellent improvement in performance.

1.2 Contributions of this thesis

- *Pygmies and Civil Servants.* Chapter 3 presents a new selection method taken directly from plant breeders. This method is shown to permit the use of very strong selection pressure even with very small populations. It is also shown to outperform current methods at some benchmark problems.

- *Sociological Modeling.* The model developed in chapter 3 is then extended to model certain sociological phenomena. Characteristics such as racism and independence are allowed to evolve, and their effect on the performance of a population is examined.

- *Demonstration of the power of Meta-GA.* Chapters 3 and 4 demonstrate the advantage of using a Meta-GA to control the parameters of a run. A number of different approaches to the evolution and the use of Meta-GA are considered.

- *Introduction of a new multi-modal function optimisation scheme.* Building on the models developed in the previous chapters, chapter 5 introduces a new scheme for multi-modal functions. This new scheme, the *Races Genetic Algorithm* is shown to outperform all currently existing genetic methods for both the discovery and maintanence of all peaks in a fitness landscape in a number of standard benchmarks.

- *Application of GP to new, real world problem areas.* A new tool, the *Paragen* system is introduced. Paragen applies GP to the problem of autoparallelisation of serial programs, and shows that autoparallelisation is possible without the application of data dependency analysis.

- *Introduction of two new diploidy schemes.* The more complex life forms on the planet, such as higher plant life and most animal life, tend to use diploid genetic structures. These structures are also often able to react more quickly to changes in their environment. These facts are often overlooked by the EA community because of the difficulty involved in the encoding of genes. Two new schemes, one for binary structures and the other for high level structures are introduced. These two schemes are shown to be more general than existing methods, and to be better at preserving genetic diversity in a changing environment.

- *Introduction of a new benchmark problem for GP.* A new benchmark problem for GP, *GPRobots*, is introduced. The benchmark is a simulated, real time environment that allows the direct comparison of two or more solutions from different approaches to GP.

- *The application of GP to event driven programming.* GP is shown to be ideally suited to the generation of event driven programs, and a model, based on the work in previous chapters, is devised which permits teams of co-operating individuals to compete with other teams.

1.3 Organisation of the thesis

The thesis is organised as follows:

Chapter 2 : Background

A brief introduction to Evolutionary Algorithms is given, along with a detailed description of the Simple Genetic Algorithm. The more common Evolutionary Algorithms are also reported on, along with reasons why they are (or are not) used in subsequent chapters.

Chapter 3 : Pygmies and Civil Servants

This chapter describes common approaches to solving multi-objective functions and introduces a new selection scheme, the Pygmy Algorithm[Ryan, 1994b]. The Pygmy

Algorithm, which is modeled on naturally occurring phenomena in nature, is compared to a number of existing methods and is shown to outperform them at a lower cost.

The chapter then extends the model to employ a meta-GA, and investigates the potential costs and benefits of allowing sociological behaviour to influence breeding schemes[Ryan, 1994c]. It is shown that using a meta-GA to control some of the population parameters makes the population far more general.

Chapter 4 : The Races Genetic Algorithm

The algorithm presented in the previous chapter is extended so it can be applied to multi-modal fitness functions[Ryan, 1995c]. A new algorithm, which automatically splits up a population to spread it across the search space, is introduced. This algorithm, the Races Genetic Algorithm, is shown to be better than current genetic methods at locating peaks in the landscape.

Chapter 5 : The Paragen System

Using the results of the previous chapters, a new system for the autoparallelisation of serial programs, the *Paragen* [Ryan and Walsh, 1995] system, is introduced. Autoparallelisation is a real world problem of great significance, which normally involves complicated analysis of a program. Paragen, however, is a fully automatic system for autoparallelisation, and is capable of taking an arbitrary serial program and converting it into a functionally equivalent, parallel one. Paragen relies completely on GP and the selection schemes described in chapters 3 and 4 to produce the parallel program, and does not employ data dependency analysis of any kind.

Chapter 6 : The Degree of Oneness

A new approach to implementing diploidy, *additive diploidy*[Ryan, 1994a], is introduced. This approach, which also occurs naturally in biology, can be used to implement diploid genes for binary structures or high level structures. A problem similar to that described by [Goldberg and Smith, 1987] is constructed, and additive diploidy is shown to outperform other current diploidy schemes at reacting to an unstable environment for this problem.

Chapter 7 : GPRobots and GPTeams

This chapter introduces a new, competition-oriented, benchmark problem for Genetic Programming [Ryan, 1995a]. This problem involves the evolution of individuals to control robots in a realistic, real time environment. Then, using techniques developed in the previous chapters, GP is used to generate *event driven* programs for use in the benchmark. GP is shown to be extremely suitable for the generation of event driven programs, and programs that use an event structure are also shown to be better able to balance a number of different behaviours than those generated using traditional methods.

Chapter 8 : Conclusions and Future Work

The final chapter provides an overview of the work presented in the thesis. It then describes further work that can be examined and new directions of research that are suggested in the thesis.

Chapter 2

Background

2.1 Introduction

Evolutionary Algorithms are computer algorithms which use artificial evolution to solve problems. This chapter introduces evolution and gives a brief overview of the simple Genetic Algorithm, the most common Evolutionary Algorithm. There then follows a description of the main Evolutionary Algorithms with a discussion on their suitability for use in this thesis.

2.2 Evolution and Natural Selection

Evolution is the process of change over time. Little of the world around is not subjected to change as time progresses, and such change generally has a distinct continuity to it. Ideas and even behaviours change as their surroundings change, machines such as aeroplanes and computers have evolved as more and more ingenious designs for them have been discovered. Plant and animal life also thrive on this change, and all species of life on this planet are the product of evolution.

The driving force behind evolution, as described by Darwin[Darwin, 1859], is natural selection. Unlike the prevailing opinion of the day, Darwin postulated that there is no deterministic or finalistic drive behind evolution. He said that evolution was driven by chance. At the time, these claims were almost heretical, in particular, his adherence to the principle of survival of the fittest.

In any population, there are always individuals who are fitter than others. Such individuals live longer and thus get the chance to produce more offspring than individuals of average fitness. Conversely, unfit individuals, or individuals poorly adapted to their environment, tend to produce less offspring than individuals of average fitness. In this way, the genes, and hence the characteristics, of fitter individuals propagate through a population, until, assuming those characteristics are better than others currently in the population, all of the population contains those characteristics.

2.3 Evolutionary Algorithms

Evolutionary Algorithms(EAs) are algorithms which have turned away from traditional top-down programming paradigms and embraced the philosophy of natural selection. Like natural evolution, EAs maintain a population of individuals. By manipulation of the genetic structure of these individuals, the *genotypes*, EAs evolve progressively better *phenotypes*, the physical expression of a genotype. To all intents and purposes, EAs treat their populations as though they were made up of live creatures. Within the confines of the electronic environment of a EA creatures are born, die, interact with their fellow individuals under certain circumstances and, of course, have sex.

It is within this electronic environment that the learning ability of EAs lies. The environment is made up of both the population and the EA, the latter which is used to decide how individuals behave towards each other. It is through a process of engineering local interactions, in particular recombination, that global behaviour emerges. On its own, each individual is usually incapable of learning and it is not uncommon for many of the individuals processed to be incapable of performing any actions at all.

In a similar manner to the very processes they model, EAs themselves have evolved and mutated until we are now faced with a number of algorithms, each taking a different approach to the modeling of evolution. Figure 2.1 shows the main algorithms. This thesis takes the view that the differences between these algorithms is more on an implementation level than a philosophical one, and uses which ever algorithm is more convenient for the problem at hand.

The most well known, and arguably the first[Holland, 1975][Goldberg, 1989a] , EA

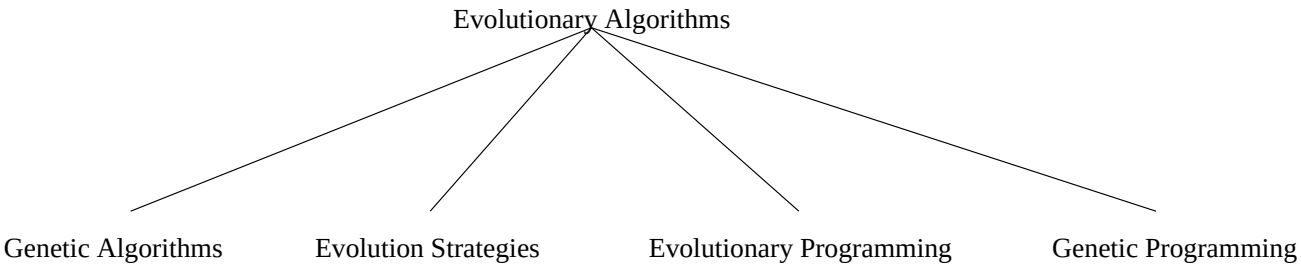


Figure 2.1: The evolution of evolutionary algorithms.

is the Simple Genetic Algorithm(SGA). The SGA is fairly representative of the other EAs, as they all use the same steps.

2.4 The Simple Genetic Algorithm

The simple genetic algorithm is made up of the five major steps in figure 2.2. Each step is now examined in detail.

2.4.1 Step 1 : Randomly create an initial population.

The origin of life in GAs happens in a somewhat less romantic fashion than the sudden spark which gave rise to life from the primordial ooze on earth. In a manner not unlike that suggested by the theory of "directed panspermia[1]" [Francis Crick and Lelie E. Orgel], the implementor of a GA seeds the initial population with an appropriate diversity of genetic material from which evolution springs. This initial seeding generally involves the encoding of every possible phenotypic trait into binary form, which then serves as the genotype.

2.4.2 Step 2 : Calculate a score for each individual against some fitness criterion.

An integral part of GAs is the fitness function; it is analogous to the lifetime of the individual and is simply a measure of how well that individual performed over its lifetime. In nature an individual must perform all manner of activities in order to be considered fit; it

[1] *Panspermia* is the belief that the atmosphere contains many invisible and dormant germs or seeds. Every so often one of these seeds is activated, and causes life on earth to change.

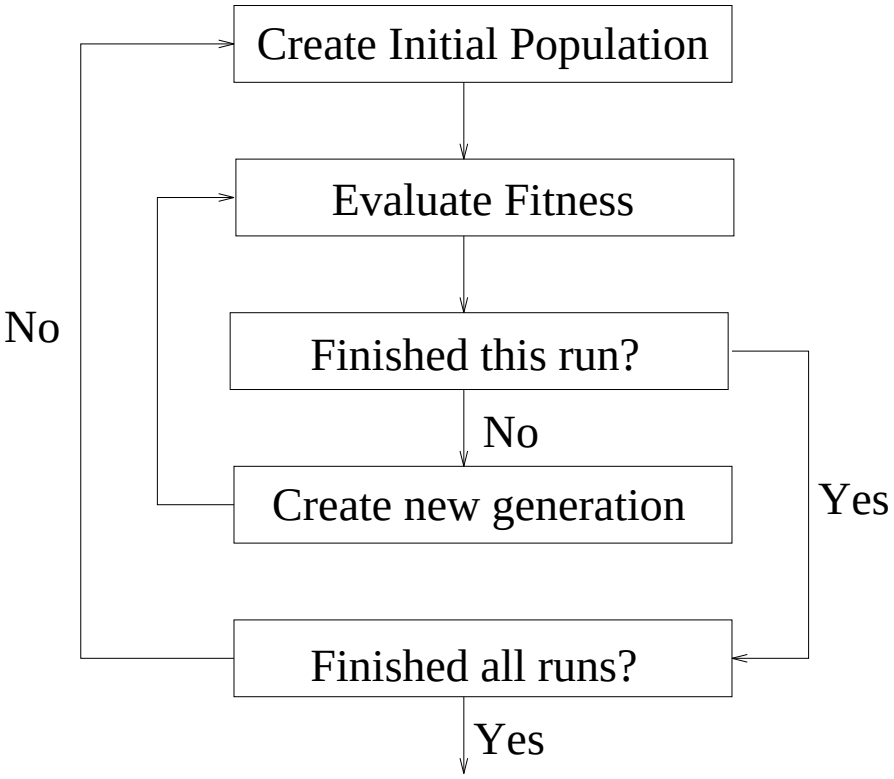


Figure 2.2: Flowchart for the simple genetic algorithm

must not be eaten, it must not drink poisoned water, it must be able to find food, and so on. In GAs, tests for these abilities are neither feasible nor useful so some fitness criterion must be imposed on the population. In this respect, GAs diverge from natural evolution, but this external fitness function is advantageous as the sole measure of an individuals fitness is its suitability to a particular problem.

2.4.3 Step 3 : Use the top scoring individuals to create the next generation.

The top scoring individuals are then selected to breed the next generation using a selection method referred to as *roulette wheel selection*, which, as the name implies, selects prospective parents at random from the entire population. The selection scheme used by a GA is intended to be analogous to natural selection described above. To bias the selection toward more fit, and thus higher performing individuals, each individual is assigned a probability $P(x)$, which is the fitness of individual x relative to the rest of the population as shown in equation 2.1 below.

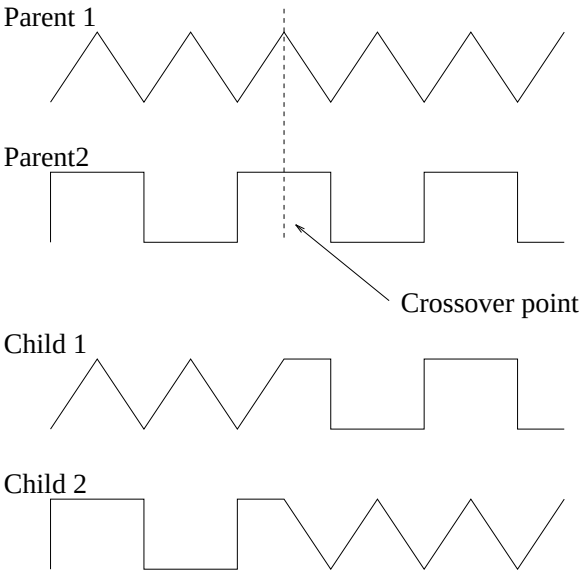


Figure 2.3: Crossover in Genetic Algorithms

$$P(x) = \frac{f_x}{\Sigma f_i} \tag{2.1}$$

Two parents are selected at a time and are used to create two new children for the next generation using crossover as in figure 2.3. Two children are created to help prevent the loss of any genetic material. These new children may be subjected to mutation which involves the "flipping of a bit", i.e. randomly changing the value of a gene. Mutation traditionally occurs in the SGA at a rate of about 1 bit per 1000. An example of mutation appears in figure 2.4. Another process occasionaly applied to a single individual is that of *inversion*, as shown in figure 2.5. Inversion involves swapping two genes on an individual, an operation which simply reorders the genetic material, but does not change it.

This process is repeated until a new population has been created. Roulette wheel selection ensures that it is likely that the top performing individuals are given the opportunity to spread their genes through the new population. It is quite possible that an individual may be selected several times for breeding, or even selected to breed with itself which leads to the creation of clones. It is also reasonable to expect some of the relatively unfit individuals to be selected for breeding, due to the inherent randomness of this process.

The final operation that can be performed on individuals is that of *reproduction*. Reproduction is the copying of a single individual into the next generation. An individual

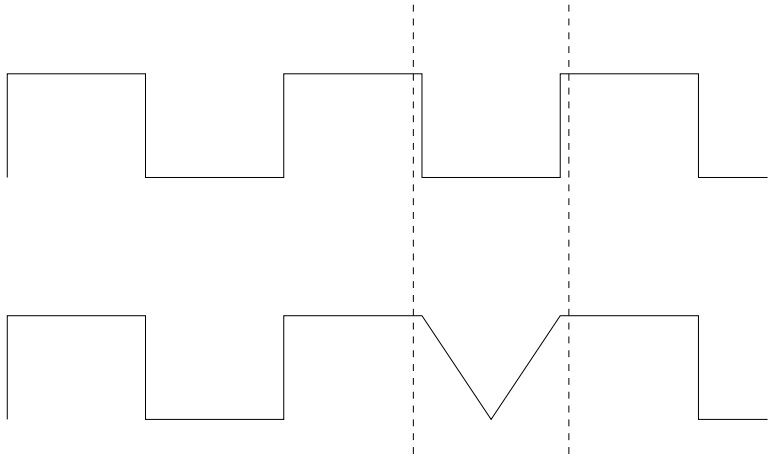


Figure 2.4: Mutation in Genetic Algorithms

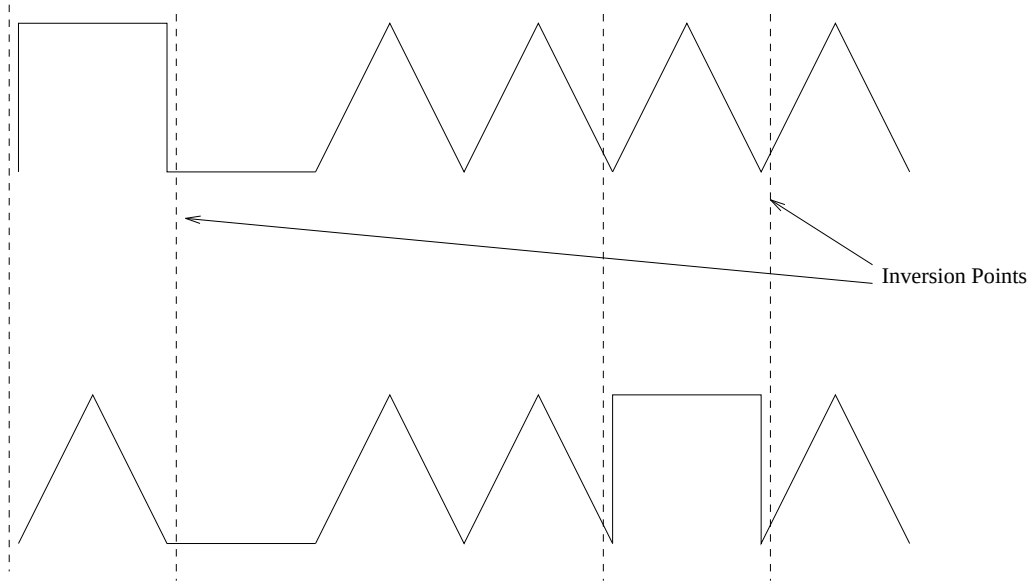


Figure 2.5: Inversion in Genetic Algorithms

copied in this way can be looked upon to be "living" longer than other individuals, and as a result, will have more opportunities to breed.

2.4.4 Step 4 : Repeat steps 2 and 3 until some stopping condition is reached.

As mentioned previously, artificial evolution is not open-ended evolution, terminating under some predefined stopping condition such as the appearance of a perfectly fit individual. In some cases it is impossible to identify a perfect individual, so a *best-so-far* individual is identified. The SGA is run for a number of generations and the *best-so-far* individual at the end of the run is reported as the solution.

Not every run of SGA terminates successfully. Sometimes a process known as *genetic drift* occurs, where a population fixates on one particular gene combination, losing all others. In some cases, this gene combination will be the optimal combination, so this is not a problem. However, in other cases it won't be the optimal, and some of the genes that are necessary are no longer in the population, so the SGA will not solve the problem. This is known as *premature convergence*, and much of this thesis is concerned with its prevention.

2.4.5 Repeat steps 1 to 4 until all runs are finished

EAs are directed search techniques, but are inherently random. Because not every run is guaranteed to produce a satisfactory individual, it is rare to run a GA only once for a problem. Typically, depending on the length of time for each run, a GA is run 20 or more times on a problem, with a different initial population. The best individual(s) produced in all of these runs is then selected as the solution.

2.5 Other Evolutionary Algorithms

Figure 2.1 shows the more well known EAs, and each is discussed below. Not all of these are used in this thesis, but most of work it contains could have been implemented in any one of them. Each chapter explains why a particular EA, or, in many cases, a variant thereof, was used.

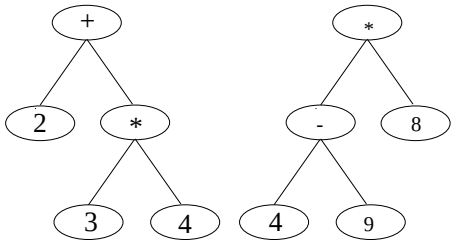


Figure 2.6: Some simple parse trees

2.5.1 Genetic Programming

Genetic Programming(GP)[Koza, 1992] is a relatively new technique for the generation of computer programs. GP differs from GA in the representation of individuals, using trees instead of fixed length strings. For example the LISP program **(+ 2 (* 3 4))** could be represented as in Figure 3.2. These program trees are made up of two fundamental building blocks: nodes and leaves. Nodes are simply functions such as + * which take one or more arguments, while the leaves are terminals, i.e. numbers or zero-argument functions. The first major step in any implementation of GP is to correctly identify the necessary functions and terminals, and to ensure that any combination of them well result in a syntactically (although not necessarily functionally) correct program.

The use of trees greatly extends power of EAs, for instead of fixed length structures, it is now possible to evolve genuine programs. Although in its infancy, GP has achieved widespread recognition in the EA community for its flexibility in evolving programs.

Crossover in Genetic Programming

GP uses a similar generational approach as the simple genetic algorithm, but, because of its tree structures, uses a different crossover scheme. Crossover in GP is implemented by swapping sub-trees from two individuals. Like the simple genetic algorithm this results in the creation of two new, syntactically correct individuals, see Figure 2.7. GP also utilises the *reproduction* operator discussed above. Like GAs, the GP reproduction operator simply copies an individual unchanged into the next generation.

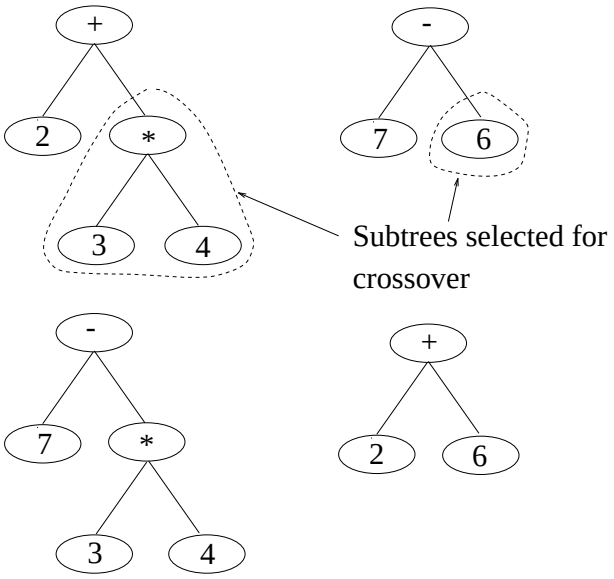


Figure 2.7: Crossing over two parent trees by swapping sub-trees

2.5.2 Evolutionary Programming

Evolutionary Programming[Fogel et al., 1966] was one of the earliest EAs. Unlike GAs, EP does not rely on fixed length structures, but permits individuals in the initial population to be of different lengths. These individuals are then tested, and parents for the subsequent generation are selected stochastically.

It is in the creation of the new generations that EP differs from most other EAs, for it does not employ any crossover. Instead, individuals selected as parents are subjected to mutation to produce children.

The advantage of EP over perhaps all other EAs is that it does not rely on any particular structure or representation. Any structure that can be mutated can be evolved using EP. In recent years EP has been enjoying a period of renewed interest, particularly with the creation of an annual conference in 1992[Fogel and Atmar, 1992]. However, most of the simulations presented in this thesis rely on crossover to combine very different individuals, quite the opposite approach to EP which does not combine any individuals. For this reason, EP is not employed in this thesis.

Parent 1 : `1 0 0 1 1 1`

Parent 2 : `0 1 1 0 0 0`

Mask: `1 0 0 1 1 0`

Child 1 : `0 0 0 0 0 1`

Child 2: `1 1 1 1 1 0`

Figure 2.8: Uniform Crossover

2.5.3 Evolution Strategies

A similar approach to that of GAs is taken by Evolution Strategies [Rechenberg, 1973]. Evolution Strategies (ES) also use fixed length structures, but instead of the usual binary structures used by GAs, ESs have real valued genes.

The emphasis in ES is more on the acquisition of *behaviour* rather than structure [Angeline, 1993]. Each position in an ES (i.e. a real number) marks a behavioural trait, and an individual's behaviour is the composition of these traits.

Crossover in ES is intended to produce children that are behaviourally similar to their parents, and there are three different approaches[Baeck, 1992]. The first, *discrete* recombination, is similar to a method often used in GAs, *uniform* crossover[Syswerda, 1989]. Uniform crossover involves creating a *crossover mask*, a binary string the same length as the parents. A 0 in the mask results in the relevant gene being selected from the first parent, while a 1 results in the second parent donating the gene. The crossover mask is a random string, and generally ensures that each parent contributes equally to the child. An example is shown in figure 2.8.

The other two methods exploit the fact that the genes are real-valued. The first of these, the *intermediate* recombination operator, determines the value of the child's genes by averaging the two values in the parents genes. The second method, the *random intermediate* recombinator, probabilistically determines the evenness of the contribution of each parent for each parameter.

In many cases, the simulations of evolution in this thesis rely on variable length

structures - in the cases where they don't, they rely on binary genes. It was found that ES were not suitable for these simulations.

2.5.4 The best EA

Although each particular EA has its proponents, it is impossible to say which is the best. Each one was designed with a particular problem in mind, e.g programming in GP, real value problems in ES etc., and so, it is the responsibility of an implentor to decide which best suits the problem.

Several of the chapters in this thesis use their own, tailor-made EA. In some cases the SGA is modified, in others, a variant of GP is used. Many details, such as population size, structure, crossover rate etc., can be applied to all varieties of EAs with little change. However, the most important aspect of EAs is that the best EA depends on what problem is to be solved, and it is often not obvious which method will work best.

2.6 Evolution and Learning

Artificial Intelligence often divides methods into two categories, *strong* and *weak*. Strong methods have much problem-specific information built into them, such as expert systems, while weak methods have little or no information. By arming them with knowledge, strong methods tend to perform better than weak methods, but only on a very limited domain.

The total separation of problem from problem solver would appear to put EAs in the weak method category, but, EAs tend to learn about a problem as they solve it, and use this knowledge to further improve any solutions they derive. This use of knowledge would now appear to suggest that EAs are in fact a strong method. At the very least, they are a combination of the two, a "strong weak method". This was noted by [Angeline, 1993] in his detailed analysis of EAs, when he coined the phrase *Evolutionary Weak Method* to describe EAs.

2.7 Summary

There are a number of Evolutionary Algorithms. Depending on the problem area, some are more suitable than others, but all are more or less grounded in the same theory of populations of individuals evolving to a solution.

All EAs are what are described as "strong weak methods", in that when applied to solving a problem, they learn about it, and apply this acquired knowledge to further improve their solutions.

Subsequent chapters demonstrate the *strong weak* nature of EAs, through successfully applying them to a diverse array of problems.

Chapter 3

Pygmies and Civil Servants

3.1 Introduction

This chapter introduces a new selection scheme, the Pygmy Algorithm [Ryan, 1994b]. The Pygmy Algorithm adopts a mating scheme taken directly from population genetics and reduces premature convergence in Genetic Algorithms and Genetic Programming. The resulting improvement in performance is shown to allow stronger selection pressure and also permits the use of small populations at very little extra computational cost and with little risk of losing genetic material.

The chapter then extends the Pygmy Algorithm further, and, through the use of several simulations which model sociological behaviour, demonstrates the power of a Genetic Algorithm to evolve its own control parameters in parallel with evolving a solution to a problem.

3.2 Reducing Premature Convergence in Genetic Algorithms

When applying a GA to problems with variable length solutions, it is often desirable to evolve a solution which uses as few instructions as possible. It is also necessary to preserve a population which, although being directed to some solution, maintains a degree of diversity to prevent premature convergence on a non-optimal solution. The degree to which a population is directed tends to reduce its diversity, and for this reason, elitism, the use of only the top percentage of the population for breeding, tends to be overlooked when

searching for a method of avoiding premature convergence.

Unfortunately, most previous methods for avoiding premature convergence, such as Crowding [DeJong, 1975], Sharing [Goldberg and Richardson, 1987], restricted mating, spatial mating etc. all involve extra computation, while methods which reward parsimony [Koza, 1992] or try to combine two measures[Horn et al., 1994], run the risk of individuals trading off various parts of the fitness function.

This chapter presents a new method for avoiding premature convergence at a very small cost. A secondary aim of this chapter is to defend the use of elitism in even relatively small populations to enable them to enjoy the benefits normally associated with elitism, i.e. quick convergence to an optimal solution.

3.3 The Problem Space

To demonstrate some of the more interesting methods mentioned above, the work described here evolves a minimal sorting network [Knuth73]. Sorting networks present a two-fold and possibly three-fold problem; a K-sorting network must correctly sort any combination of K numbers, but it must also do so in as few exchanges as possible. One can also consider the parallelism of a network, for swaps such as the first three in figure 3.1 may be executed in parallel. However, this work will not consider parallelism.

Sorting networks, not to be confused with sorting algorithms, are a list of fixed instructions which sort fixed length lists of numbers. Sorting networks have a convenient diagrammatic representation as in Figure 3.1; each input line represents one number and each connection, or "comparator module" compares the numbers at those positions and swaps them if necessary. For instance, the top left hand connection would be coded as

(if (> K[1] K[2]) (swap K[1] K[2]))

3.3.1 Genetic Programming Or String GA?

All the simulations in this chapter use a variable length String GA, which allows individuals to grow and contract during evolution. This permits individuals to vary the number of instructions that they use. Genetic Programming would also permit this however, and the Pygmy Algorithm has already been shown to work with GP [Ryan, 1994b].

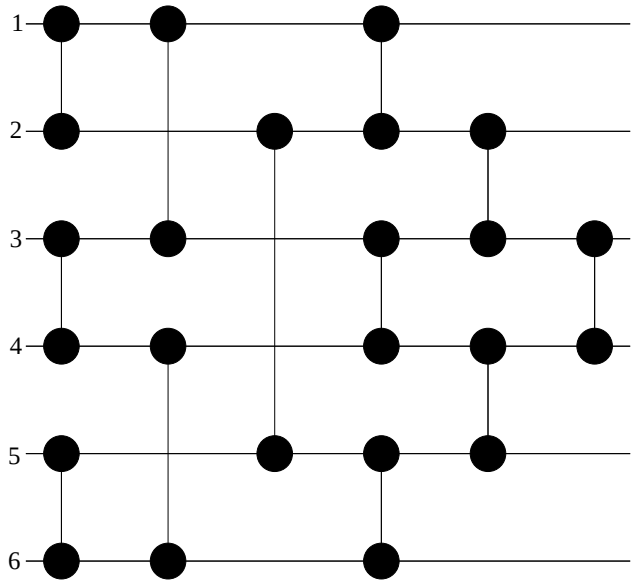


Figure 3.1: Network sorter for six numbers

3.3.2 Implementation Notes

Sorting networks can be generated for any amount of numbers. In fact, this is not the first time artificial evolution has been applied to their generation. The most notable effort was that of Hillis [Hillis, 1989] in which he evolved sorting networks for 16 inputs. As the amount of inputs to a sorter increases, the amount of compares required increases exponentially, as does the time required to evolve a sorter. It was considered more important to be able to examine a large quantity of results for relatively easier problems than a small quantity of results for extremely difficult ones, so this chapter concentrates on smaller networks.

A single run to evolve a 9-number sorting network with 5000 individuals running for up to 500 generations took up to 48 hours of CPU time on Decstation 5000/125, but for a 6-number sorter took less than ten minutes. For this reason, all the experiments in this chapter try to evolve a 6-number sorter, which results in a very large amount of empirical data.

As with most GA experiments, the smaller the population, the more difficult it is to produce a satisfactory solution. Because of this, the population size in the experiments below is changed to vary the difficulty. The relationship between population size and difficulty was not linear, but in general, the smaller the population, the more difficult the

problem.

3.3.3 The Benefits of Elitism

An elitist breeding strategy differs from traditional, roulette-wheel strategies in that only the top 10-20% performing individuals are permitted to mate. A potential problem, especially with small populations, is that a group selected in this way may not be representative of the population at large. Genes which are common in the population may not appear at all in the elite group, and one cannot discount the possibility that those genes may be useful at some future time.

The advantage of this is that poorly performing individuals do not hold back the rest of the population; elitism tends to be used when the implementor is more concerned with quick convergence on a solution than maintaining variety in a population. A method which permits elitism while still maintaining diversity could well vindicate elitism.

This chapter concentrates on methods which permit the use of elitism, but which can also use other selection methods, depending on their suitability to the problem.

3.4 Traditional Methods

Any suggested improvements to genetic algorithms can be appreciated only when viewed beside existing methods. To aid this understanding, a simple implementation of GA to the problem of evolving minimal sorting networks is presented using the following strategy. Individuals were of variable length, up to a maximum of 50 instructions, all of which were of the form **(Compare X Y)** which compared the numbers at positions X and Y and exchanged them if necessary. All the other experiments in this chapter use instructions such as these.

An elitist strategy was used with a breeding pool of 20% of the total population maintained. In the event of a tie in the fitness score, the individual with the fewest instructions was rated higher; this ensured that as the population evolved, the genes of shorter individuals propagated through the population.

As stated in the abstract, it is the intention to produce an algorithm that can preserve diversity under very strong selection pressure. To exaggerate the selection pressure

associated with elitism further, and thus increase the difficulty in maintaining diversity, a bias in favour of the higher performing individuals was included.

3.4.1 The Fitness Function - Punish or Reward?

It has been shown [Knuth, 1973] that a K-number sorting network that will correctly sort every K combination of 0s and 1s will sort every K combination of numbers. For 6-number sorters this involved 2^6 (64) fitness cases. To measure how well a set of numbers is sorted, the means usually employed is the counting of what Knuth called "inversions", which are considered to be numbers out of order. In a list sorted in ascending order an inversion is considered to be each number that has a larger number preceding it. Thus the sequence 1 2 3 4 5 6 has no inversions while the sequence 1 2 3 4 6 5 has one inversion.

A problem with this method is that it is more suited to traditional methods of sorting which recursively move through the list, so a number can be counted as being out of order several times. A result of this is a situation where a sequence such as 6 2 3 4 5 1 has 9 inversions with only two numbers needing reordering, while the sequence 1 4 2 5 3 6 has three inversions yet has four numbers in the incorrect positions.

This is not very suitable for sorting networks which do not move through the list but which arbitrarily select two numbers to put in the correct order. Measuring inversions seems to take the approach of punishing an imperfectly sorted list rather than rewarding a partially sorted list. To ensure the latter occurred the following scoring method was used :

```
score=0;
for (i=1;i<=5;i++)
     if (num[i]<=num[i+1]) score++;
```

This means that it is possible for an individual to score five points for each correctly sorted test case so that an individual which correctly sorts all 64 cases will receive a score of 320. This partial fitness was extremely useful early on in each experiment as it was not uncommon for every individual in the initial population to fail to sort every test case (except those already sorted). The score derived in this way is referred to as an individual's reward; for the initial experiments an individual's fitness is equal to its **reward**.

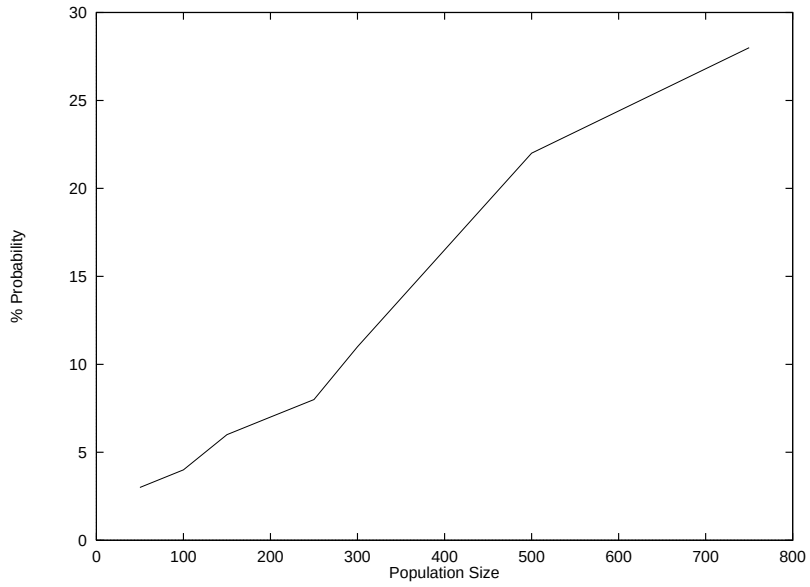


Figure 3.2: Probability of a perfect Individual appearing

3.4.2 Early Results

The above experiment was applied to a number of different population sizes, varying from 50 to 750 individuals, the results of which can be seen in Figure 3.2. Each experiment was repeated on 1,000 different populations, and, to aid comparison, each population size started with identical seeds when generating the initial population in each case. A perfect individual is a sorting network which will correctly sort any combination of 6 numbers in 12 exchanges. Twelve exchanges has been shown [Knuth73] to be the minimum needed for such a sorting network. An example of one such network was given earlier in figure 3.1.

High performing individuals were kept in the breeding pool until dislodged by a better individual; because a constant training set was used these individuals did not need to be tested again, so the entire population could be replaced each generation. To get a true idea of how well the better genes were being preserved, no mutation was permitted. Thus, if a crucial gene disappeared from the elite group, there was no way of retrieving it.

It was found that every experiment produced a 100% correct individual, but not all experiments produced a perfect individual, i.e. 100% correct using only 12 exchanges.

Figure 3.2 shows that the adjusting of population size has a significant effect on the probability of a perfect individual appearing. This is because of the effect a population's size has on the variety of individuals contained within it. A population which does not

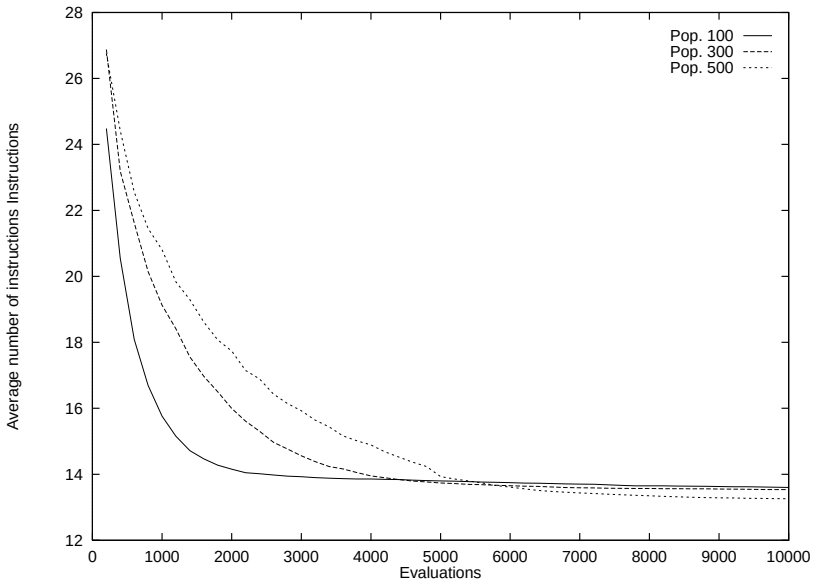


Figure 3.3: Average length of the best individual with the number of evaluations held constant

produce a perfect individual is said to have prematurely converged; such a population does not contain sufficient variety of potential parents to evolve further.

Increasing the population size also increases the number of individuals evaluated if the number of generations is kept constant. Figure 3.3 shows the relative performances when the total number of individuals *evalutated* is kept constant, showing the average length of individuals who can sort 100% of the testcases. The smaller populations tend to converge more quickly, due to the smaller gene pool, and, up until about 5000 evaluations the smallest population performs the best. However, due to the slower convergence of the the larger populations, they tend to produce shorter individuals after 5000 evaluations.

The remainder of this chapter is concerned with the avoidance of premature convergence.

3.5 Maintaining Diversity In Artificial Evolution

Due to their very nature, GAs actively encourage a population to converge on the same fitness level; however, premature convergence does not seem to be a problem for many of the species in nature which have survived to this day, a system which apparently works on the same principles. There are countless life forms on the planet which have evolved

from humble beginnings and show no signs of being trapped on some local minima. If they have become stationary in evolutionary terms it is either because it suits the species at this current time or it forces the species to become extinct.

In natural evolution it is rare, if ever, that a species develops needing to perform only one task; even in the relatively primitive world of plants, a species requires several abilities. It must extract nutrients from soil, its leaves must be efficient at photosynthesis, its flowers must be attractive enough for insects to visit and collect pollen. Few species exist in very varied environments however, and the requirement of several abilities to survive enables plants to fill different environmental niches; depending on the local environment, plants with extremely productive leaves may flourish, or a plant may need to excel at nutrient extraction in order to survive in a different area.

This leads to adaptation as different ecotypes of the same plant or species exploit different niches; they are different in their phenotypic expression, and are similar enough at genotypic level to mate. A result of different ecotypes is the maintenance of diversity in the breeding pool, a prerequisite for avoiding premature convergence.

3.5.1 Sharing And Crowding

A simple method which encourages niche formation is the Crowding scheme. [DeJong, 1975] This scheme uses overlapping generations with new individuals replacing not the individual of the lowest fitness as one might expect, but individuals that are genotypically similar to themselves.

However, to prevent an impractical number of comparisons when an individual is created, newly-born individuals are not compared to every other individual. Instead, a Crowding Factor(CF) is decided upon; that is, the number of individuals which a new individual will be compared to. This has been used with some success by De Jong with a crowding factor of 2 and 3. Crowding was originally designed with multimodal functions in mind, i.e. functions with several peaks in the fitness landscape. It has been found [Goldberg and Richardson, 1987] that crowding does not prevent the population from ending up on one or two peaks within the fitness landscape. Moreover it is not really suitable for putting pressure on a problem that has only one aim in mind.

Another niche method, that of Sharing, was introduced by Goldberg and Richardson [Goldberg and Richardson, 1987]. Sharing is based on the maxim that an environment contains limited resources and that phenotypically similar individuals must share these resources, an idea that seems very reasonable from a biological point of view. Similar individuals suffer penalties to their fitness depending on their similarity to others within the population. Again, Sharing was designed with multimodal functions very much in mind, and, like Crowding above, only attempts to ensure diversity, and not put pressure on a single solution.

3.5.2 Labels

The previous two methods each employ some distance metric to decide what degree of sharing is present. Spears [Spears, 1995] presented a method involving the use of labels on individuals to measure similarity between them. Specifically, every individual has a label, and only mates with an individual who also has the same label. This promotes the likelihood of individuals searching the same area mating with each other, as individuals searching the same area tend to have the same labels.

Spears did not claim that this method would ensure all peaks would be found, rather he suggested that the use of labels could replace distance metrics in crowding and sharing schemes, thus increasing their efficiency. It is possible that individuals with several different labels could end up searching the same peak, but would not mate with each other because of the different labels. Unlike the Sharing and Crowding methods, however, using labels does not require any extra computation.

It would seem then, that using labels gives many of the benefits of Crowding and Sharing, but with none of the associated costs. Indeed, as will be seen in section 3.7.1, the Pygmy Algorithm works in a similar way.

3.5.3 Isolation by Distance

The mating schemes looked at thus far are Panmictic schemes, i.e. any individual may mate with any other in the breeding pool. For some populations, particularly large populations, it is unreasonable to assume that any individual may mate with the individual

of its choice; even the concept of elitism becomes difficult to rationalise, both from a practical point of view and from biological analogy. In the relatively small populations used here, i.e. <=750, it is reasonable to assume the global knowledge necessary for the use of the elitism strategy, but, as populations become larger, >5000, the effort required to maintain this centralised control becomes unwieldy. In addition, large populations require prohibitively long runs, even for a few generations, so are commonly implemented on parallel machines, where centralised control is avoided as much as possible.

To avoid this excessive centralisation, the notion of isolation by distance is used, a notion widely held in natural evolution. There are two widely used techniques for the implementation of Isolation by Distance; Spatial Mating; [Hillis, 1989] [Collins, 1992] [Ryan, 1995b] and the Stepping Stone or Islands model. The Islands model divides the population into several demes, each of which evolves at its own rate and in its own direction with some amount of emigration between islands, while the Spatial Mating model has individuals placed in a toriodal grid and allows mating only with neighbours. This method implicitly creates dynamically sized demes which grow and contract as their fitness varies relative to neighbouring demes.

While both methods, in particular Spatial Mating, have been shown to maintain diversity, they are not very comparable to other methods examined in this chapter, which permit the use of elitism and high selection pressure. For this reason they will not be considered further. There are however, many circumstances in which Spatial Mating is extremely useful, and Chapter 7 presents a simulation which depends on Spatial Mating.

3.5.4 Steady State Genetic Algorithms

SSGA, originally described by Syswerda [Syswerda, 1989], has shown itself to be a good method for the maintenance of diversity. SSGA differs from traditional GA/GP in that in each generation, very few, as few as one, individuals are created, using the current population as potential parents. A new individual created in this way is permitted to remain in the population only if it is fitter than the current worst individual and is unique with respect to all other individuals. The uniqueness requirement involves comparing the individual to all others in the population rather than the 20% required by elitism. 20% is

required because clones only become a problem when they are potential parents.

SSGA can be looked upon as a special case of elitism [Baeck, 1992] with the only difference being the elite group, which numbered 20% of the population in the above experiments, is a much larger proportion of the total population. However, the more extinctive a population is, i.e. the fewer individuals kept from generation to generation, the more directed the population will be.

As this chapter is concerned to a large extent with highly directed populations, SSGA will not be implemented. However, because of its close resemblance to elitism, any strategies which use elitism can also be implemented in SSGA, simply by increasing the number of individuals preserved from generation to generation. Indeed, in subsequent chapters, several of the experiments employ SSGA.

3.5.5 Restricted Mating

An investigation of the gene pool in the runs that failed to produce a perfect individual above would reveal it to be choked with clones - identical individuals. It has been suggested [Eshelman, 1991], and is used as a matter of course by many, that the prevention of clones helps avoid premature convergence in Artificial Evolution. In fact, the first extension to the experiment described above was the disabling of clones, an act which normally requires the comparison of each individual to every other individual in the population as in SSGA.

One of the advantages of elitism over traditional roulette-wheel breeding strategies, like that used in GP, is that it is necessary to ensure only that the breeding pool contains no clones. To ensure this in the current simulation one must compare an individual to 20% of the total population in the worst case, reducing the required computation by a factor of 5. This reduction in computation was a deciding factor in the choice of strategy for selecting a breeding pool. The considerable benefit derived from the disabling of clones can be seen in Figure 3.4.

Figure 3.5 shows the relative performance of the two methods with an identical number of evaluations. Early on in the run there is actually some benefit to allowing clones but, as the population begins to converge and the graph levels off, the method which

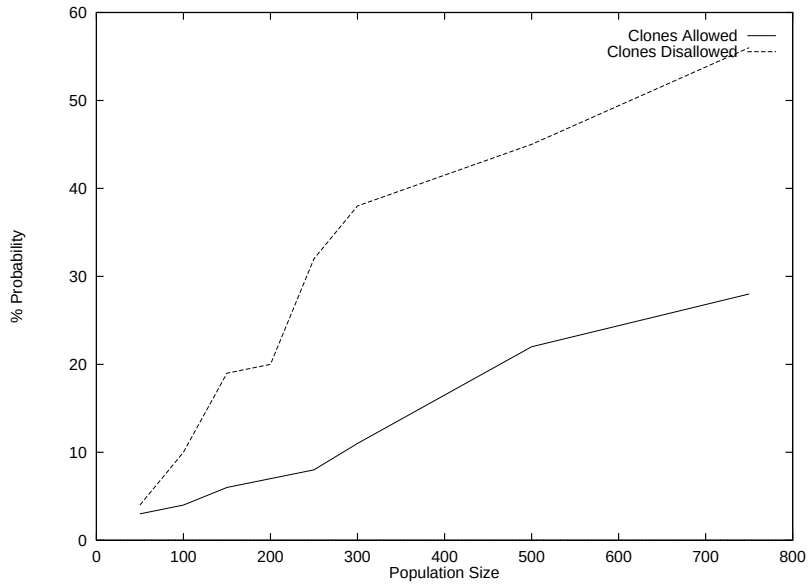


Figure 3.4: The effect on performance of disabling clones

disallows clones begins to outperform the original approach.

Eshelman went further than this and suggested using incest prevention, where only genetically different parents are allowed to mate, to prevent a population from becoming fixated. However, this strategy does have some problems. How different should parents be? How does one decide what is different? Eshelman's strategy is to calculate the Hamming distance between two individuals chosen to become parents. If the difference is above a threshold the individuals are allowed mate. However, as a population converges, parents become more and more alike so it becomes increasingly likely that potential parents will be rejected. If a situation where no parents are allowed to mate arises the threshold mentioned above is relaxed slightly.

In GP, and the high level string GA in this chapter, calculation of Hamming distances is not possible. Any calculation of differences between high level structures would mean choosing in advance the difference between every operator, which would require much prior knowledge of each operator and is to be avoided. It is also quite likely that parents will be of different lengths, leading to some uncertainty as to the classification of difference. One faces a dilemma when comparing two individuals of unequal length, but who would be identical save for the extra instructions in the longer one. Clearly, they are different, but how different?

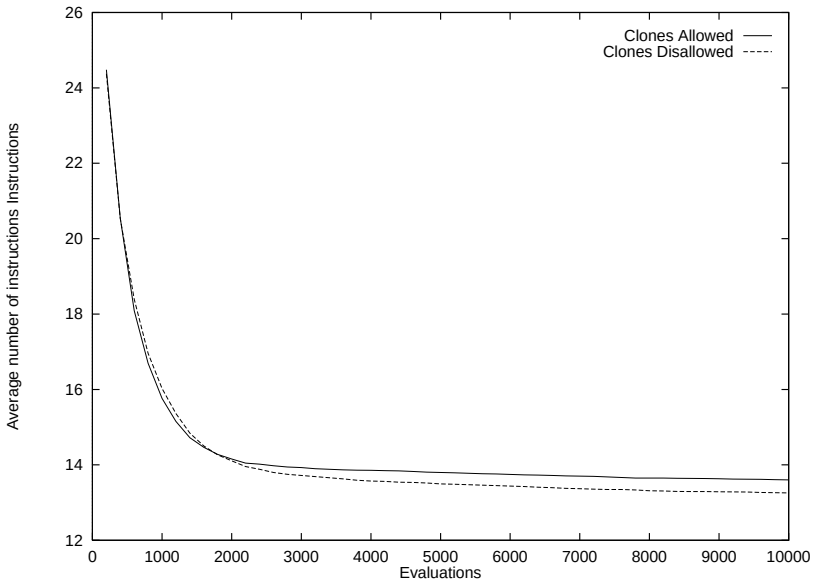


Figure 3.5: Average length of best individual with and without clones disabled

Eshelman's method for incest prevention, although shown to drastically reduce clones at a lower cost than the method for clone prevention described above, aims to prohibit inbreeding to as great an extent as possible. Preventing inbreeding generally improves evolution [Brindle, 1981], but there are times where inbreeding is desirable - because one of its main effects is the exaggeration of certain traits of the parents, not necessarily a bad thing when individuals are being evolved with one aim in mind[1]. This exaggeration could well be considered an advantage in the case where a high selection pressure is being used.

What is needed is a method which does not attempt to explicitly measure genetic differences, for this leads to much difficulty when defining exactly what constitutes difference. Also useful is a method which restricts inbreeding to some extent to keep some degree of diversity in the population; however, one must not lose sight of the fact that the primary purpose of GA and GP is to converge on a solution, not solely to maintain diversity.

3.6 Breeding For Secondary Features

A method which can be implemented cleanly and easily is that suggested by Koza [Koza, 1992] which involves the inclusion in the fitness function of a weight for some sec-

[1] As will be demonstrated in Chapter 4, this is particularly true when individuals are evolving in a specialist niche

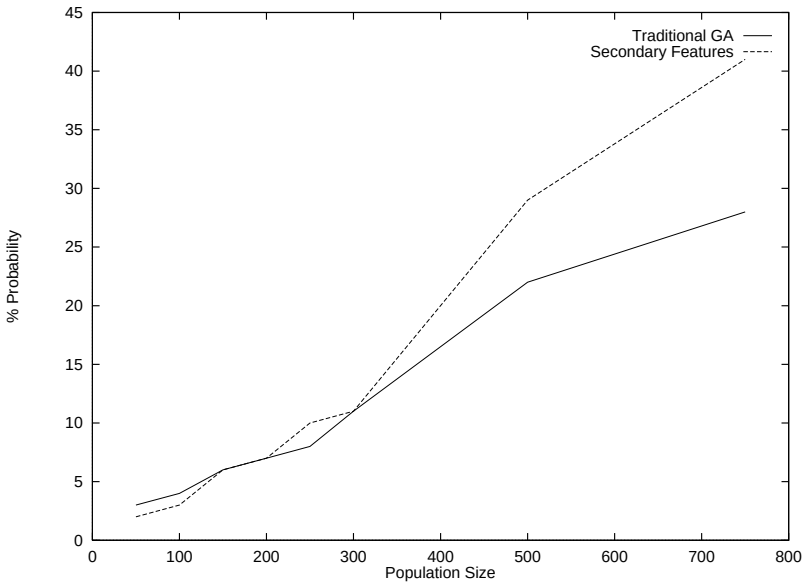


Figure 3.6: A comparison of traditional GA against breeding for secondary features

ondary feature, in this case length. The inclusion of this secondary feature permits the survival not only of efficient(fit) individuals but also that of short individuals. The raw fitness is now modified by the adding of a size factor(sf) similar to that used by Kinnear [Kinnear, 1993] to the fitness function which now becomes

$$reward + ((maxsize - size) * sf)$$

The results of this are now compared to the original scheme, without disabling clones.

Figure 3.6 shows that, at higher population levels, this outperforms the traditional method, and this outperformance is even more impressive when clones are disabled. The cost of disabling clones is identical to the cost in the previous simulation.

This method was not intended to maintain diversity, but to aid the evolution of functions with more than one requirement, and therefore it is not unreasonable to expect that individuals tend to trade off one part of the fitness measurement against the other. For example, the above fitness function gives an efficiency weighting of 90% and weighting of 10% to shortness(parsimony), but in the case where the best-so-far individual has a fitness function with a score of 85 in efficiency and 8 in length, which is easier to evolve, an individual which is more efficient or an individual which is shorter? Of course, one cannot know.

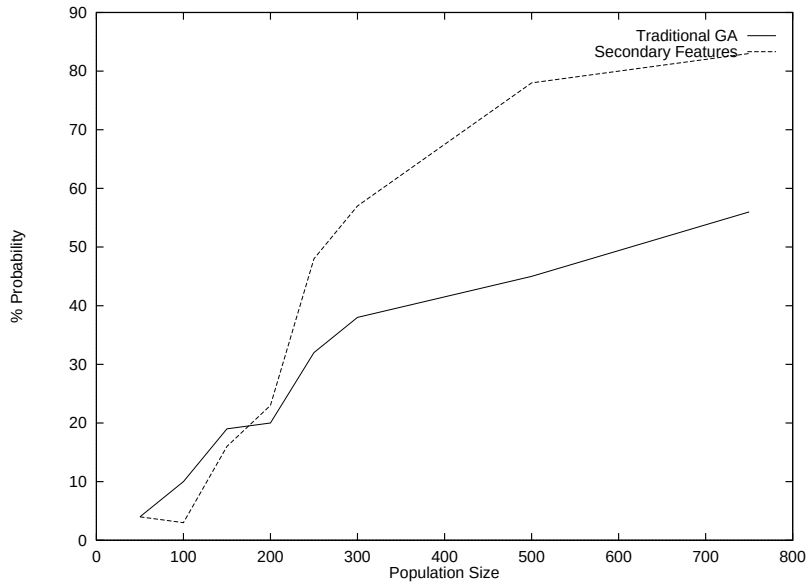


Figure 3.7: The effect on the performances with clones disabled

Neither can one know whether a population would find it easier to lose an instruction that contributes relatively little to the fitness and thus gain a score in the parsimony measure than to utilise an extra instruction which might add to the fitness at a cost to parsimony.

The possibility of "malicious" loss of genetic material gives rise to concern. In certain cases, i.e [Kinnear, 1993], where the size factor is used more as a nudge in a particular direction than a significant portion of the fitness function, the problem does not arise. However, where the size, or indeed any secondary feature, is critical to the performance of any solution, difficulties may arise. It was found that at lower populations, where traditional GA outperformed this method, that genetic material was more prone to loss.

3.6.1 Pareto Optimality

It has been recognised [Horn et al., 1994] [Langdon, 1995] that combining two fitness measures can lead to individuals trading off performance on one measure for another, without actually improving. Horn [Horn et al., 1994] devised a method whereby he combined a standard GA and *Multi-Attribute Utility Analysis* to discover a set of *non-dominated* solutions, i.e. solution that which, although with different scores in each part of the combined fitness function, have the same overall fitness, and then search through these solutions

until a satisfactory one has been found.

Pareto Optimality does not suggest how the overall fitness measurement should be constructed, so in the case where one is not sure of how much each fitness measure should contribute to the overall fitness, it is difficult to construct an accurate Pareto front. Specifically, in this case, we know that the ability to sort is more important than size, but not how much more. Perhaps there are a number of non-dominated solutions, or perhaps there is only a single, dominating solution.

Because of this uncertainty, and the ambiguous nature of many GP experiments, Pareto Optimality will not be considered further.

3.7 Pygmies And Civil Servants

The methods for avoiding premature convergence examined so far fall into two categories. The first is niching, where different individuals tend to congregate together, and the second is the breeding for secondary features described above, where individuals try to maintain a balance between two different goals. The problems with these are the extra computation needed, difficulty in defining difference, the trading off of fitness for length etc. To overcome these problems, a new method, which uses disassortative mating, is now suggested. Disassortative mating, the breeding of phenotypically different parents, is known to occur in some natural populations [Parkins, 1979], and is sometimes used by plant breeders to maintain diversity.

Plant breeders do not have nearly as much information about the genetic make up of their populations as genetic programmers, and it is not uncommon to make decisions based on the appearance of parents rather than to over-analyse their genetic structure. In the world of Artificial Life, many have fallen prey to the problem of trying to over control their populations only to find that the resulting pay off was not worth the effort [Goldberg, 1989b].

The implementation of disassortative mating introduced below requires very little computational effort yet still provides a diverse array of parents. Rather than explicitly select two very different parents, the method presented here merely suggests that the parents it selects are different. No attempt is made to measure how different.

This implementation of disassortative mating, the Pygmy Algorithm, uses a strategy rather like the elitist method, in that a sorted list of the top performing programs is maintained. The Pygmy Algorithm, however, requires the maintenance of two lists of individuals, each with its own fitness function. For the purpose of this chapter, individuals in the first list are referred to as Civil Servants, the fitness function of which is simply their performance(efficiency at sorting). Ties are resolved by rating the shorter individual higher as before. Individuals that do not qualify for the "Civil Service" are given a second chance by a slight modification of their fitness function to include a weighting for length, i.e. the shorter the better. Such an individual will then attempt to join the second list, members of which will be referred to as Pygmies.

When selecting parents for breeding, the Pygmies and Civil Servants are analogous to differing genders. One parent is drawn from the Pygmy list and one from the Civil Servant list, with the intention of a child receiving the good attributes of each and resulting in a short, efficient program. The presence of the Civil Servants ensure that no useful genetic material will be lost, and the presence of the Pygmies increase the pressure for the shortening of programs. The difference between genders in the Pygmy Algorithm and those in nature is that the gender of an individual is not adopted until after the fitness function(s) are evaluated; whereas in nature, individuals tend to be born with a particular gender.

By analysing the data during runs it was found that each group influenced the new members of the other. Pygmies ensured that new Civil Servants became progressively shorter while the Civil Servants maintained a relatively high efficiency among the Pygmies. While the length of the Civil Servants never became shorter than the optimum length, the length of the Pygmies was frequently shorter as they tried to trade off efficiency for length.

The ancestors of this method are plain to see, for its roots are firmly entrenched in niche and speciation; elitism with its powerful selection pressure is also involved as is the strategy of breeding for secondary features.

It must be stressed that no effort is made to calculate how different Civil Servants are from their short cousins, the fitness function automatically decides which type an individual is. When selecting for breeding it is possible that two close relatives, even siblings, may be chosen; but this will only ever happen when they have complementary features, i.e. efficiency and shortness.

3.7.1 Labels revisited

The use of Labels, described in section 3.5.2 bears a resemblance to the Pygmy Algorithm, i.e. each group could be considered to have a different label. The crucial difference, however, is the individuals in the Pygmy Algorithm are not born with a predefined label, rather, it is given to them depending on their performance. Moreover, it is possible that an individual may not even receive a "label" in the Pygmy sense, for if it is not good enough to get into either list, it is simply discarded.

3.7.2 Implementation

To give a fair comparison, the Pygmies and Civil Servants were split evenly among 20% of the population. In the previous experiments an elite group of 20% was maintained. The fitness of the Civil Servants is simply their efficiency, while the fitness of the Pygmies is much like that above

$$reward + ((maxsize - size) * sf)$$

but with a higher size factor, in this case 20% of the total fitness.

When running the Pygmy implementation it was found that the Pygmies tended to be smaller by several instructions and about 90-95% efficient. Soon into each run, the Civil Servants were all 100% efficient, but somewhat taller than the Pygmies. As each population progressed, the size of the Civil Servants followed that of the Pygmies, the size of which, in turn, decreased as that of the Civil Servants approached.

As can be seen from the graph of performance, the Pygmy method consistently outperforms each of the other two methods. It was expected that under easier conditions, i.e. higher populations, that the breeding for secondary features strategy would approach the Pygmy method, but it never outperformed it. It was mentioned previously that the secondary features strategy makes the possibility of losing genetic material more likely, and it was found that the more difficult the problem was, the higher the probability of this happening was and the worse the relative performance of this strategy became.

As described in Section 3.3.2, the problem was scaled down to obtain a large quantity of results, and populations were similarly scaled down to increase the difficulty of a problem. The size of a population is directly related to the diversity of possible solutions

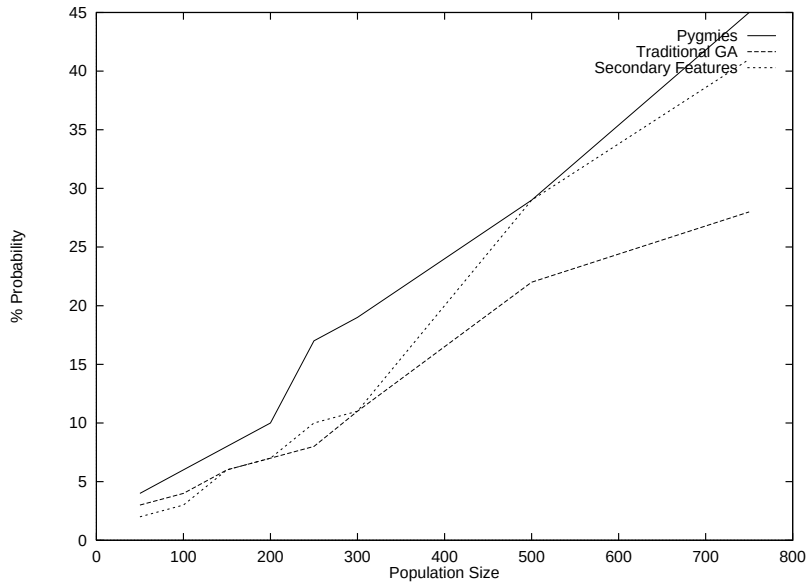


Figure 3.8: A comparison of the three methods with clones allowed

within it and it is significant that the relative performance of the Pygmy Algorithm is best under the more difficult conditions.

3.7.3 Extending the model

It was pointed out earlier, in section 3.5.5, and was a focus of much of Eshelman's work, that while the disabling of clones maintains diversity to a great degree, there is much effort involved, an effort greatly reduced by the use of elitism. The use of the Pygmy model effectively halves the effort required by elitism : no Pygmy will ever be a clone of a Civil Servant, and vice versa.

When clones are disallowed, the results are as in Figure 3.9.

From these results it appears that the Pygmy algorithm and the strategy of breeding for secondary features have an almost identical performance, except that at higher populations the Pygmy algorithm fares slightly worse. However, as mentioned above, the disallowing of clones in the Pygmy algorithm is done at half the cost.

3.8 Gender or Race?

Due to the fact that all the individuals were looked upon as belonging to one of two genders, inbreeding between individuals with similar performance was prevented. It

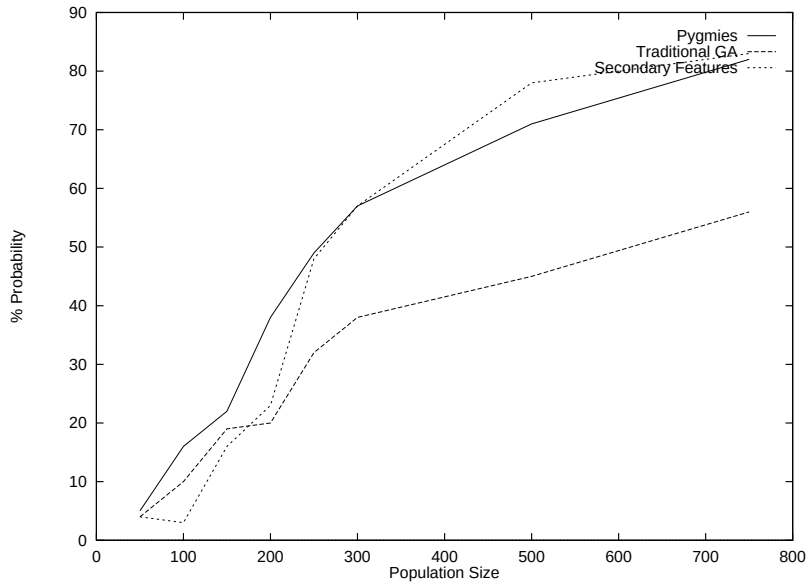


Figure 3.9: A comparison of the three methods with clones disabled

was still possible, however, for individuals to breed with close relatives - siblings perhaps, or even a parent of the opposite sex. In the Pygmy Algorithm, such incestuous behaviour did not appear to cause problems because of the diversity of parents maintained in each of the two lists.

The nature of multi-objective problems is such that individuals must solve two or more smaller problems in order to solve the main problem. However, despite the fact that the Pygmy Algorithm maintained groups of individuals who were good at each of the sub-problems, neither group explicitly attempted to produce individuals who excelled at its own sub-problem, simply because individuals in the same group could not mate with each other, a situation which is the exact opposite of the Labels method.

One of the motives behind this work is to investigate whether or not it would be better to try to solve the main problem together with each sub-problem in parallel. The only way a GA can solve a problem is through evolution, so individuals in the same lists, up to now physically unable to do so, would have to be permitted to mate. For this reason, individuals are no longer of a given gender, but are assigned a *race*, which allows every individual to mate with every other, but still makes an individual's membership of a list readily identifiable.

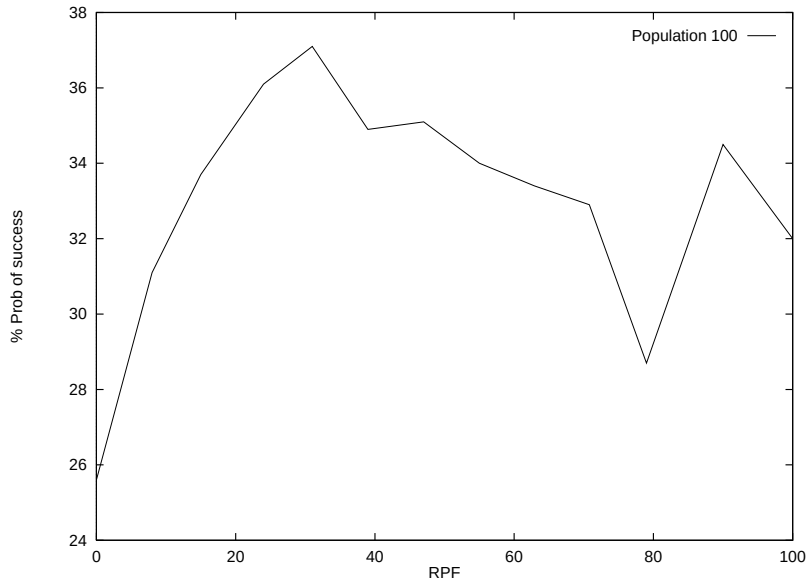


Figure 3.10: Varying the RPF with a population of 100

3.8.1 Racial Preference Factor

If the original Pygmy Algorithm is taken to be using races, then each race would *always* outbreed. The descriptive name chosen for an individual's tendency to outbreed is *Racial Preference Factor* (R.P.F.) - which is simply a measure of the probability that an individual will choose an individual from the *other* race when selecting a mate. In the case of the original Pygmy Algorithm, individuals display behaviour characteristic of having an RPF of 100%, i.e. always outbreed.

Initial experiments were designed to investigate whether or not it was worthwhile using RPFs of different values - which would permit inbreeding within each race to a certain degree. As there was no way of knowing in advance which value of RPF (if any) would be the optimal, several experiments varying its value were carried out. The RPF was varied from 0% (always inbreed) to 100% (always outbreed), and yielded the results as shown in figure 3.10.

Although experiments for *each* value of RPF were repeated on 3000 different initial populations - the same 3000 for each value to aid comparison - there was no one value for the RPF which was obviously better than the rest. However, an important result was that all of the higher results were in the range 20% to 40%, which shows that always outbreeding in this case was not the optimal strategy, and that inbreeding to quite a significant extent

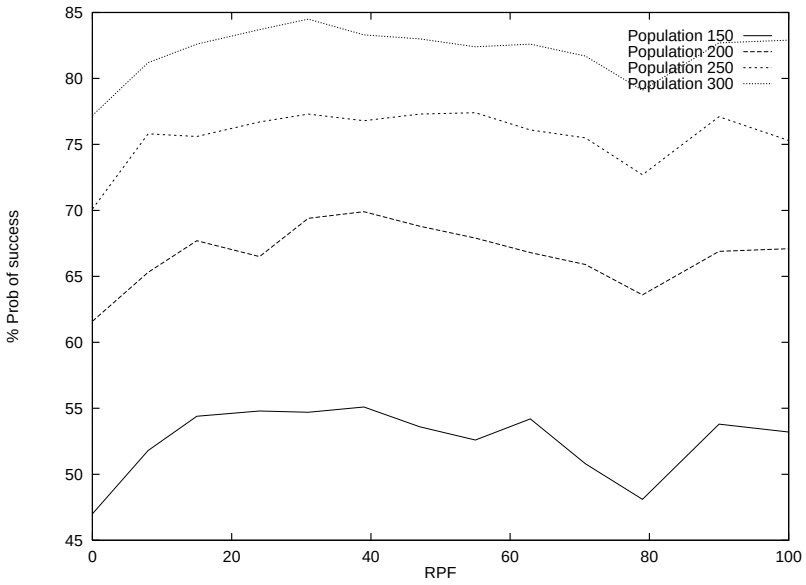


Figure 3.11: Varying the population over a number of RPF values

improves performance. To what extent is, at this stage, still unclear.

To further test the result of there being no optimal value for the RPF, the same experiments were carried out on a number of other population sizes, ranging from 150 to 300 individuals, and yielded the results as in figure 3.11.

These experiments served to add to the confusion over the value for the RPF: not only were there several different possible optimal values, but these also changed when the population size changed. The only consistent results were that the approach of always inbreeding, not surprisingly, yielded poor results, and that a RPF of around 80% also tended to produce poor results.

The conclusion can only be that there is no single optimal value for the RPF, rather that it is dependent on the initial state of the population. It is certain that an RPF of somewhere between 15% and 75% would be best, but such vagueness would be useless when using the Pygmy Algorithm on different problems - how could one choose the value of RPF in advance? If there is some uncertainty about the optimal value, as there is here, which value should one choose?

3.9 Tuning the RPF

The conclusions of the previous section show how dependent on the initial state of a population the RPF is, as is any parameter for controlling evolution. An ideal situation would be to choose a separate value RPF for each population, which would be tailored specifically for that population.

A similar view was taken by [Baeck, 1992] when trying to select an optimal value for mutation rate for a population. Trying to find an optimal value for mutation rate [DeJong, 1975], [Grefenstette, 1986], [Schaffer, 1989] yielded much the same conclusions as the early experiments in this chapter on RPF. The optimal value of mutation rate varies from problem to problem, and even from population to population within a single problem. The approach taken was to incorporate mutation rate as part of an individual's genes and allow it to *evolve* as the population evolved. Like other genes, an individual's mutation rate could be subjected to crossover and mutation. This strategy meant that each individual had its own mutation rate which it would examine when testing to see if mutation was to be performed. Baeck found that this improved the performance of his GAs over those GAs for which he arbitrarily selected a value for mutation rate.

Applying this strategy to RPF, each individual was assigned its own personal RPF which reflected its attitude to outbreeding. This attitude was shaped by the experience of its parents and ancestors - an individual who was the product of outbreeding would be more inclined to outbreed, reasoning that if it worked for its parents, then it should help it produce children with good performance.

3.9.1 Meta-GA

Three approaches were taken to the tuning of RPF. All three involved individuals having their own RPF which could evolve in the same manner as any other gene. The three approaches are as follows :

- **Species Average (SA)**
- **Racial Average (RA)**
- **Individual Average (IA)**

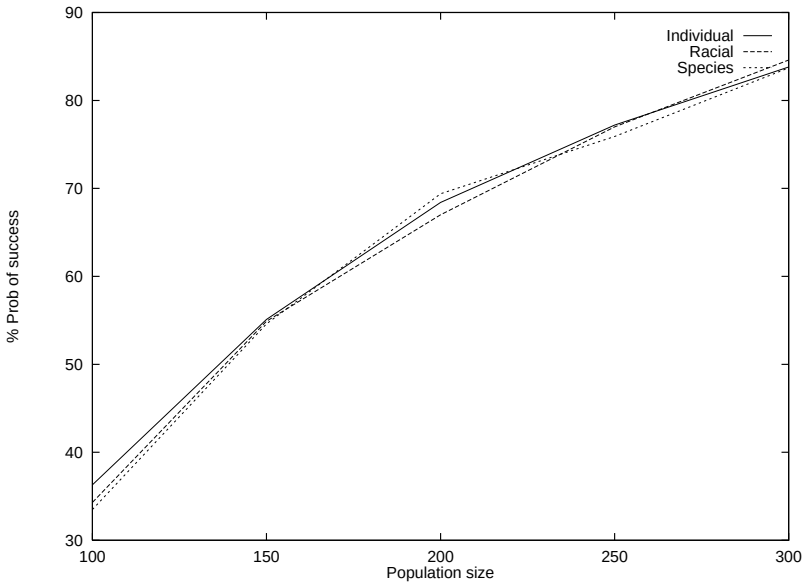


Figure 3.12: Evolving the RPF

The SA experiments maintained a single value for RPF which was simply the average of the entire species. While this does incorporate the overhead of calculating the average of the parent population - 20% of the entire population - it does have the advantage of allowing one to keep track of the effective RPF as the population evolves.

The RA experiments maintained *two* values for RPF - one for each race, with each RPF being the average of that race. As a run progressed, the two values deviated considerably from each other and even changed at different rates, showing that, depending on the current state of the population, different amounts of inbreeding and outbreeding suited each race.

Finally, the IA experiments maintained a separate RPF value for each individual, and individuals did not consult or examine the RPF of other individuals when choosing a mate. This approach is the closest to that of [Baeck, 1992] but had the slight disadvantage of making it impossible to figure out what exactly was happening to the value of the RPF.

To maintain a balance, half of the parents were chosen from each race, and these parents then chose from which race they wanted their mate.

The results are shown in figure 3.12.

Despite the advantages of being able to track the effective value of RPF in both the SA and RA experiments, the IA yielded the best results, and at the smallest computational

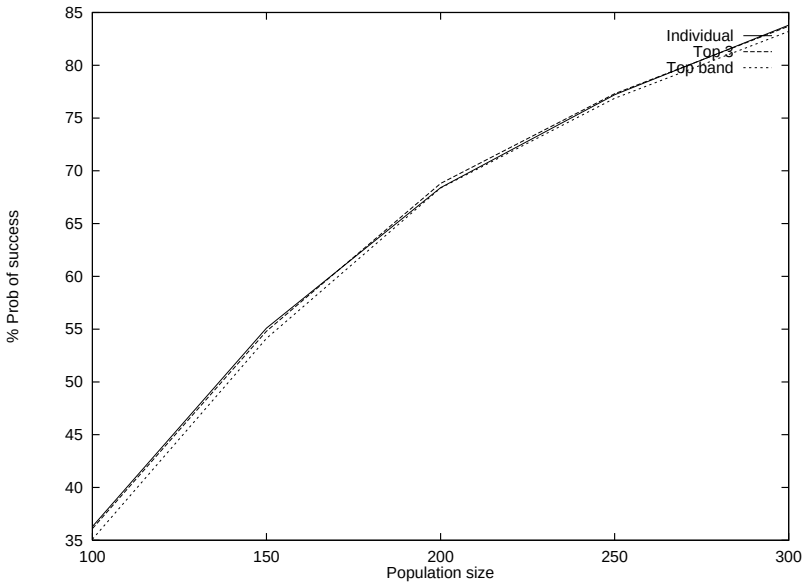


Figure 3.13: A comparison of evolving RPF and fixed RPF

cost. However, there is such a small difference in the performance of each approach, it appears that a meta-GA is robust enough to evolve the parameters, regardless of how it is implemented.

Figure 3.13 shows the best of the evolving RPF experiments compared to two of the fixed RPF results, against an average of the top three results for each population and against the top band for each population, typically in the region 15% to 75%. Although the results appear practically identical from the graph, the evolving RPF, henceforth known simply as "IA", slightly outperformed all of the fixed experiments, with the bonus that using IA did not involve many runs to try and find the optimal value for RPF. The IA experiments did not perform better than the best result found by brute force, but as the values for brute force involved some 39,000 experiments it was felt that using an average of the top results gave a fair enough impression.

3.9.2 Sociological Modelling

So far, as in all previous implementations of meta-GA, the RPF of an individual is looked upon solely as being a genetic feature. However, because of nature of the current model, which is modelling the behaviour of individuals in races, it was decided that treating the RPF as an *attitude* rather than simply as a phenotypic trait would be more appropriate.

As well as using RPF, individuals used a variety of methods for calculating their own RPF. Some individuals were incapable of making up their own minds and simply followed the prevailing opinion, the same as SA above, while others were a bit more tribal in their attitudes, following the general opinion of their race, in the same manner as RA. A final, independently-minded group was the same as IA, in that the members made up their own minds when deciding their RPF.

Taking RPF to be an attitude loses none of the ability to perform the experiments outlined above, but, like any opinion, RPF can be influenced or swayed by other opinions, and this observation led to another suite of experiments.

- Free Choice Model(FCM).
- Independence from Prevailing Opinion.
- Influential Partners
- Opinion reinforcement.

The first set of experiments, the free choice model, influenced by [Todd and Miller, 1991], allowed individuals the choice of whether or not to accept another individual as a mate. This was implemented as below:

1. Select first parent from one race.
2. Select, according to first parent's RPF, which race to choose a mate from.
3. Select individual (the second parent) probabilistically from that race.
4. Test, according to the second individual's RPF, if it wants to mate with an individual from the first race.
5. If the overtures of the first individual are accepted then mate, otherwise select another second parent.

If, after nine attempts, an individual cannot persuade any others to mate with him, he is deemed too unattractive and is rejected.

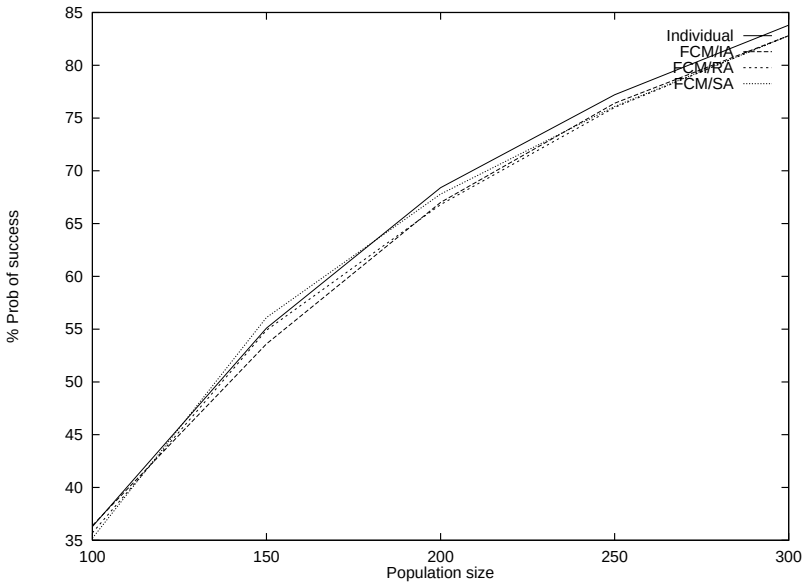


Figure 3.14: A comparison of the Free Choice Model and IA

Figure 3.14 shows that allowing individuals free choice of whether or not to mate with a potential suitor didn't give any improvement over the original IA experiments. Several other experiments were tried, varying from allowing individuals of type RA and SA some degree of independence from the racial or species average, to permitting individuals to influence each other to some extent. As the most interesting results have all come from the IA type experiments, only the extensions to these will be discussed.

3.9.3 Influential Partners

Like any opinion, RPF can also be subject to change. In this section, individuals were influenced by (potential) partners, which allowed their RPF to vary depending on that of those around them. The implementation was as follows:

1. Select *father* from one race.
2. Select, according to father's RPF, which race to choose a mate from.
3. If father's RPF>mother's RPF, then let her effective RPF be the geometric mean of the two.
4. Test, according to mother's *effective* RPF, if she wants to mate with an individual from the father's race.

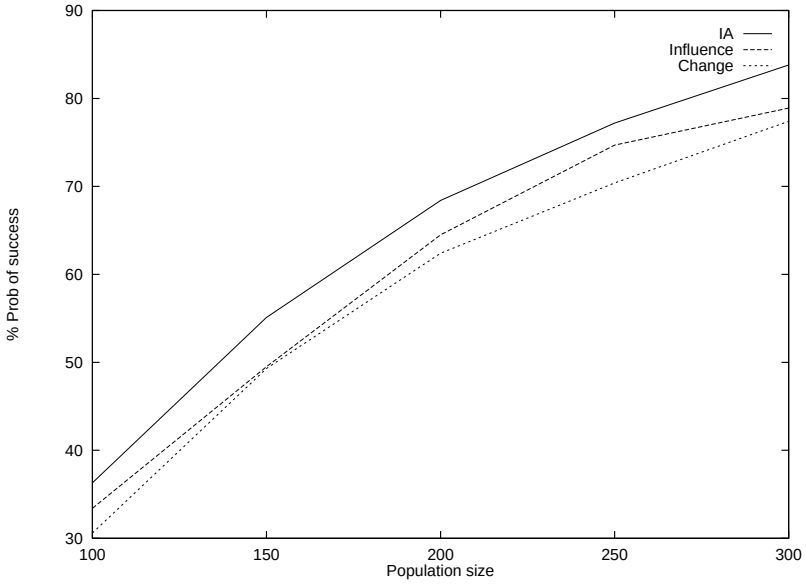


Figure 3.15: A comparison of the influential models and IA

5. If she accepts the father's overtures, then mate, otherwise select another mother.

The reasoning behind this experiment is rooted more in sociological than biological thought[Todd and Miller, 1991]. If the probability of the father choosing the mother is greater than the probability of her accepting his advances, then her RPF is adjusted upwards - reflecting the influence his enthusiasm has on her. Two different versions of this experiment were run. In the first, the mother's RPF returned to its initial value after each mating, whereas in the second, the value of her RPF remained at the new value, reflecting a situation where a mate had a lasting effect on her.

The rather easily led individuals in this experiment did not fare too well relative to the IA experiments, as can be seen in Figure 3.15, and resulted in lower performance under every circumstance. Relying on other individuals for information about how to behave towards other races would not appear to aid the population as much as a simple meta-GA.

3.9.4 Opinion Reinforcement

The final simulation concentrated solely on the IA experiment, as this had yielded the best results so far. Again, individuals were allowed to change their RPF during their lifetime, and any changes were permanent. All changes were based on their *own* experiences and, based on the results of the experiments in which individuals were influenced by each

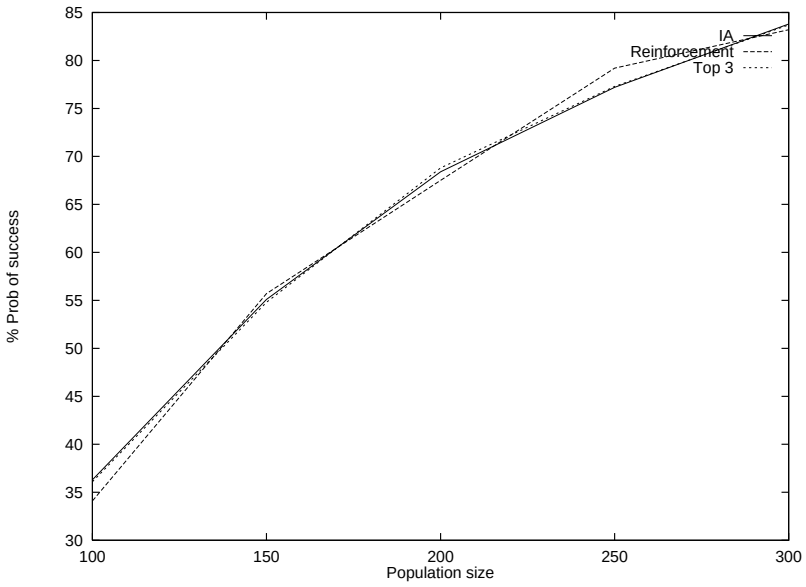


Figure 3.16: A comparison of the Reinforcement Model, IA and the top 3.

other, did not concern themselves with the opinions of anybody else. This was implemented as follows:

1. Select father from one race.
2. Select, according to father's RPF, which race to choose a mate from.
3. Produce child.
4. Test child.
5. If child is fit enough to enter parent population, adjust father's RPF so he is more likely to make the same decision the next time he mates, otherwise adjust the RPF so the father is less likely to make the decision.

Of all the experiments which exploited the fact that RPF was an opinion, the Opinion Reinforcement fared best, giving the same performance as IA. This suggests that allowing individuals to make up their own minds, whether through evolution or from personal experience, may lead to better performance than either forcing a value on them, or by letting them influence each other.

3.10 Conclusion

A new selection method, the Pygmy Algorithm, for use with multi-objective functions has been introduced. Unlike previous models, the Pygmy Algorithm does not depend on the implementor being able to measure exactly the relationship between each objective. It was also shown that the Pygmy Algorithm can be used to prevent clones at half the cost of normal methods.

Also considered was the use of sociological modelling, where different types of individuals were looked upon as being races. In a curious parallel with human societies, experimental results showed that situations where individuals are allowed make up their own minds and judge from their own experiences produce better performing populations than those where individuals are easily led, and give more weight to the prevailing opinion of their society than to their own feelings.

This chapter also served to show the power of a meta-GA, the use of a Genetic Algorithm to evolve the parameters that are controlling it. A simple meta-GA performed almost as well as the brute force methods, but is much more general.

Chapter 4

The Races Genetic Algorithm

4.1 Introduction

This chapter introduces a modified Genetic Algorithm, *RGA* which extends the notion of races as described in the previous chapter to use more than two races. RGA is tested on a number of multi-modal functions and is shown to outperform a number of current methods, both in discovering solutions and maintaining them on a set of one and two dimensional problems.

4.2 The Problem Space - Multi-modal Functions

Multi-modal functions differ from multi-objective functions in that there are a number of different solutions, *all* of which must be discovered and maintained. Many problems of interest, particularly real world problems, contain several peaks in the fitness landscape, as in Figure 4.1. Sometimes, only one of these peaks will be of interest, but often [Spears, 1995] [Deb, 1989] [Beasley, 1993] it is desirable to find all peaks of the landscape.

Unlike problems in which the peak of interest changes with time[1], [Ryan, 1994a] [Hillis, 1989] [Siegel, 1994], it is important to maintain solutions at all peaks in the landscape, ensuring that one peak does not dominate and ensuring that all peaks are actually discovered by a population. Traditional GAs find locating all peaks in the landscape a difficult problem[Goldberg and Richardson, 1987] [Goldberg, 1992] and several solutions which

[1] Problems where the location of the peak of interest does vary with time are considered in Chapter 6

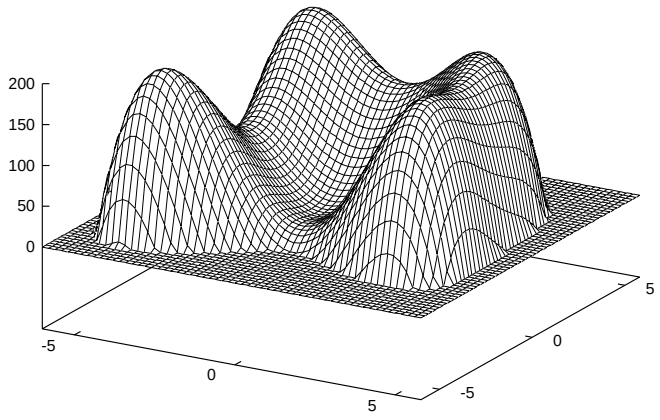


Figure 4.1: Example Multimodal Function.

are reviewed below have been suggested.

This chapter presents a modified version of the Pygmy Algorithm discussed in the previous chapter, which results in a new method for the maintenance of several solutions to a multimodal function in a single population - the Races Genetic Algorithm(RGA). The tests on the RGA algorithm are those originally used by Deb [Deb, 1989] and since adopted by Beasley and Spears as the standard test bank for assessing the proficiency of algorithms in multimodal function optimization.

Like Beasley, the measurements of performance used is the number of peaks located in the fitness landscape that exceed a certain *threshold*. Beasley described such peaks as *maxima of interest*. Much of the earlier work on multimodal functions [Deb, 1989] [Deb and Goldberg, 1989] [Goldberg and Richardson, 1987] assumed that (a) the number of maxima of interest in a function is either known or can be quite accurately estimated, and (b), these maxima are spread reasonably evenly throughout the search space. However, RGA does *not* require any such assumptions and can be applied with little or no knowledge of the type of maxima of interest present in the search space.

4.3 Previous work

Several of the methods described in the previous chapter, e.g. Crowding and Sharing etc. were designed primarily with multimodal functions in mind. However, a new method [Beasley, 1993], which will be described in Section 4.3.3, has been shown to outperform these methods. For this reason, RGA will be compared only to this, newer work.

4.3.1 Labels

The use of labels was discussed briefly in Chapter 3, where it appeared that they would have the opposite of the desired effect. Spears did not claim that this method would ensure discovering all peaks of a multimodal function, but suggested that the use of labels could replace distance metrics in crowding and sharing schemes, thus increasing their efficiency. It is possible that individuals with several different labels could end up searching the same peak, but would not mate with each other because of the different labels. Unlike the sharing and crowding methods, however, using labels does not require the extra assumptions described in Section 4.2.

Spears' use of labels is discussed further in section 4.4.1.

4.3.2 Parallel sub-populations

Another technique, reported on by Beasley [Beasley, 1993], is to divide a population into various sub-populations and evolve them in parallel. There is, of course, no guarantee that each sub-population will investigate a different maximum of interest, nor is there any reason why each sub-population shouldn't investigate the *same* maxima, a situation likely when all maxima are not of the same height, which results in some maxima having a greater fitness associated with them.

4.3.3 Iteration

Running sub-populations in parallel, particularly when there is no communication between them, is tantamount to running a population several times in the hope that each, or at least some, of the runs find different peaks. Again, there is no reason why each run

subsequent to the first should find a different peak, and, particularly in functions with uneven peaks, it would be expected that all runs would tend to find the same peak.

Beasley [Beasley, 1993] used *derating functions* to address the problem of all runs searching the same area. Derating functions are used to modify the fitness function from run to run so that areas already searched will not be searched again, so knowledge attained in one run can be used in subsequent runs. The basic algorithm used by [Beasley, 1993] is as follows:

1. Initialise: equate *modified fitness function* with raw fitness function.
2. Run the GA using the modified fitness function and keep a record of the best individual found in the run.
3. Update the modified fitness function to give a depression in the region near the best individual, producing a new modified fitness function.
4. If the raw fitness of the best individual exceeds the solution threshold, display this as a solution.
5. If all solutions have not been found, return to step 2.

Of course, the difficulty arises in knowing when all solutions have been found. Beasley [Beasley, 1993], however, produced some impressive results using the same test bed of functions as this chapter. These results are discussed further in Section 6.

Beasley measured the "region near the best individual" using a distance metric based on the decoded parameter space, which, as individuals decode to a real number, was trivial to measure.

4.4 Races in multimodal functions

Optimizing multimodal functions requires a somewhat different approach from multi-objective problems in that the most important goal is the discovery of a number of solutions, whereas multi-objective problems are more concerned with the balancing of a number of goals.

However, if one could have fitness functions that would evaluate an individual's suitability for each peak, using a fitness function designed specifically for that peak, then a population would be strongly encouraged to be spread evenly across the search space. Each fitness function would allow convergence on its respective peak. Unfortunately, such a situation is unlikely in the real world, as one simply doesn't know where the peaks are located in the search space, and indeed, how many peaks are present.

If one cannot use information about the peaks themselves, then the only other information available concerns the individuals. In function optimization, individuals are typically real numbers, so calculating the difference between them is a simple matter. Indeed, it was by a method such as this that Goldberg decided if individuals were close enough that they would be sharing fitness.

By using fitness functions that reward individuals simply because they belong to a particular race, a population can be forced to search many points of the search space at a time.

Combining each of these fitness functions with the multimodal function to be optimized, the problem now effectively becomes a multi-objective one. As well as having to find a peak in the search space, an individual is also expected to belong to a race. Far from complicating the search however, this improves it vastly, as the races promote a wide variety of individuals that can concentrate on different parts of the search. The fitness function for each race is of the form:

$$FitnessFunction_x = Distance(x) + Weight.Function(individual)$$

Where $Distance(x)$ is a measure of how close an individual is to the racial perfect, $Weight$ is a weight given to an individual's performance in the function to be optimized, and $Function(individual)$ is the individual's score in the function.

The racial perfect is decided by dividing the search space into a number of equal parts, one for each race. The racial perfect is then an individual that matches the division for a particular race. The distance of other individuals from the racial perfect is calculated in the same way as Beasley's Sequential Niche Technique [Beasley, 1993], which is simply the absolute difference between the decoded individual and the racial perfect.

If, for example, the search space was 0..1, and it was divided into five races, the

Race	Racial Perfect
1	0
2	0.25
3	0.5
4	0.75
5	1

Table 4.1: The search space 0..1 spread across five races

Race	Racial Perfect	0.33	0.755	0.55
1	0	0.33	0.755	0.55
2	0.25	0.08	0.505	0.3
3	0.5	0.17	0.255	0.05
4	0.75	0.42	0.005	0.2
5	1	0.67	0.245	0.45

Table 4.2: Scoring individuals for their suitability to five different races.

racial perfects as outlined in table 1 would appear. The distance of three decoded individuals is also shown to illustrate how simple it is to calculate how suitable an individual is to a race.

Three example individuals, who's phenotypes are $0.33, 0.755$ and 0.55 would then score as in table 4.2.

The individual that decodes to 0.33 is most likely to join the second race having a racial perfect of 0.25. But, that same individual is also more likely to join the first or the third race than the individual that decodes to 0.755 by virtue of the fact that it has a smaller distance from the racial perfect.

The weight given to the performance of the individual at function optimization is the same for each race. Like most GA parameters, there is no absolute rule for the setting of the weight, so a certain amount of trial and error is involved in the choosing of a weight that ensures the population does make some effort at optimizing the function.

4.4.1 Tracing the ancestry

The question now arises of how to decide to which race an individual belongs. Spears[Spears, 1995] explained his use of tags with the description of how one decides whether or not they are Portuguese. A person does not decide they are Portuguese simply

because they resemble their relations, but because their ancestors were identified as being Portuguese, and they simply "culturally inherited" [Spears, 1995] the label.

The RGA uses this reasoning to a certain degree, but does not adhere to it as strictly as Spears did. Using the previous analogy of a person trying to decide whether or not they are Portuguese, we can explain how it works. In the event of both a person's parents being Portuguese, there is little doubt that that person will also be Portuguese. Indeed, in Spears' model, mating was restricted to individuals with common tags, so this would always be the case. But, if a Portuguese man was to mate with, say, an Irish woman, what then would the offspring be, Irish or Portuguese? The simplest answer is a bit of both, although depending on the upbringing, the offspring may tend to consider themselves one or the other. Because RGA does not insist on inbreeding (for reasons which will be made clear in the following section) the situation of inter-racial mating is only to be expected.

To overcome the question of deciding which race an individual belongs to, let us return to question of nationality, the half-Portuguese, half-Irish offspring who is trying to discover his true nationality. Let us assume that he travels to Portugal and every other Portuguese he meets accepts him as being Portuguese. If this is the case, then it is fair to say that he is Portuguese. He might also decide to visit Ireland, and every Irish person he meets accepts him as Irish. What does this mean? Is he Irish or Portuguese? The only answer can be is that he is both.

On the other hand, the tale might not end with such acceptance, and both Irish and Portuguese decide that he is not one of them, so he is not considered to be a member of either race.

This analogy could be taken much further. His Irish mother could have been the offspring of an Irishman and Englishwoman, so perhaps he might belong to the English race also. The moral of this story is that when races interbreed, there is no way to know which race(s), if any, the offspring will belong to. RGA overcomes this uncertainty by having an individual examine *each* race, and see if the other members of the race accept it as being a member. Because it is crucial to maintain the "cultural identity" of each race, a Steady State approach is taken, whereby only one individual is created at a time, evaluated, and tested to see if it can get into the parent population.

The term "nationality" could also be used here, but "nationality" and "race" are

considered to be the same in this context. The term "race" is used because it has more genetic connotations, and in RGA it is an individual's genes rather than geography that determines their race.

The implementation, then, of RGA is as follows:

1. Create and test individual.
2. Test to see which, if any, races the new individual can join.
3. If initial population(*Generation 0*) has been created go to step 4, otherwise, repeat steps 1 through 3.
4. Select race at random.
5. Select individual probabilistically from race.
6. Examine if individual wishes to inbreed or outbreed.
7. Select second individual from appropriate race probabilistically.
8. Create and test new individual.
9. Test to see which, if any, races the new individual can join.
10. If finishing criteria haven't been fulfilled, go to step 4.

Ideally, as many races would be created as there are peaks in the landscape, and each race would evolve towards one peak. But one cannot assume that enough information is available to do this, so the safest thing to do is to create at least twice as many races as one *thinks* there are peaks. "Thinks" is emphasised because the exact number of peaks does not matter to RGA, and creating extra races will not adversely affect performance.

4.4.2 The benefits of inter-racial mating

Practically every other application of GAs to the problem of solving multimodal functions has involved the use of restricted mating in some sense, usually restricting individuals to mating with individuals on the same peak in the landscape. But individuals occupying the same peak in the function landscape may be spread across several nearby races, as illustrated by figure 4.2, so this does not suit RGA.

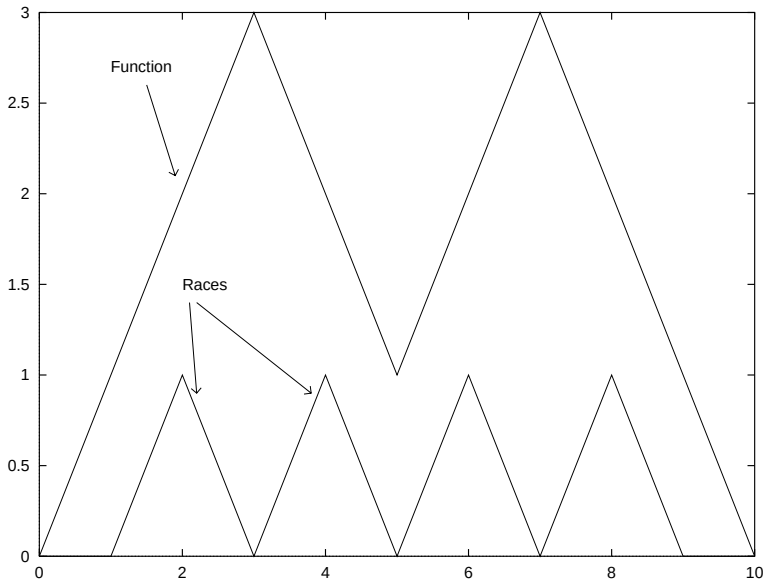


Figure 4.2: A case where four races are evolving toward two peaks. Neither peak coincides with a race, but both appear between two.

4.4.3 The cost of inter-racial mating

All algorithms considered in this paper are reported upon in terms of their quality at locating and maintaining solutions, not in terms of computational cost. Using races is akin to steady state genetic algorithms [Syswerda, 1989] except that instead of having one sorted list of N individuals, one maintains M sorted lists of $\frac{N}{M}$ individuals. In this respect, one is attaining a large increase in performance, as outlined below in Section 6, at a cost marginally greater than that of using a steady state genetic algorithm.

In figure 4.2 there are many more races than peaks in the function, so several races will evolve towards each peak. Inter-racial mating is useful because otherwise each race would search in isolation for the peak, and may not contain sufficient information to converge on it. But, as will be demonstrated in section 7.6.5, as the difficulty of a problem increases, inter-racial mating becomes more important.

4.5 Test functions

As stated in section 4.2, the tests introduced by Deb will be used to test RGA. Both Deb [Deb, 1989] and Spears [Spears, 1995] considered how long their algorithms could

maintain a stable population spread across each peak as a measure of success, while Beasley measured the probability that his algorithm would actually find all of the peaks. Due to the steady state nature of RGA, once all peaks are found, maintaining stable sub-populations on each is a trivial task, and peaks will never be lost. For this reason, the same measures used by Beasley will be employed. That is, *success rate*, which is the probability that all peaks of a particular function will be found, and *Root Mean Square error* (RMS). RMS is a measure of the accuracy of solutions found. The value for RMS is calculated by finding the distance between each solution found and the exact solution. This distance is then squared and the RMS value is the square root of the mean of these values.

All of the test functions were described by Deb to test his sharing function. In functions F1 to F4 a 30-bit binary chromosome was used to represent x in the range 0 to 1. In function F5 there are two parameters, x and y, which are represented by two 15-bit numbers in the range $-6 \leq x, y \leq +6$.

4.5.1 F1 - equal peaks

The first function has five equally spaced peaks of equal height:

$$F1(x) = sin^6(5\pi x)$$

The peaks are at x values of 0.1, 0.3, 0.5, 0.7 and 0.9.

4.5.2 F2 - decreasing peaks

Deb's second function has peaks at the same locations as F1, but with heights decreasing exponentially:

$$F2(x) = exp\left(-2log(2) * \left(\frac{x - 0.1}{0.8}\right)^2\right) * sin^6(5\pi x)$$

In this case, the peak heights vary from 1.0 to .25.

4.5.3 F3 - uneven peaks

The third function described by Deb has peaks of equal values, but at uneven intervals:

$$F3(x) = sin^6(5\pi(x^{\frac{3}{4}} - 0.05)$$

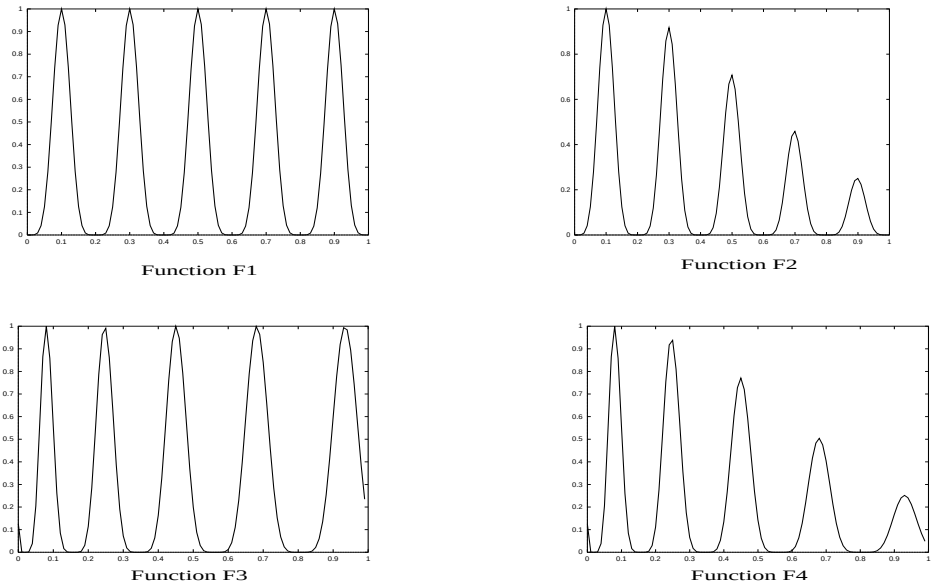


Figure 4.3: Functions F1 - F4 which provide a variety of solution landscapes

The peaks are located at, roughly, values of x of 0.08, 0.246, 0.45, 0.681 and 0.934.

4.5.4 F4 - uneven, decreasing maxima

Deb's fourth function has peaks at the same locations as F3, but with the values of those in F2.

$$F4(x) = exp\left(-2log(2) * \left(\frac{x - 0.08}{0.854}\right)^2\right) * sin^6(5\pi x)$$

4.5.5 F5 - Himmelblau's function

This two-variable function was modified by Deb to be a maximization problem:

$$F5(x, y) = 200 - (x^2 + y - 11)^2 - (x + y^2 - 7)^2$$

There are four peaks, all around 200 in height, at approximately (3.58, -1.86), (3.0, 2.0), (-2.815, 3.125) and (-3.78,-3.28).

4.6 Implementation Issues

Functions F1-F4 were run with races located as shown in Figure 4.5. The locations of the races were decided simply by dividing the search space (0..1) into ten different divisions.

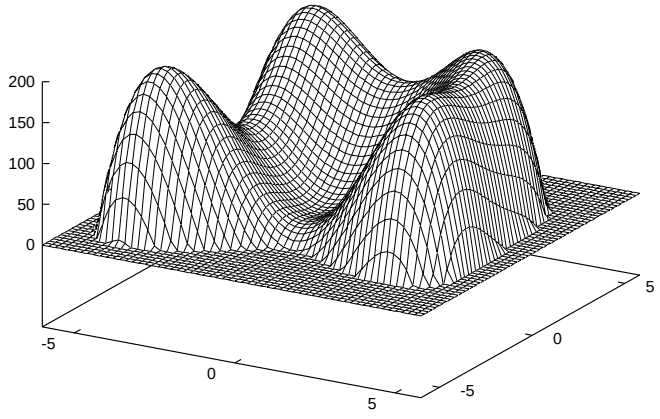


Figure 4.4: Modified Himmelblau's Function.

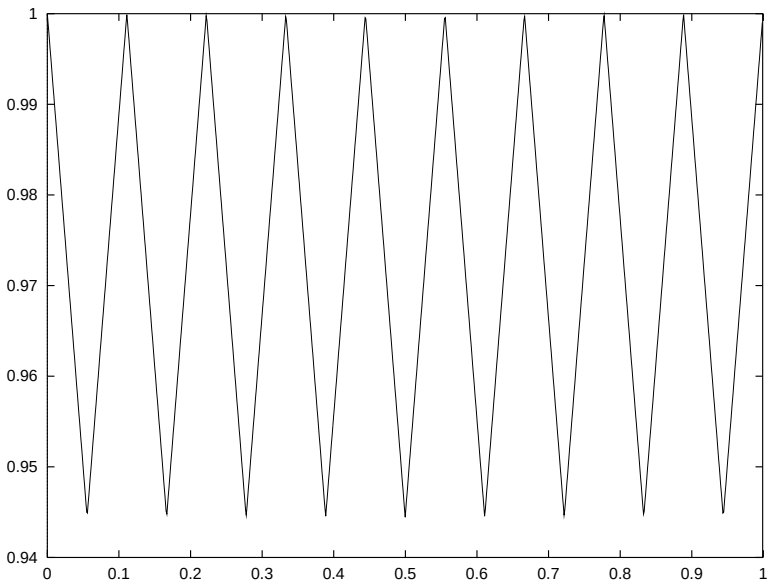


Figure 4.5: Location of racial perfects in solution landscape for functions F1 - F4.

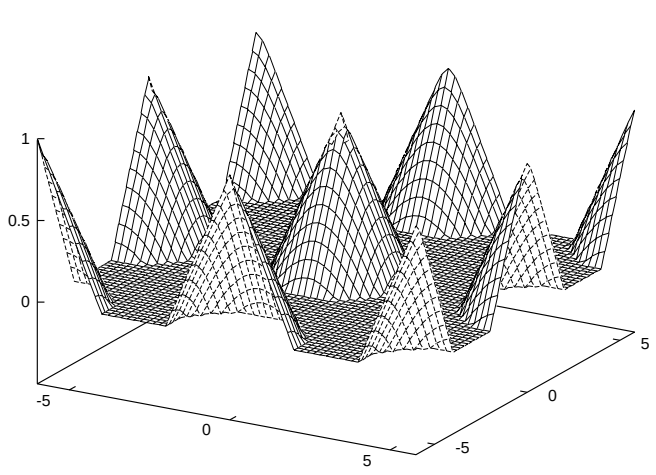


Figure 4.6: Nine races evenly divided for function F5.

In no case did the location of any race coincide with the exact location of any peak. In the cases where all five peaks were located, they were distributed evenly across the races, in the ratio 2:2:2:2:2. Occasionally one peak dominated three races, and, while some times the sole race left for the peak from which the first was taken was able to retain it's own cultural identity and evolve in its own direction, it was often the case that the remaining race was assimilated by another neighbouring race, resulting in a distribution along the lines of 2:2:3:0:3, and leading to the failure of that run to produce a solution.

Because function F5 is a two-variable function, the races had to be laid out in two dimensions. The first approach used nine races, just over twice the number of peaks, arranged evenly throughout the landscape as in figure 4.6. However, there was no overlap of races and many individuals were quite a distance from any race. It was found that using 16 races, four times the number of peaks, yielded far superior results. But, no race ever appeared at the same location as a peak. Those races which found peaks were simply near those peaks.

4.6.1 Inbreeding, outbreeding and the problem of clones

A meta-GA as described in section 2.2 was used to control the level of inbreeding taking place. For the sake of comparison, a variety of approaches were taken:

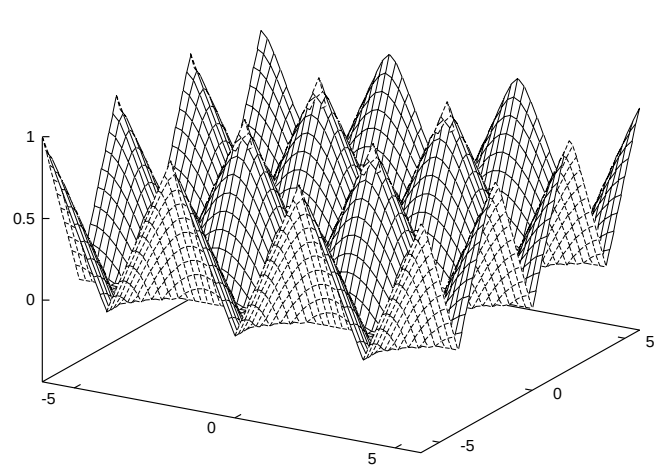


Figure 4.7: Sixteen races evenly divided for function F5.

1. Control inbreeding through meta-GA.

2. Always inbreed.

3. Meta-GA controlled inbreeding with clones disallowed.

4. Always inbreed with clones disallowed.

Disallowing clones was simply allowing no clones *in the same race*. It is still possible for several clones of an individual to be present in the population by virtue of the fact that that individual is a member of several races. The effects of disallowing clones is described in the following section.

4.7 Results

As mentioned in section 4.5, measuring how long RGA can maintain stable populations at several peaks is meaningless, as once discovered, a peak will not be lost. The first set of results compare the various implementations of RGA, while the second compare RGA with Beasley's [Beasley, 1993] sequential niche technique. Beasley's results are used in the comparison because they are the most readily available. Beasley also demonstrated that the Sequential Niche Technique greatly outperformed a standard GA on the same problem

	Fixed	Evolve	Fixed, no clones	Evolve, no clones
F1	99%	97%	98%	98%
F2	100%	100%	100%	100%
F3	99%	98%	99%	99%
F4	100%	100%	100%	100%
F5	13%	77%	21.8%	90%

Table 4.3: Comparison of RGA results.

set.

For functions F1-F4, the most striking aspect is how well RGA performed on F2 and F4. The reason RGA found those functions easier is because of the lower peaks. Ironically, it is for precisely this reason that other algorithms would find these functions the most difficult to optimize, as lower peaks are often deserted in favour of higher, more rewarding peaks.

Function F5 is perhaps the most illuminating of the five, as it is in this case that not only is the advantage of evolving parameters shown, but the advantage of disallowing clones is also clear. The advantages of these are not so clear with the other functions, as once a peak is located, optimizing it is much simpler when there is only one variable involved.

When clones are allowed, they tend to choke the exploratory power of a race. By forcing all members of a race to be different, inbreeding in that race can still improve the optimization of that peak. Varying the amount of inbreeding by evolution also improves performance. By breeding between races, cooperation is allowed between races evolving towards the same peak. Although in two cooperating races it is possible for clones to appear, there can only be one copy of each individual in each race, so the level of clones in the population is kept at an acceptable level.

In general then, the best method is to disallow clones, and allow a meta-GA to decide on the level of outbreeding. The results produced this way are now compared to Beasley's.

For functions F1-F4, the performance of both algorithms is quite similar, although probability of success for the Sequential Niche Technique varies from 90% to 100%, while the variance for RGA is much more consistent, varying only from 98% to 100%. Another

	SNT succ.	RGA succ.	SNT RMS	RGA RMS
F1	99%	98%	.0043	.0028
F2	90%	100%	.0075	.0031
F3	100%	99%	.0039	.0023
F4	99%	100%	.0041	.0024
F5	76%	90%	.20	.10

Table 4.4: Comparison of the Sequential Niche Technique and RGA.

difference is that the RMS for RGA is much smaller, at least half that of the Sequential method. Again, the most interesting result is the most difficult function, F5, and the performance of RGA is much better in this case, both in success rate and RMS. It is significant also, that SNT *requires* knowledge of the number of peaks in the landscape, while no such knowledge is needed for RGA.

4.8 Limitations of RGA

RGA reduces the knowledge of a landscape required, but does not eliminate it. In cases where one has no information about the number of peaks or where there are an enormous number of peaks, and particlarly a problem with more than two dimensions, RGA will run into difficulty.

To tackle these kinds of problems, RGA must be modified to search part of the search space at a time. If the races were *nomadic*, it would be possible for them to move about the landscape, searching for peaks, adjusting their fitness function as they did so. However, this chapter will not consider such problems.

4.9 Conclusion

A new algorithm, *RGA*, has been presented and has been shown to be able to locate all maxima of a multimodal function with little or no prior knowledge of the number or nature of the maxima. RGA explores the fitness landscape through the use of different races of individuals, each of which has its own cultural identity. In the case where there are many more races than peaks in the landscapes, several races will have a similar identity, and, under control of a meta-GA, can crossbreed.

Previous methods, such as sharing, crowding or sequential niche techniques, all require prior knowledge of a problem. Sharing and crowding are concerned to a large degree with the preservation of knowledge, and are rated by how stable their populations are when spread across several peaks. RGA's populations are inherently stable and, once discovered, a peak will never be lost. The sequential niche technique is more concerned with the discovery of all peaks of a function, and RGA has been shown to outperform it at this.

Again, a GA has been modified so that there is now greater encouragement for it to search a wider space, but no explicit knowledge about the space has to be programmed in. In this and the previous chapter, the use of races in GAs has been shown to work on two quite different areas traditionally of interest to the GA community. In both cases, using races outperformed current methods. The next chapter examines a problem area not normally associated with GAs, to demonstrate that races can also be used in genuine, real world applications.

Chapter 5

The Paragen System

5.1 Introduction

The previous two chapters showed how the Pygmy Algorithm, and the use of races in general, compared favourably against contemporary GA techniques on common, one and two dimensional, benchmark problems.

This chapter uses the foundations laid down in those two chapters to apply a genetic search to a new area, the automatic conversion of sequential programs into functionally equivalent parallel programs. This system, the *Paragen System*, utilizes GP to generate a highly parallel program from an original sequential program, while preserving functionality.

Genetic search avoids the complexities introduced by standard data dependency techniques and can also introduce further efficiencies by automatically reordering program statements. This chapter gives a brief introduction to the problem of autoparallelisation followed by a discussion of the design and implementation issues of the system. Results demonstrate the effectiveness of the system introduced.

5.2 Auto Parallelisation

The performance of serial machines is limited by the "von Neumann" bottle-neck, whereby at each instruction step, a single processor must access data stored in memory. Parallel processing systems overcome this limitation by having a number of processors working at the same time, often with their own memory. Unlike the general purpose von Neumann

architecture there is no fixed parallel processing architecture, but they can be broadly classified as those machines that operate synchronously or asynchronously. Synchronous machines have a common clock where each processor executes the same instruction at each time step and are widely programmed in the Single Instruction Multiple Data (SIMD) paradigm. With asynchronous machines, however, each processor can operate independently and these machines are widely programmed in the Multiple Instruction, Multiple Data (MIMD) paradigm. The SIMD paradigm is generally less complex than the MIMD paradigm but is less flexible and can only exploit low levels of parallelism. Therefore in this chapter we consider the more general purpose model of parallel computation, the MIMD paradigm.

Despite the power of parallel processing machines the software development cycle is considerably more complex and error prone than that for sequential software. The diversity and complexity of parallel processing architecture limits significantly the portability between different parallel machines, thus requiring the development of new software for each parallel machine configuration. Moreover there is already a vast amount of serial software for many applications, and this code would have to be totally rewritten to execute in parallel. Clearly there is a need for efficient software translators and generators to address these problems. Such parallel software development tools could perform automatic parallelisation (autoparallelisation) in full or offer the programmer the facility for exploiting low level parallelism in certain sections of code [Banarjee, 1993].

5.3 Parallelisation Problems

Current techniques for autoparallelisation rely heavily on data dependency analysis techniques. This consists of analyzing the statements of the program to determine if there is any data dependency between them. If there is no chain of dependence between two statements then they can execute in parallel. For example, to analyse whether two statements S1 and S2 are independent then the set of used variables S1.U and S2.U, and the set of modified variables S1.M and S2.M must be determined. That is, the statements S1 and S2 are independent if

$$S1.M \cap S2.U = \phi$$

and

$$S1.U \cap S2.M = \phi$$

and

$$S1.M \cap S2.M = \phi$$

If all three of these conditions hold, then statements S1 and S2 are independant [Zima, 1990]. Each of these conditions is known as a *data dependency* and are known as flow dependencies, anti-dependencies and output dependencies respectively.

With current autoparallelisation techniques these dependencies must be determined before the correct transformation can be carried out. With the Paragen system, however, these dependencies can be automatically determined, and subsequently punished, by the fitness function.

Below are examples of each of the three data dependencies.

Flow Dependence

In a sequential program a statement S1 is flow dependent on another statement S2 when the statement S1 writes to a variable that is subsequently read by statement S2.

```
S1 : A = B + 10;
S2 : C = A + 5;
```

In this case, statement S1 must be executed before statement S2 to ensure that the variable C will have the correct value.

Anti-dependency

A statement S2 is anti-dependent on statement S1 if S2 writes to a variable that was previously read by statement S1.

```
S1 : B = A + 10;
S2 : A = C + 5;
```

Again, statement S1 must be executed before statement S2. This time it is to ensure that B has the correct value after statement S1 has been executed.

Output Dependency

A statement S2 is output-dependent on a statement S1 if S2 writes to a variable previously written to by statement S1.

```
S1 : A = B + 10;
S2 : A = C + 5;
```

The order must be preserved in this case to ensure that A has the correct value after executing S1 and before executing S2. Typically, in a real world example (as will be shown in Section 5.5), there will often be one or statements between S1 and S2 that employ A.

Problems with Data Dependency Analysis

Autoparallelisation techniques that rely on data dependency analysis - the isolation of the dependencies outlined above - have limitations due to both the complexity of applying data dependency rules and the considerable processing required by the application of any subsequent transformations to sequential code[Blume, 1994].

The *Paragen* system avoids the complexities associated with data dependency analysis by utilising the power of Genetic Programming to allow the automatic conversion of serial programs. Paragen does not employ any data dependency rules or analysis of any kind, either algorithmically or on the part of the programmer, thus considerably reducing the effort required for parallelizing code.

5.3.1 Loops

The parallelisation of loops is one of the most important aspects of autoparallelisation, as the bulk of computing workloads are controlled by loop type structures. In parallelizing loops, there may be data dependencies both within the loop itself and across the different loop iterations [Braunl, 1993]. Data dependencies that span loop iterations are known as *cross iteration dependencies*. This is especially significant as many parallelisation techniques attempt to execute the different iterations of a loop in parallel, but ignore any parallelism within the loop. This approach alleviates the data dependency constraint within the loop as the original sequence of statement is often preserved. However with the flexibility of the Paragen system we will also attempt to simultaneously extract parallelism both within the loop and across loop iterations.

Another consideration in the parallelisation of loops is array access. Loop control structures are often used in the processing of arrays and this further complicates data dependency analysis techniques. The parallelisation of loop iterations in the Paragen system is specified by the **DoAcross** function, described in Section 5.4.1. For example, the following code segment has a flow dependency within the loop iteration.

```
for i := 1 to n do
  begin
    a[i] := b[i];
    c[i] := a[i];
  end;
```

The code generated by Paragen would have to preserve the data dependency within the loop iteration, but also parallelise each of these iterations. The parallelisation of loop iterations in the Paragen system is made possible by the introduction of another function, the **DoAcross** function. **DoAcross** takes three arguments, the initial value of the index, the number of iterations and the loops to be executed. A corresponding sequential loop was also made available, **Do**, for any cases where the loop cannot be parallelised.

```
DoAcross i := 1 to n
  begin
    a[i] := b[i];
    c[i] := a[i];
  end;
```

In the following example, described in [Braunl, 1993], there is a data dependency across different loop iterations due to the statements S2 and S3. Here statement S3 of one iteration must be executed before statement S2 of the next iteration, due to the flow dependency caused by e[i-1]. This requires a synchronization mechanism to be used between different loop iterations. However, because the Paragen system can also reorder the statements within the loop iteration, it is possible in many cases to minimise synchronization operations such as semaphores.

```
for i := 1 to n do
  begin
    S1: a[i] := b[i] + c[i];
    S2: d[i] := a[i] + e[i-1];
    S3: e[i] := e[i] + 2 * b[i];
    S4: f[i] := e[i] + 1;
  end;
```

5.3.2 Arrays

Another consideration in the parallelisation of loops is array access. Loop control structures are often used in the processing of arrays and this further complicates data dependency analysis techniques. The loop functions available to the Paragen system are outlined in the previous section.

In the following example, which is an extended version of the problem described in [Braunl, 1993], (and section 5.3.1) there is a data dependency across different loop iterations due to the statements S4 and S8. Here statement S4 of one iteration must be executed before statement S8 of the next iteration, due to the flow dependency caused by

e[i-1]. This requires a synchronization mechanism to be used between different loop iterations. However, because the Paragen system also converts the statements within the loop iteration, by re-ordering and / or parallelisation, the effect of the synchronization mechanism on performance is minimized. In the case of generated code being run on a machine that requires synchronization, a simple post-processing can be applied to the program to determine where synchronization mechanisms are needed.

Our extensions to this problem are the introduction of a number of dependencies within the loop, between S1 and S2, between S6 and S7 and between S5 and S6.

```
for i := 1 to n do
  begin
    S1: a[i] := f[i] * 2;
    S2: f[i] := 57;
    S3: g[i] := d[i] + 8;
    S4: e[i] := e[i] + d[i];
    S5: b[i] := a[i] + d[i];
    S6: a[i] := b[i] + 2;
    S7: a[i] := d[i];
    S8: c[i] := e[i-1] * 2;
  end;
```

5.4 The Paragen System

The Paragen System was designed to be able to take a serial program, and automatically transform it into a functionally equivalent one. Individuals in the Paragen system are parse trees which map a program onto a (virtual) parallel machine. Once the program has been downloaded to this machine, the individual can be tested. As in all GP experiments, the important factors are the functions, the terminals and the testcases. However, due to the nature of the Paragen system, it is possible to select all of these *automatically*.

5.4.1 Functions

Four functions, all of the form **XN**, were made available to the system where **X** denotes whether the subtrees are to be evaluated in **P**arallel or **S**erially. **N** denotes the number of subtrees attached to this node. These functions are **P2, P3, S2** and **S3**. Every experiment used these functions, and runs which were being applied to programs with loops were given two extra functions, as described in Section 5.3.1 to deal specifically with loops.

The two functions, namely **DoAcross** and **Do** were treated as though they were ADFs. Each one could be invoked as follows :

```
(DoAcross (Block_of_statements))
```

The "block_of_statements" could consist of one or more statements, and could be executed in either parallel or serial mode, or a combination of the two.

The loop functions can be identified by a simple scan of the original program. Each loop in the original program generates *two* functions, one which executes the loop in parallel, the other in serial fashion.

5.4.2 Terminals

Unlike most other parallelisation techniques, Paragen makes no attempt to analyze the original program. In fact, Paragen completely disassembles a program into its constituent statements, and attempts to rebuild it in a parallel form using these statements and a combination of the parallel and serial functions available.

Each statement(line of code) in the original, serial program, is used as a terminal by Paragen. Automating the selection of terminals is a trivial task, as each can simply be read from the original program.

5.4.3 Testing individuals

The first step in testing an individual in Paragen is to instantiate it on a (virtual) parallel machine. Individuals are then subjected to a number of test-cases, each of which involves different starting values for the variables used. The program is then executed on the parallel machine and the resulting values compared to the those values produced by the

serial program. Each time the parallel version gets the same result as the serial version, its score is incremented.

Due to the simple nature of testing the individuals, there can be as many test-cases as desired, as each test-case can be randomly generated. It was found that this helped avoid the problem of "overfitting", i.e. a situation where the individual generated by GP learns a small number of testcases perfectly, but cannot generalise to unseen data.

5.4.4 Selection in Paragen

Most problems tackled by GP involve solving a set problem, such as sorting numbers, recognizing a shape, performing regression etc., but the problem of autoparallelisation is not quite so straightforward. The problem faced by Paragen is a two-pronged one, not only must individuals be functionally identical to the original program, but they must also be as parallel as possible.

This is fairly typical of problems often faced when attempting to solve real world problems[Beasley, 1993] - there is rarely just a single goal to be optimised. The Pygmy Algorithm, however, is perfectly suited to the problem, and can be implemented using two races.

The first fitness measure for an individual, X, is $correctness(X)$, which is a measure of how well the individual replicated the functionality of the original program. For the system to be of any practical use, the *best-of-run* individual must have a 100% score in this.

The second fitness measure for an individual, X, is $time(X)$, which is the number of time steps it takes to execute the individual. The lower this score is relative to the original program, the more parallel an individual is.

The first race of individuals were scored only using the *correctness* measure, with any ties being resolved by comparing their *time* score. Individuals in the second race used a combination of the two fitness measures :

$$fitness(X) = correctness(X) + time(X)$$

When an individual is tested, it is compared against the individuals in each of the parent populations. If the individual has scored high enough to enter the population, the

Objective :	Produce a functionally equivalent program that is as parallel as possible
Terminal Set :	All the statements in the original Program
Function Set :	P2, P3, S2 and S3. For the programs which involved loops, there were an extra pair of functions, Do and DoAcross for each loop
Fitness cases :	50 randomly generated initial values for the input variables
Raw Fitness :	The number of fitness cases for which the parallel program outputs the correct values
Standardized fitness	Same as raw fitness
Hits :	Same as raw fitness
Wrapper :	None
Parameters :	M = 20 - 100, G=20 - 50
Success Predicate :	None

Table 5.1: A Koza-style tableau summarizing the control parameters for a typical Paragen run

lowest member of that population is removed. In this respect, Paragen adopts the steady state approach described in the previous chapter, as individuals are always preserved.

5.5 Paragen - The First Results

This chapter serves very much as an introduction to the workings of Paragen. To illustrate how Paragen operates, it has been tested on a variety of common parallelisation problems from the literature[Braunl, 1993], all of which tested Paragen's ability to deal with the dependencies described in Section 5.3.

Paragen successfully transformed all those programs, including those which involved loops and arrays. As the programs with loops and arrays are generally held to be the most difficult [Banarjee, 1993], and produced the most interesting results, only their results will be discussed.

5.5.1 Test program from Section 5.3.1

The resulting code generated by Paragen:

```
DoAcross i := 1 to n do
  begin
    S3: e[i] := e[i] + 2 * b[i];
    S1: a[i] := b[i] + c[i];
    S4: f[i] := e[i] + 1;
    S2; d[i] := a[i] + e[i-1];
  end;
```

This code runs the loop in parallel, but has also several of the statements reordered to cope with dependencies. Notice how S3 now occurs *before* S2 to avoid any problems with the cross-iteration dependency. If these statements were not reordered, the program would still be correct, assuming that the loop was executed serially. However, the extra selection pressure for parallelism ensures that this is not the case.

When solving this problem, Paragen was run with a population of 80 individuals for 10 generations.

5.5.2 Test program from Section 5.3.2

In this case, not only is each iteration of the loop executed in parallel, but so too are several of the instructions within each iteration. The execution time for this program would be reduced from 8*n time steps to just 4 time steps.

The resulting code produced by Paragen is over the page.

```
DoAcross i := 1 to n
  begin
     PAR-BEGIN
       S1: a[i] := f[i] * 2;
       S3: g[i] := d[i] + 8;
       S4: e[i] := e[i] + d[i];
     PAR-END
     PAR-BEGIN
       S8: c[i] := e[i-1] * 2;
       S5: b[i] := a[i] + d[i];
     PAR-END
     PAR-BEGIN
       S2: f[i] := 57;
       S6: a[i] := b[i] + 2;
     PAR-END
     S7: a[i] := d[i];
  end;
```

As mentioned in section 5.3.2, this code segment contains no less than three data dependencies, and also a cross-iteration dependency between S4 and S8. In the resulting code, not only is each iteration of the loop executed in parallel, but so too are parts of the body. This problem required more effort on the part of Paragen, and required a population of 100 individuals running for 20 generations.

5.5.3 GP without races

While there can be no doubt that the Paragen system is successfully transforming individuals, it is difficult to say how much of the success is due to the use of the Pygmy algorithm. To see how a normal GP system would cope with the problem, an experiment with identical functions and terminals, and with the same parameters as described in table 5.1.

It was found that GP suffered the same problem as a standard GA did in Chapter 3.

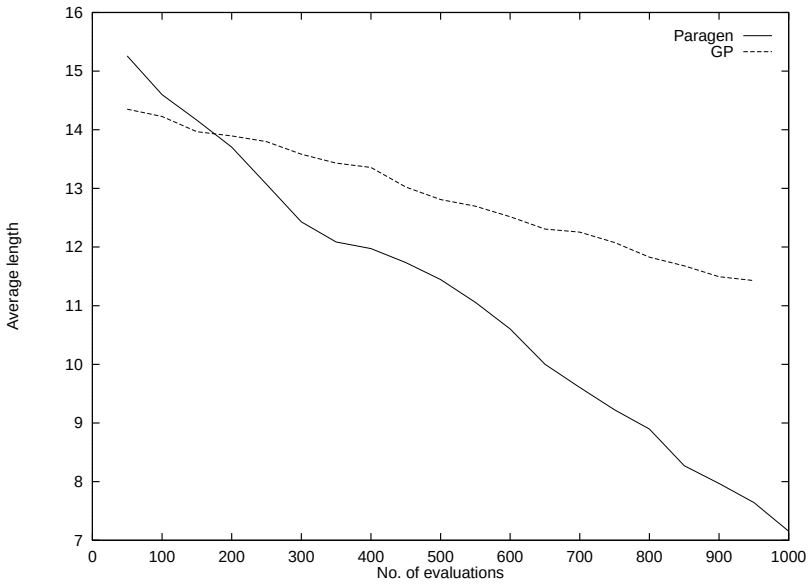


Figure 5.1: Average length of equivalent individuals

The population was unable to balance the two requirements, and, while parallel individuals were generated, they were often created at the cost of equivalence. The two methods are compared below in figure 5.1, where it can be seen that the Paragen system produced shorter equivalent individuals.

5.5.4 Paragen and loopholes

As mentioned earlier, the fitness of a program is a combination of how correct it is and how parallel it is, but Paragen (and GP in general) proved to be quite adept at discovering loopholes. A common strategy, especially in the shorter examples, was for Paragen to execute all the instructions in parallel, regardless of dependencies. This gave the individual a high score in terms of parallelism - the program executing in only one time step - and a high score in terms of correctness was achieved by ensuring that each statement was executed so many times that the variables affected ended up with the correct values. This behavior was curbed by reducing an individual's fitness each time it repeated an instruction.

For the following program, which could be reduced from 3 to 2 time steps:

```
a=b+c;
b=d+e;
f=g+h;
```

A number of individuals of the form

```
DoPar
    f=g+h;
    DoPar
        a=b+c;
        a=b+c;
    DoPar
        b=d+e;
        b=d+e;
```

These individuals took only one time step to evaluate, but execute each of the first two statements twice each in parallel. This led to uncertain results, as it is impossible to know if they will preserve the data dependency. However, in many cases they did, which resulted in the individuals concerned getting an exaggerated score. This behaviour was curbed by reducing an individual's fitness each time it repeated an instruction.

Paragen also noticed that in the case of output dependencies, where statement S2 writes to the same variable as S1, S1 could often be left out of the final program as it did not affect any final values. Clearly, this is not correct, as when a program is being executed the first value of the variable might be used in a statement that does not alter any variables. This made it necessary to reduce an individual's fitness each time it left out an instruction, i.e. instead of producing code like

```
S0 : ...
S1 : A = B + C;
S2 : A = B + 10;
S4 : ...
```

Paragen instead produced

```
S0 : ...
S2 : A = B + 10;
S4 : ...
```

This resulted in a correct program because A had the correct final value, and also resulted in a shorter program.

Another strategy that had to be discouraged was that of individuals who did not do anything. These individuals exploited the fact that several variables used by many programs do not change, and by not doing anything these individuals would get a score by virtue of the fact that these variables will always contain the correct value at the end of the run. To prevent individuals such as these from having a chance of being selected, only those variables which appear on the left hand side of statements are considered when calculating an individual's score.

5.6 Paragen and Races

The Paragen system employs two races to help balance the population between correct and parallel individuals. There is no reason why GP with a single single population couldn't be used to evolve similar individuals, but, as in Chapter 3, would have difficulty balancing the population.

To compare the two schemes, a normal GP system was applied to evolving the individuals reported on above, with the fitness function as below:

$$\text{fitness} = \textit{score} \text{ - } (\textit{repetition punishment}) \text{ - } (\textit{time taken})$$

Where *score* is how closely the individual matches the result of the original program, *repetition punishment* a punishment for the number of instructions repeated, and *time taken* a punishment for the length of execution.

5.7 Conclusion and Future Directions

The Paragen system is a new auto-parallelisation tool which can be used to convert serial programs without any analysis for dependencies. Paragen has been successfully tested with code containing a number of data dependencies, and has successfully converted loops containing cross-iteration dependencies. As well as converting entire loops, Paragen can also determine sections of code within those loops that can be parallelized while still preserving any dependencies contained within.

Paragen also demonstrated the ability of GP to discover any "loopholes" in a fitness function, and demonstrated that it will always take the path of least resistance. The next step for Paragen is to be applied to larger programs, consisting of several functions. This work is beyond the scope of this thesis, and is being continued in [Ryan and Walsh, 1995] [Walsh, 1996].

Perhaps most importantly, GP and The Pygmy Algorithm have been introduced to a real world problem. For GP to be taken seriously by outsiders to the field, it is problems such as this that it must be shown to excel at.

Chapter 6

The Degree of Oneness

6.1 Introduction

Genetic Algorithms tend to be applied to problems with fixed or canned sets of test cases. Indeed, this thesis has so far considered only problems such as these. However, as problems become larger, the testing time frequently becomes impractical. To reduce testing time, sometimes only a subset of the test cases are used in a particular generation. The subset may vary with time[Siegel, 1994], may evolve[Hillis, 1989], may vary with spatial locality[Ryan, 1995b] or perhaps the goal of the problem itself may vary with time[Goldberg and Smith, 1987]. In problems such as these it is important that the population does not "forget" useful solutions or gene combinations previously discovered.

A natural method for creating a genetic memory is the use of diploid genetic structures. Diploid structures, as opposed to traditional, haploid structures, maintain two copies of each gene. A result of having two copies of each gene is that calculating the phenotype is no longer as simple as mapping a genotype on to a phenotype and some method must be employed to calculate a single phenotype from two genes.

The fact that two genes are kept for each phenotype permits the shielding of genes which might otherwise be lost through selection pressure, but that might be useful at some time in the future or might have been useful in the past. A genetic algorithm which can remember previous solutions, which may be needed again in the future, while still performing well with the current problem set, would be very useful for the sort of problems mentioned above. This chapter describes an implementation for diploid structures, which

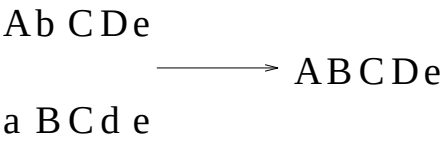


Figure 6.1: A simple dominance scheme

suits not only standard, binary genetic algorithms, but also nonstandard genotypes, such as high level implementations used in previous chapters and Genetic Programming.

6.2 Diploidy in Genetic Algorithms

At its simplest level, diploidy involves two copies of each gene, one of which is selected for expression in the phenotype. This method of selection, often termed Mendelian diploidy, generally operates using dominant and recessive genes. When decoding a heterozygous gene locus, i.e. two different genes, one selects the dominant gene. In the case of a homozygous locus, the problem of which gene to use does not arise. Figure 6.1 shows an example of a simple dominance scheme, where upper case letters signify the dominant form of a gene, while lower case letters signify the recessive form.

Simple dominance such as this is adopted directly from nature. For example, the genes which govern the ability to roll one's tongue and those that determine whether or not one has hairy fingers are governed by Mendelian dominance. The genes which bestow a person with the talent to contort their tongue are all dominant, while the gene that results in hairy fingers is also dominant[Pai, 1985].

Perhaps there is some reason why it is more likely that one can roll one's tongue than not, or perhaps there may even be a selective advantage in having the ability to do so. If there is, nature has had several billion years to discover it. Those using GAs do not have the luxury of time, and cannot decide in advance which gene is to be dominant without using prior knowledge of the solution to the task at hand. This is a problem because the dominant form of a gene is more likely to be expressed - the recessive form will only be expressed at homozygous locus, meaning two copies of it must be present.

Goldberg's comprehensive study of diploidy[Goldberg and Smith, 1987] showed that Hollstein's triallelic scheme, which recognised the problem of bias in simple domi-

	1	1_0	0
1	1	1	1
1_0	1	1	0
0	1	0	0

Table 6.1: Triallelic dominance map

nance, was the most effective of current diploid schemes. Hollstein overcame the problem of bias with the use of three alleles, namely 1, 1_0 and 0. In his scheme, 1 dominates the other genes, and a locus with 1 always resulted in a phenotype of 1, while 0 dominated 1_0. A homozygous locus with 10 resulted in a 1 being expressed. This is summarized in Table 6.1.

Although there is still bias to a certain degree, in that there are twice as many loci that result in 1 being expressed, use of this scheme allows either phenotype to be held in abeyance, unlike the simple dominance described above. However, Hollstein's method does not map very obviously on to high level, nonstandard implementations, i.e. where there are more than two possible phenotypes. For instance, in an implementation with 4 or 5 alleles there is no simple way of choosing which should dominate which, while still ensuring that no one allele is completely dominated and that each can be held in abeyance, a serious setback as nonstandard genotypes are being used more and more frequently.

6.3 Natural Diploidy And Incomplete Dominance

Although simple dominance is taken directly from nature, it is not the only scheme employed by nature to resolve diploid structures. Another scheme, incomplete dominance, is used by many plant and animal alleles for resolving heterozygous loci, particularly in traits that have more than two simple values[Pai, 1985] [Suzuki, 1989] [Strickberger, 1990]. An example of incomplete dominance in operation was described by Pai using four o'clock flowers. She described how, if one was to mate two of these flowers, one red and the other white, all of the offspring would be pink flowers. On the other hand, if one were to mate two pink flowers, the resulting offspring would be of three different colours, red, pink and white, in the ratio 1:2:1.

The resulting colour of an offspring can be explained by examining the intensity

of colour of its parents, and by looking upon colour as being a quantitative trait, where the interactions between gene pairs is an additive one.

Pai also showed that a red flower contains the genes r_1r_1 and a white flower the genes r_2r_2. Each gene contributes toward one particular colour, r_1 to red and r_2 to white. So the more r_2 genes present the more likely a flower will be white, on the other hand, the more r_1 genes, the more likely a flower is to be red. In the case of both genes being present, each contributes towards its own particular colour, resulting in half the genes suggesting the plant be red, and the other half that the colour be white. Because colour is a quantitative trait, the colour expressed is simply mid-way between the two: pink. Moreover, both genes contribute to a trait, rather than there being a choice between them.

If one were to describe additive traits in Genetic Algorithms, a simple 1 or 0 phenotype would appear to be insufficient. Choosing say, 0 for red and 1 for white, is suitable for homozygous loci, but cannot cope with pink. However, if one could describe the colours in degrees rather than absolutes, one might have more success. By describing the flowers in terms of white only, one could say that red flowers are "not white", pink flowers are "sort of white" and white flowers are simply white.

Applying this to binary Genetic Algorithms, one could describe a locus in terms of its degree of oneness. Some loci would have a very low degree of oneness, i.e. be expressed as 0, while others would have a high degree of oneness, i.e. be expressed as 1. Using two alleles as in the flower example is unsuitable however, because there are three phenotypes. Using three alleles results in six phenotypes, which appears suitable at first. However, it is difficult to assign values to each allele, for, while two alleles can tend towards either 1 or 0, the choice of the third value can result in a bias. Using four alleles however, makes the choosing of values far easier, as two alleles can tend towards each phenotype. The values selected were 2, 3, 7 and 9, although many other combinations could have been chosen.

The alleles were named A, B, C and D, and the following degrees of oneness resulted as described in Figure 6.2. Degrees less than or equal to 11 were expressed as a phenotype of 0, while those greater than 10 were expressed as 1.

Using this new scheme, there is no bias toward either phenotype, as illustrated in Table 6.3. The fact that dominance of sorts emerged while using this scheme, i.e. D dominates all other alleles, is of no consequence as bias toward either phenotype is eliminated.

Phenotype = 0	Phenotype = 1
AA(4) AB(5) BB(6) AC(9) BC(10)	AD(11) BD(12) CC(14) CD(16) DD(18)
The Degree of Oneness ⟶	

Figure 6.2: Mapping the degree of oneness onto two phenotypes

	A	B	C	D
A	0	0	0	1
B	0	0	0	1
C	0	0	1	1
D	1	1	1	1

Table 6.2: Dominance map for the additive scheme.

As one would expect, using diploidy allows the shielding of genes as described in the introduction. It is possible that two individuals of the same phenotype could mate and produce a child of a different phenotype. For instance, mating AD and AD, both phenotype 1, would result in four possible children, three of phenotype 1 and one of phenotype 0. In the case of two phenotype 0s mating, e.g. AC and AC or BC and BC, there would be three offspring of phenotype 0 and one of phenotype 1. In certain cases, genes might not be shielded against.

Parents both of extreme degrees of oneness will tend to produce children similar to them. In any case, the additive effects ensure that degree of oneness of a child will resemble that of its parents, leading to a smooth change of phenotype, i.e. a parent consisting of AA would never produce a child with a degree of oneness greater than 10, while parents possessing genotypes with a degree of oneness close to 10 can result in children of either phenotype.

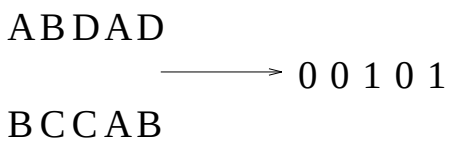


Figure 6.3: Additive Diploidy

6.4 Comparison

To test the viability of the additive scheme, both it and Hollstein's triallelic scheme were applied to the following problem : Evolve a string of binary digits that matches a goal string, which consists of either all 1s or all 0s. The goal string switches values every n generations.

This problem is beyond the ability of traditional genetic algorithms [Goldberg and Smith, 1987], which cannot change to a different goal string during a run as the population fixates on the initial goal. Both types of diploid schemes, however, are able to adapt to this change in the environment.

Two variants of this test were used in experiments for this paper. In the first test, the goal string was switched between the two possible values every twenty-five generations, while in the second, the values were switched every five generations. Goldberg's test bed for diploid structures was a variant on the knapsack problem where the maximum permissible weight rose and fell every n generations. While the problem is difficult when the weight falls, as current high performing individuals now present infeasible solutions, the problem of individuals being totally unsuited to the environment does not arise when the weight allowed is increased. In the problem presented here, each change in the environment presents a totally different problem, and the population must maintain both solutions.

All experiments were carried out on populations varying from one- to four-hundred individuals, with a probability of crossover of 75%, no mutation and with a goal string 20 bits long. To ensure a representative sample was attained, each experiment was repeated on three hundred populations. Individuals were thus assigned a score ranging from 0..20. To test each method, the probability of an individual attaining a score of 20 was calculated. The probability of an individual attaining a score of 19 or higher was also calculated. Figure shows that in the case of the goal changing every twenty-five generations, the additive

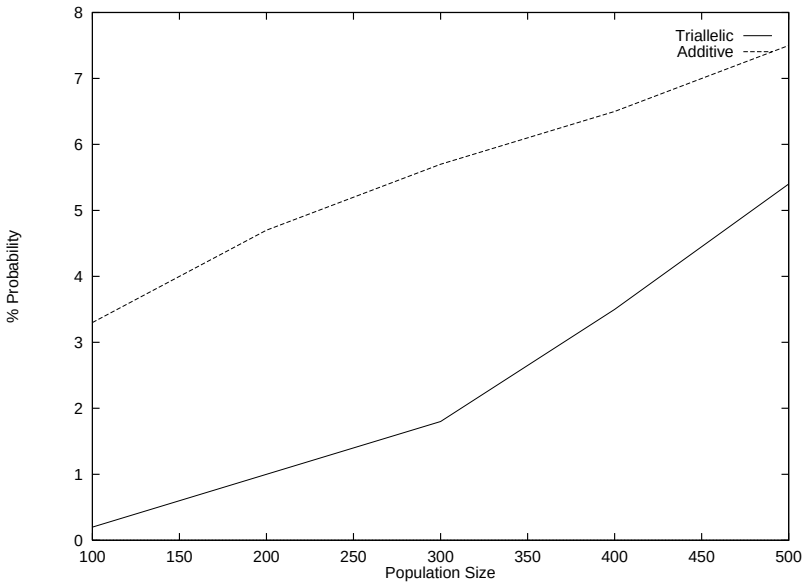


Figure 6.4: A comparison of the triallelic and additive schemes with an environment change every 25 generations.

scheme is more likely to produce an individual scoring 20.

To further test the two schemes, another probability was calculated, this time the probability of an individual with the desired score appearing after the 300th generation. However, in no case did either scheme produce an individual scoring 20 after the 300th generation. The graphs are charted this time for individuals with scores of 19 and 18 appearing and are in Figure 6.5.

Testing the schemes this late in a run shows how difficult it is for them to still maintain a balanced population after much time has passed. In the lower population sizes high scoring individuals did not even appear. The performance of both schemes is very close, but the additive scheme again outperforms the triallelic.

The second test involved the goal symbol changing every five generations. Although populations have less time to converge on a particular symbol, they also have less time to react to a change in the environment. The turbulent environment proved the dominant factor and both schemes found this test much harder. Again, probabilities were calculated over 500 generations and in the period after generation 300.

Figure 6.7 suggests that in this case the triallelic scheme outperforms the additive one, showing a greater probability of individuals of either type appearing at any population

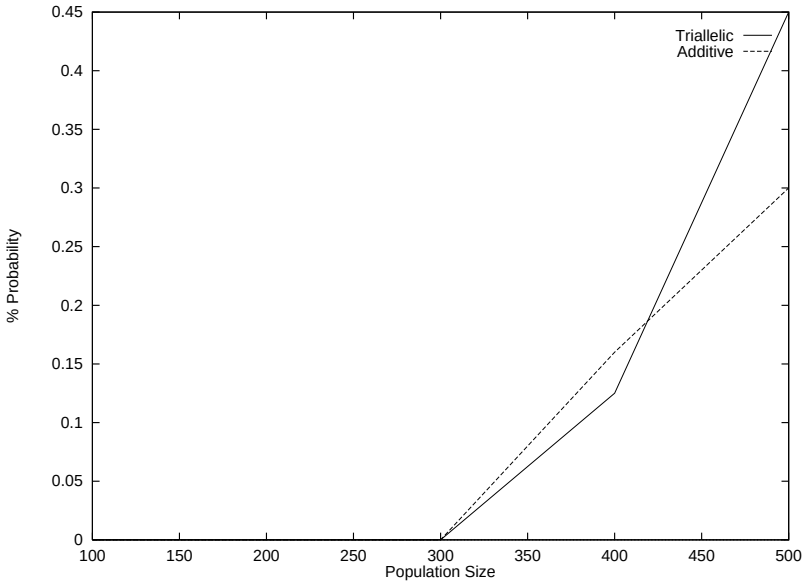


Figure 6.5: A comparison of the triallelic and additive schemes scoring 19 after the 300th generation with an environment change every 25 generations

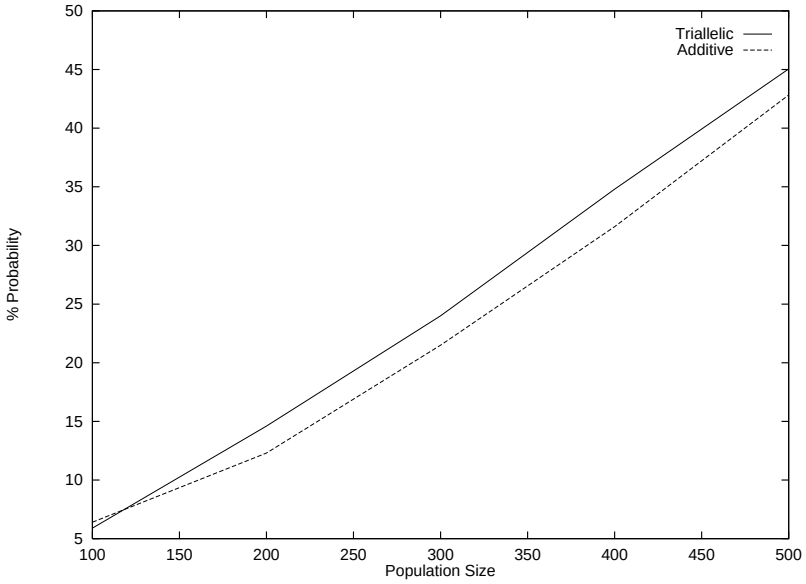


Figure 6.6: A comparison of the triallelic and additive schemes scoring 18 after the 300th generation with an environment change every 25 generations

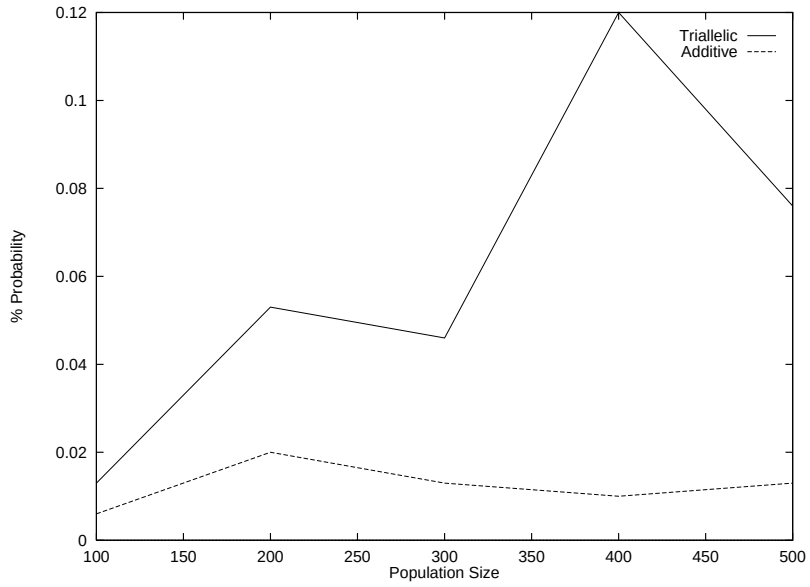


Figure 6.7: A comparison of the triallelic and additive schemes with an environment change every 5 generations

size, but the graph of the probabilities measured after generation 300 tells a different story. Although under this measurement the additive scheme failed to produce an individual scoring 19 or higher, it consistently outperformed the triallelic scheme at producing individuals scoring 18 or higher at this stage in the proceedings. The trialleic scheme did produce some individuals scoring 19, but the probability of such individuals appearing was calculated to be .008%.

Each scheme accumulated most of its score for the first measurement in the early part of each run and, at this stage of the experiments, the results were somewhat inconclusive. In the case where the environment changed after twenty-five generations, which allowed a population to converge before being shaken up, the additive scheme was clearly better. However, in the case where the environment changed every five generations, which forces a population to react quickly, the triallelic scheme performed better.

The justification for using diploid structures is the genetic memory they give to a population, in most cases in each of the two experiments the additive scheme is clearly the stronger toward the end of a run, showing that it is the stronger at remembering solutions after time has passed.

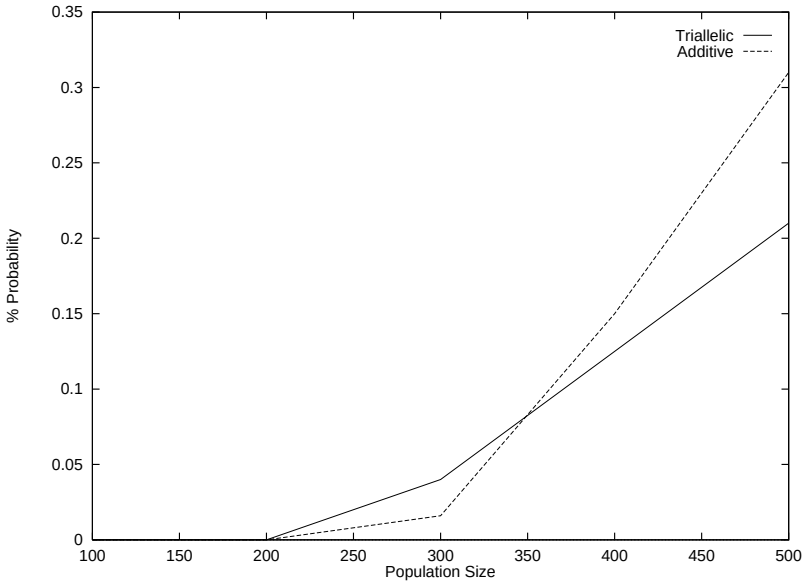


Figure 6.8: A comparison of the triallelic and additive schemes scoring 18 with an environment change every 5 generations

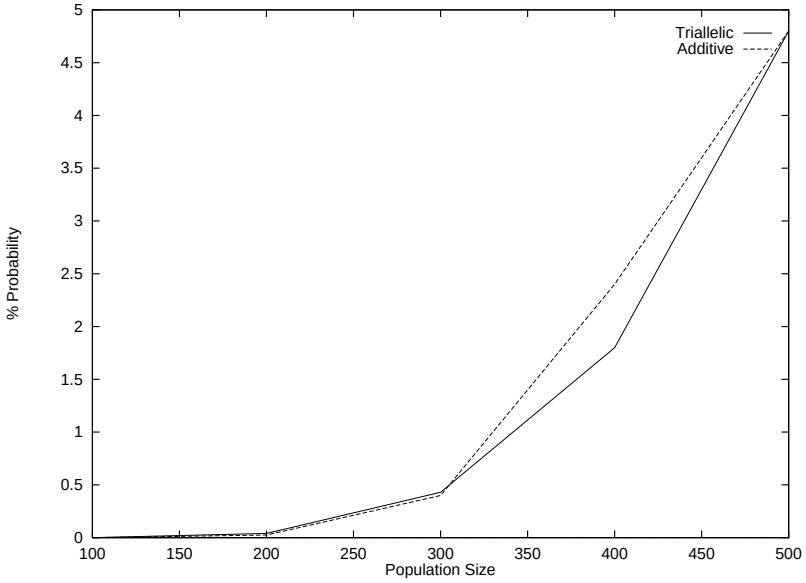


Figure 6.9: A comparison of the triallelic and additive schemes scoring 17 with an environment change every 5 generations

6.5 The Degree of *N*ness.

While additive diploidy has been shown to compare favourably to Hollstein's triallelic scheme for binary GAs, it is in the area of non-standard GAs that the additive scheme excels, for it can be applied to GAs with any number of phenotypes.

The only other implementation of diploidy in non-standard GAs was that by [Hillis, 1989], in his classic co-evolution paper. The method employed by Hillis for resolving a heterozygous locus was to include both alleles, while homozygous loci were resolved by the inclusion of a single allele. Due to the possibility that differing numbers of alleles may be included, this method is only suited to variable length implementations, and there is also no way for the GA to know in what order to include alleles.

Using additive diploidy, there is no need to use variable length structures and the problem of ordering alleles does not arise. The implementation of high level diploidy presented here is referred to as the degree of Nness, where N is the number of the last phenotype. Phenotypes can take any form, not just numbers, and need only be listed in order. Some care is needed when ordering phenotypes, as those that are close together are easier to change between than those at extremes. To design a diploid structure for a high level genetic algorithm, with say, N phenotypes, one simply selects the appropriate number of alleles which will produce either N phenotypes, or, preferably, some number divisible by N. One must then set about choosing a value for each allele, and this must permit the choosing of appropriate cut off points, i.e. the areas at which the phenotype changes. The number of phenotypes produced by $\mathbf{t}$ alleles is

$$C(t+1,2) = \frac{(t+1)!}{2! * (t-1)!}$$

For example, to implement a Genetic Algorithm with five phenotypes, one could use four alleles, which result in ten intermediate phenotypes, which can then be interpreted as five phenotypes. It was found that, in general, implementations with an odd number of phenotypes were the easiest to design, and generally resulted in smoother phenotypic maps. In this case, again using A, B, C and D, although this time with the respective values 0, 1, 2 and 3, the phenotypic map is as in Table 4 below. Like the previous example, these are not the only possible values, and were chosen rather arbitrarily, value sets such as 0, 2, 4

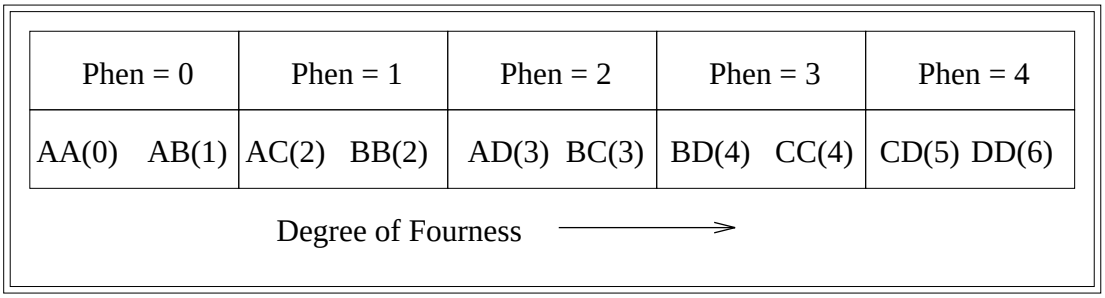


Figure 6.10: Mapping the degree of fourness onto five phenotypes

and 6 or 0, 1, 3 and 4 are also possible as they produce a set of phenotypes similar to those in Table 4.

It is also possible to use a number of other alleles. One could use 5 which would result in 15 phenotypes, or 9 which would result in 45 and so on. In general, the more phenotypes available, the smoother the progression from each expressed phenotype will be, e.g. using 30 alleles would produce 465 phenotypes, leading to very smooth progression and less possibility of fixation on one particular phenotype. Of course, the more alleles, the more difficult it is to discover suitable values for them.

6.6 Polygenic Inheritance.

Additive traits need not only be governed by the interaction of a single locus or gene pair. The first instance of this in natural biology was discovered in 1909 by Nilsson-Ehle[Pai, 1985] when he showed that the kernel colour in wheat, an additive trait, was in fact managed by two pairs of genes. Inheritance of genes of this type is known as polygenic inheritance. Polygenic inheritance could be of interest in this case because the more gene pairs involved in the calculation of a trait, the more difficult it is to distinguish between various phenotypes - clearly a situation which would benefit the diploid genetic algorithms in this chapter. The most important property of additive effects is the continuity of change of phenotype, and the effect of polygenic inheritance on the problem used earlier is shown below. In this case, five alleles were used, and two gene pairs were used in the calculation of each phenotype, resulting in 70 different phenotypes, 35 of which were described as 0, and 35 as 1. The five alleles used were A, B, C, D and E with degrees of oneness of 0, 1, 2, 3 and 5 respectively. The threshold value was 9. This set of values was chosen rather arbitrarily,

as there are many sets that would be suitable. With different values for each allele in the region 0..5, there were 7 possible ways to encode the system so that the resulting alleles could be split into 2 even groups. The greater the range of values examined for each allele, the greater the number of encoding schemes result, e.g. using a range of 0..9 results in 73 different possibilities. The values used in this chapter were simply the first encountered, using the smallest possible range.

Polygenic inheritance was compared to the two original schemes in the same set of experiments and, as can be seen from Figures 6.11 and 6.12, outperformed each of the other two schemes in all tests. The figures below show polygenic inheritance compared to the other two methods for the highest score in each experiment. Not only does polygenic inheritance outperform the other two methods in all cases, but because it is based on additive gene effects, it too can be applied to nonstandard, high level systems in a similar manner to standard additive diploidy.

6.7 Conclusion

This chapter has introduced a new diploidy scheme, additive diploidy, taken directly from nature, which not only outperforms current methods for binary genetic algorithms, but can also be applied to non standard GAs with more than two alleles. An extension of additive diploidy, polygenic inheritance, which uses several genes to control a phenotype has also been described and been shown to considerably outperform other methods. The use of diploid structures in GAs endows a population with a genetic memory and thus permits the use of a nonstationary set of testcases. Using a nonstationary set of testcases allows the testcases to be either evolved, and thus increase the difficulty of a problem as time goes on, or to reduce the testing time. Previous diploid schemes were either problem specific, or confined to binary genetic algorithms. Additive diploidy, on the other hand, can be used with any number of alleles and is general enough to be used with practically any problem. It has also been shown that for high-level GAs there need not be a simple one-to-one mapping from phenotype to genotype, and that the more genes involved in each phenotype the smoother evolution performs in a variable environment.

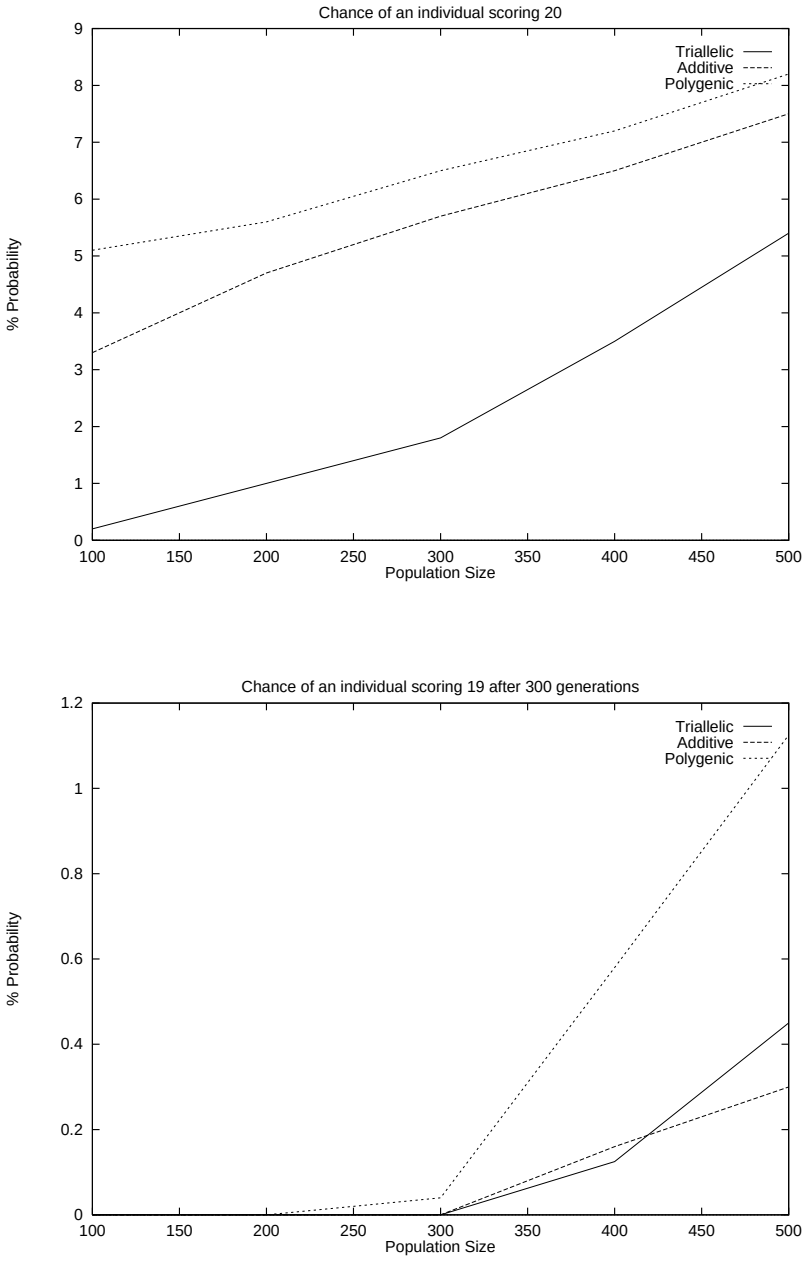
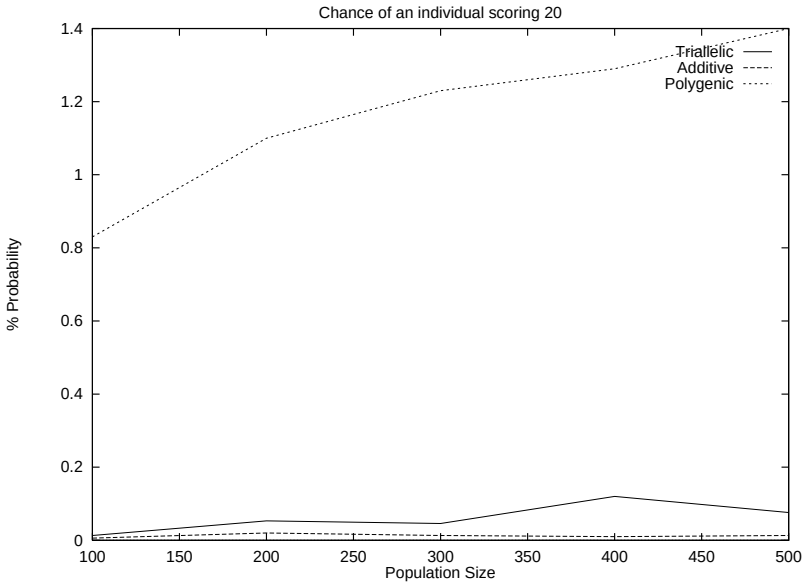


Figure 6.11: A comparison of polygenic inheritance and the other two diploidy schemes with an environment change every 25 generations.



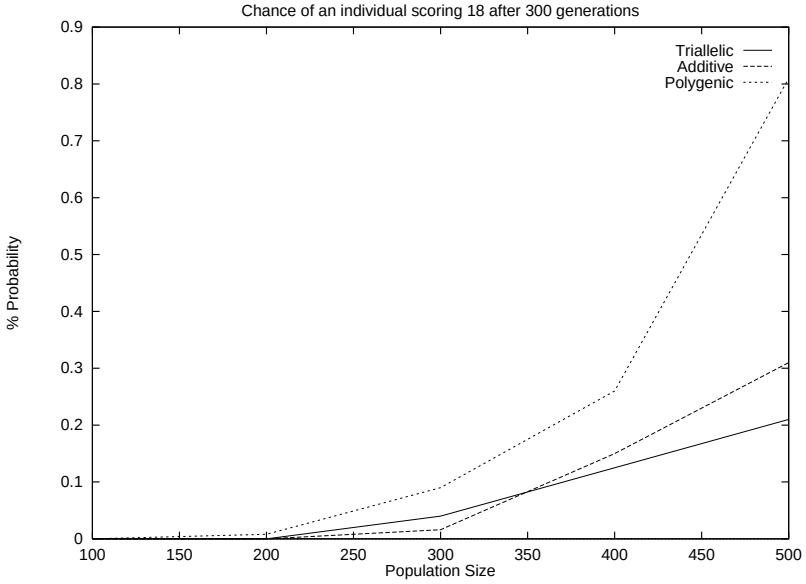


Figure 6.12: A comparison of polygenic inheritance and the other two diploidy schemes with an environment change every 25 generations.

Chapter 7

GPRobots and GPTeams

7.1 Introduction

This chapter presents two simulations, ***GPRobots*** and ***GPTeams***. GPRobots is a new, competition oriented benchmark for GP, where control programs are evolved for robots. These are then executed in a simulated *real time* environment, where execution time for instructions varies with instruction complexity. The competitive nature of GPRobots allows the direct comparison of different evolutionary approaches, using two or more *best-of-run* individuals from various simulations.

GPTeams, the second simulation in this chapter, introduces the idea of using GP to produce *event driven* programs, using a novel technique where a population of main control programs are co-evolved with the callback functions they use. The callback functions, although competing against each other on an evolutionary scale, co-operate in teams organised by the main program at a local level.

The experiments produce a wide array of behaviours from individuals, from pacifist to violent, from parasitic to general.

7.2 Programming competitions

Games of competition are often used as a test of programming skill [Dewdney, 1984] [Rognlie, 1995] [Timin, 1995][Schick, 1995]. These comparisons usually take the form of a competition betwe en different programmers who try to outwit each other in the game.

Often, the inner workings of the programs entered into these competitions are kept secret and the source is not publicly available. Under certain circumstances an optimal solution is known, as in the well known Prisoner's Dilemma game, so it is not really useful to directly compare different development methodologies. This paper develops a similar test for GP, where individuals produced by different evolutionary approaches may be tested against each other.

Competition, particularly in the form of co-evolution, [Siegel, 1994] [Koza, 1992] [Jannick, 1994] in GP is not uncommon, although competition between individuals in a single population has also been investigated [Reynolds, 1994b]. Invariably, these are one-on-one competitions[1], but GPRobots allows more than two individuals in a tournament, and, as points are awarded for longevity relative to the others in a tournament, it is possible for a number of different strategies to emerge, e.g. a pacifist robot who avoids combat may survive longer than a very aggressive robot who constantly attacks.

In scenarios where individuals are pitted against each other, usually a move is determined for each, and then executed, implying that each move takes the same length of time to execute. GPRobots differs from this, in that all instructions take an argument, e.g. how far to move forward, how many degrees to turn left etc.. We share the view of [Reynolds, 1994a] that more complex or longer moves should take more than one time unit to execute.

A relatively new programming paradigm is that of *event driven programming*. Event driven programming differs from most paradigms in that instead of writing a sequential program, a programmer writes functions to deal with particular events that may (or may not) happen during the running of a program. Each function is registered with an *event handler* which calls each function as appropriate. It is possible to prioritise these events, so an event with a higher priority may interrupt a lower priority event.

In practice, event driven programming is really interrupt driven programming at a user rather than hardware level, in that events can interrupt the normal flow of a program, and, in some cases, pre-empt each other.

[1] Although some researchers, notably [Siegel, 1994] and [Hillis, 1989], pit a single individual from one population against several from another, it does fight each opponent singly.

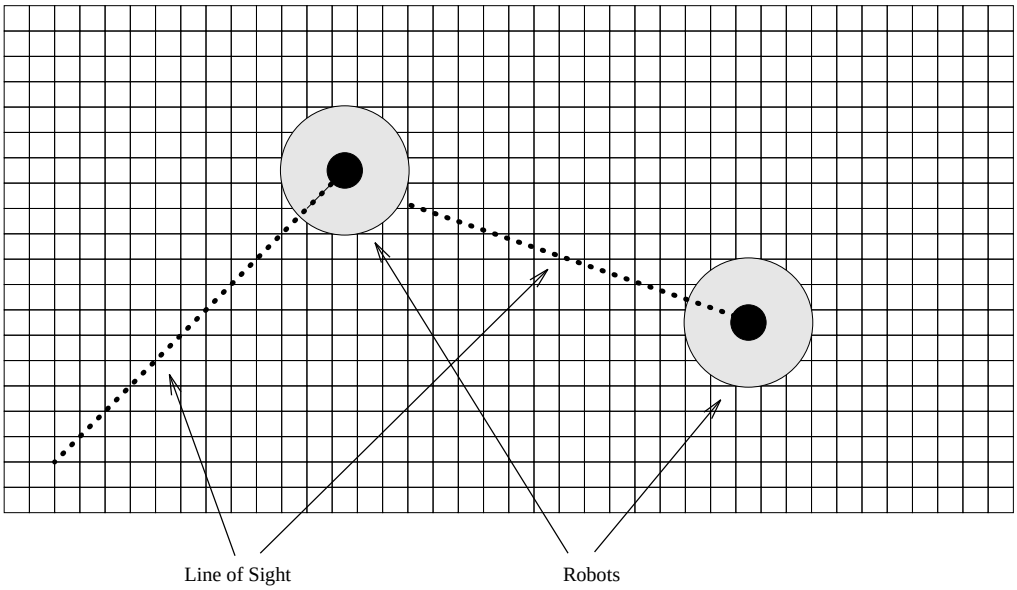


Figure 7.1: Simulated robot on the arena. (Scale 1:5)

7.3 GPRobots Tournaments

Most previous work with GP involving competition between individuals has been concerned with one-on-one competitions. GPRobots conducts its tournaments between two *or more* robots - three robots in the simulations presented here. Robots score points for the amount of other robots killed while they themselves are alive. The arena in which tournaments take place measures 75x20 squares, with robots taking up 3x3 squares.

Each robot starts with 100 energy units. Units are lost by shooting at other robots, by being hit by another robot's shot, or by crashing into another robot. Energy is *not* lost by colliding with a wall of the arena.

The term *round* is used to denote one time step, and the term *turn* to denote the action a robot wants to take. As will be seen, some turns require multiple rounds to achieve.

7.3.1 Robot Actions

As stated above, it is common practice when evolving control programs with GP (such as the well known *Artificial Ant* problem[Koza, 1992]), that the actions involve the robot moving ahead one square, going back one square, turning left 90° etc.

The most comprehensive work in this area with GP has been that of Reynolds [Reynolds, 1994b] and his is the only work to permit robots to be continuous in orientation.

Action	Maximum per round
Left	15° rotation
Right	15° rotation
Forward	Up to 3 squares
Reverse	Up to 3 squares
Fire	One shot, plus an extra round for the gun to cool down

Table 7.1: Limits on the actions available to the robots

GPRobots adopts this approach, and also assumes that robots accelerate if they are traveling continuously in the same direction. All but one of the robot instructions are of the form: **(action argument)**, and all return a value indicative of the success or otherwise of the instruction.

Robots may turn **left** or **right**, and may move **ahead** or **reverse**, and, of course, may **fire**. Left and right take an argument of degrees to turn, while ahead and reverse the number of squares to move. The fire instruction doesn't take any arguments.

GPRobots is a simulated real time environment. For this reason, there was a limit imposed on what a robot could do in a single round. The limits were as in table 7.1.

Moving forward or backwards, although subjected to a maximum of three squares per round, depended also on the robot's current speed. A robot could accelerate at a rate of one square per round until it reached the maximum speed. If the robot attempted another move, e.g. firing or turning, it would stop first.

7.3.2 Robot sensors

Robots have available a number of sensors to them which provide information about the arena. These include a sensor which reports a robot's involvement in a collision and a sensor which reports the robot being struck by a missile. These sensors return the number of degrees which the robot must turn to face either the other robot involved in the collision or the robot who fired the offending missile. Robots can see other robots who are directly in their line of vision, and can determine how far away they are.

For the purpose of evolving robots, conditional commands which directly access the contents of the sensors such as **(IF-Collide** x y**)** could be designed. It is the view of the author that functions such as these are not an integral part of the robot structure, no

more than say, a mathematical function would be, and for this reason, are not considered to be "hardwired" into the robot. All the parameters used in this particular simulation are shown in table 7.3.

This is not to say that the programs controlling the robot cannot use these, or, indeed, any other reasonable functions that an implementor wishes to use.

7.4 Fitness Function

One of the reasons that games such as this are so popular as a test of programming skill and strategy is that there is usually no obvious optimal strategy[2]. While this makes the benchmark more interesting and useful, it complicates the fitness function.

One approach is to hand code a few expert robots, and use these to test the *best-individual* produced by a GP run[D'Haeseleer, 1994][3]. However, this would seem to suggest that the ability of GP to produce individuals is determined to some extent on the ability of the implementor to produce hand coded solutions to the very problem that is to be solved, a situation we would rather avoid.

Given that there is no obvious fitness measurement, the only other choice is to measure an individual's fitness relative to other individuals. Problems such as this involving competition between individuals lend themselves to this sort of measurement to some degree. Only to some degree because, although it is easy to compare k individuals, where k is the number involved in a tournament, it is another matter to compare N individuals, where N is the size of the population.

A variety of these methods were reported on by [Reynolds, 1994a] and are summarized in table 7.2.

This paper uses a method which employs both the *New versus several* [Reynolds, 1994a] and the *New versus neighbour* [D'Haeseleer, 1994]. Individuals live in a one dimensional neighbourhood, and are chosen in groups of k for tournaments.

Individuals in this particular simulation of GPRobots are arranged on a one di-

[2] While this is not strictly true in the case of the Prisoner's Dilemma game, it is the modelling of games such as those mentioned in the introduction that we are most concerned with.

[3] D'haeseleer and Bluming's work also used the problem of robots fighting each other in an arena, but they were more concerned with the effects of locality than the evolution of interesting behaviours as is the case in this paper

Competition	Matches per ind.
New versus all	$n-1$
New versus several	k
Knockout tournament	$\log_2 n$
New versus best	1
New versus new	1
New versus neighbour	1

Table 7.2: A number of different approaches to competitive fitness measures.

Objective:	Evolve an individual to control a robot in the arena.
Terminal set:	The random constant $\Re_{integer}$
Function set:	$+, -, *, \%$, IF-Collide, IF-Missile, IF-See, FIRE
Fitness cases:	Three fights against neighbouring robots
Raw fitness:	The score of a robot over the three fights
Hits:	Not applicable
Wrapper:	None
Parameters:	M=100, G=25

Table 7.3: A Koza-style tableau summarizing the control parameters for the GPRobots simulation.

mensional toroidal grid, in a manner similar to [Collins, 1992]. Rather arbitrarily, it was decided to implement overlapping demes each with three individuals. This resulted in each individual being involved in three fights being awarded one point for each robot killed while they are still alive. Their fitness is the sum of their points over three games. A robot who comes last in all three fights amasses three points.

Crossover is implemented by using these demes, in a similar manner to [Collins, 1992], except, that if the individual to be replaced has the highest fitness in the deme, then that individual is selected for reproduction instead.

The reader should be aware that all implementation decisions, such as those involving the spatial structure, points allocation, tournament size etc. are peculiar to this attempt to evolve interesting robots. Other approaches could yield vastly different results. In fact, it is the hope of the author that other implementations will be examined, and it would cause neither surprise nor disappointment if a different implementation yielded superior results.

7.5 Preliminary Results

Normally, a GP run reports a *best-of-run* individual who performed better than any other, and when reporting on results, one can often quote statistics showing the percentage of successful runs / average generations to success etc. Unfortunately, the success or otherwise of a run in this case isn't quite so clear-cut. Firstly, there is no absolute "solution" to the problem. The second problem is that the performance of an individual can be measured relative to nearby individuals only. Both these problems were also encountered by [D'Haeseleer, 1994], mentioned above, and their solution was to test potential *best-of-run* individuals against a group of handcoded individuals.

In this chapter, we would rather not rely on the ability of the implementor to gauge the quality of the results, and so the discussion below will concentrate on the more interesting individuals who appeared and the strategies they adopted. These individuals were chosen for their longevity, as an individual who was reproduced several times is clearly employing some sort of useful strategy. Individuals who were rated "interesting" using this criterion were then set fighting against each other, and in this way, a *best-of-run* individual was discovered.

7.5.1 Sloths, Triggers and Spinners

There were two surprising aspects of the results. The first was how few of the successful individuals ever utilised the **AHEA** or **REV** commands, most individuals being content to remain stationary, turning in one position whilst searching for opponents. The second surprise was how many of those robots selected as interesting actually did nothing. The robots had available to them a number of mathematical functions, and it wasn't uncommon for these individuals to consist of a simple mathematical expression, such as

$$(+ (* \ 5 \ 2) \ 1)$$

These individuals, dubbed *Sloths*, often appeared near each other in the population structure, very often on either side of another common individual, which we name *Trigger*, who simply fires every turn. Unless a Trigger was very fortunate, it usually used up all its energy firing and died quite early, while the Sloths simply sat out the hostilities until the time limit

for a fight ran out. The Sloths then, operate as a form of parasite, living off the behaviour of the Triggers, who, acting as "hosts" of sorts, provided them with an easy life.

It is reasonable to assume that with a bigger deme size, Sloths would find it more difficult to find the right conditions to allow their survival. For the sake of consistency though, this chapter will not try an experiment with a larger deme.

Rather reassuringly, another type of individual also appeared, who, while not quite as numerous as the Sloths, were more successful. These individuals are known as *Spinners*. The common trait in Spinners is that they turn slowly, scanning the grid as they do so, until they see another robot. Upon seeing another robot they fire continuously until the robot is either dead or runs away. These individuals are normally of the form

(IF-See (FIRE) (LEFT 2))

Spinners represented the best strategy, and, in direct competition with a Sloth, a Spinner would almost always win. In certain cases, however, in a competition involving two Spinners and one Sloth, the Sloth won the competition because the two Spinners found and attacked each other.

7.5.2 The best of run

To find the best-of-run individual, several (up to ten) individuals were selected from a run based solely on their longevity. These ten were then pitted against each other in another tournament, with the winner of that being declared the *best-of-run* individual. This then automated the selection of such an individual from a run.

The best individual to appear was called a *Smart Spinner* and was as follows

(IF-See (FIRE) (LEFT (IF-Missile (bearing_of_enemy) 5)))

Like its ancestor, the Smart Spinner turns slowly, looking for enemies, but, if the Smart Spinner is hit by a missile, it turns to face the offender. In contrast, the ordinary Spinners do not react to being shot, instead continue turning and searching.

It is suggested that different approaches to evolving robots could be compared in a similar manner, with perhaps the best two or three from each approach being entered into

a tournament with each other.

Only a small, albeit representative, selection of individuals is presented above. Other individuals also appeared, such as *Panickers* who turned and fled upon being shot or involved in a collision, or *Corners*, individuals who backed themselves into a wall or corner and tried to wait out the fight there. A number of other individuals are shown in Appendix A.

7.6 GPTeams - Evolving event driven programs

A relatively new programming paradigm is *Event Driven Programming*, used extensively when programming in GUI environments such as Xview, Motif etc. Events are stimuli which cause the program to take notice, and such notice is given by an *event handler*. Events may be a user clicking a button on a menu, a user closing a window or anything else that may affect the operation of a program.

Generally, a function, a *callback function*, is written for each event and then registered with the event handler. As events occur the appropriate function is called. In the case of two events happening at once, or an event occurring while the callback function of another event is executing, the events may be queued, or, in the case of prioritised events, the highest priority event may interrupt other events.

As mentioned in the introduction, event driven programming is really interrupt driven programming at a higher level. As such, there is an enormous number of real world applications, from washing machines to microprocessors, that can utilise this type of programming. It is suggested that Genetic Programming is suitable for the automatic generation of event driven programs, and the remainder of the chapter shows how, using a combination of competition, co-evolution and co-operation, GP can evolve event driven programs.

7.6.1 Evolving Events

Multiple co-operating populations have been used in several of the previous chapters as well as in the literature[Ryan, 1994b] [Ryan, 1995c] [Siegel, 1994]. In this case, two populations are used. One population contains individuals who will be used as callback

functions(CBs), while the other will contain the main programs(MPs) which decide when it is appropriate to call functions provided by the CBs. Each MP is made up of a number of *event registers* which register an event with the event handler. Each event is registered as follows:

(Reg_Event *Condition Priority Code*)

Where *Condition* is the condition under which the callback function is to be executed, *Priority* is the priority of this event, and *Code* is a pointer to the callback function.

Evolving event driven programming involves both competition and co-evolution. Individuals from the MP population compete with each other by organising CB individuals into co-operating teams. The co-operating teams then compete against each other. Each MP robot contains pointers into the CB population to call individuals when appropriate. The individual pointed to by *ord_event* is called repeatedly until interrupted (assuming they have a higher priority) by one of the other functions.

7.6.2 Choosing Events

Although it would be possible, and indeed, more useful, to allow evolution to choose under what conditions an event should be registered, this chapter will only consider the most simple model, which consists of a fixed set of events. In this case, every program consists of four registered events, and it is the responsibility of the event handler to monitor the robot's sensors and call the corresponding functions. For the purposes of this chapter, the MP individuals contain information for four events which are called under the following circumstances.

- The robot is involved in a collision (Cld_event)
- The robot is struck by a missile (Hit_event)
- The robot sees another robot (See_event)
- No other event is active (Ord_event)

An MP individual contains a priority level for each of these events, as well as a pointer into the CB population to the individual to be used for that event. When a robot's

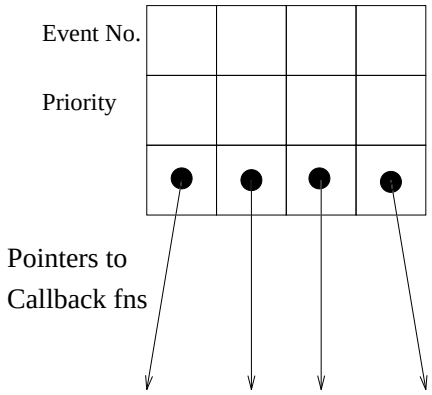


Figure 7.2: Structure of an individual in the MP population.

sensor registers information, such as being struck by a missile, this information is retained for a number of time steps before being set to a blank value. An event can only interrupt another one if it has a higher priority, and, if an event is interrupted its callback function will only be called again if the condition that originally caused it still holds.

The individuals in the MP population were arranged in a similar demetic structure to the individuals in the original experiments, and fights / reproduction were arranged in the same way. The individuals in the CB population were also put into a demetic structure, which made it a trivial task for the MP population to keep track of what individuals they were pointing at. Again, crossover and reproduction was implemented using a deme size of three.

Each fight was conducted as follows:

- Select MP robots to fight.
- Using CBs pointed to by MPs, resolve conflict.
- Award points to MPs in the same way as before.
- Each CB used by an MP gets the same reward. The rewards given are as follows:

  1. +3 for each CB.
  2. +1 for each CB.
  3. -1 for each CB.

It is possible for a CB individual to be pointed at by two or more MP individuals in the same fight, or even used for several events by the same MP individual, so points were awarded for *each time* a CB individual was pointed at.

7.6.3 Early Results

The initial results for this simulation were quite disappointing, with the more successful members of the MP population using only one or two callback functions in a sensible manner, i.e. an individual fired if it saw another one, but did nothing if it was hit itself, or if involved in a collision. It was very rare that an individual was produced who made proper use of the event structure.

Another disappointment was the difficulty encountered when comparing the individuals - sometimes an individual who reacted appropriately upon seeing another robot performed better than an individual who took evasive action upon being shot, but other times it did not.

There were two reasons for these disappointing results. Firstly, the MP population became very stable extremely quickly, so newly created individuals in the CB population often didn't have an MP pointing at them, and so didn't even get tested. The second reason was that the CB population was trying to maintain a balance of radically different individuals - crossover between an individual successfully used as callback function for an MP being shot and an individual called when an MP didn't have any other active events often produced unviable individuals.

The solution to both of these problems was to divide up the CB population into several sub-populations, one for each event.

7.6.4 Multiple Callback Populations

This decision resulted in four CB populations. An MP individual now pointed at one individual in each population. Again, it was unusual for an MP individual to be produced that used the event structure properly, most using one or two good events, but pointing at less useful ones also.

One distinct advantage of this method, however, is that one could select good per-

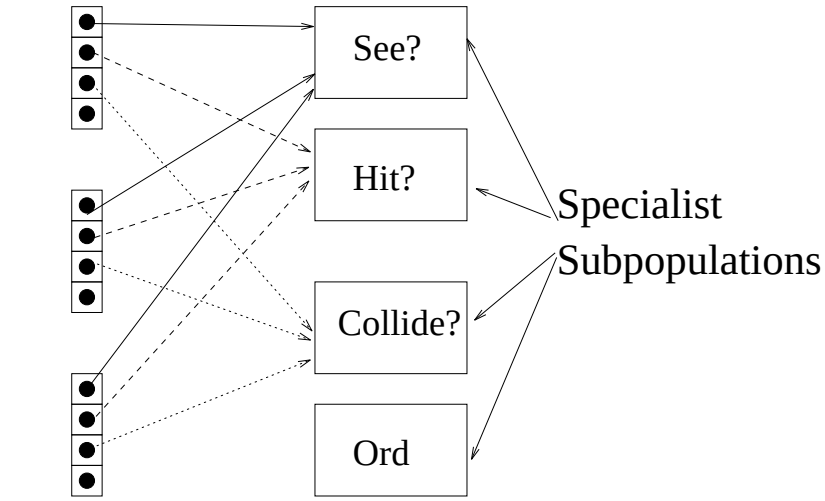


Figure 7.3: Multiple Callback populations.

forming individuals from the CB populations in a similar manner to the original simulation - in this case there was no doubt which individuals were a good choice for each event. Like the original simulation, GP produced a number of different, but equally interesting individuals for each event. Individuals for the *See_event* for example, ranged from the aggressive

(PN2 (REV (FIRE)) (FIRE))

or

(IF-Collide (REV 2) (FIRE))

to the passive, who shied away from any contact with other robots

(PN2 (REV 5) (RT 1))

and, to the cautious, who only fired if it was absolutely necessary

(IF-Collide[4] (FIRE) 1)

Similarly, each event produced individuals who adopted interesting and different approaches. For the *Cld_event* individuals either moved away from the collision very quickly,

[4] Although it may appear strange that the callback function for the *See_event* investigates a sensor associated with another event, this often occurred, particularly in the case of a high-priority event.

turned to face the offending individual, or simply started firing. *Hit_event* produced individuals remarkably similar to *Cld_event*, but, as both these events cover somewhat similar circumstances this was not too surprising.

The final event is the *Ord_event* and these individuals fell into two categories noted earlier - Sloths and Spinners. When no other events were called, individuals either scanned the arena slowly, or simply waited for something to happen.

It should be noted that only the successful individuals are reported in this chapter. Other individuals also appeared, who would warrant interest because of their unusual and often amusing tactics - individuals who tried to ram others for example, or those turned their backs on individuals who shot them.

The problems of skewed fitness measures still existed, however, with several individuals not being tested. This was due to no MP individuals pointing at them. Other individual had an abundance of MP individuals pointing at them even though they themselves didn't do much to help that individual. These parasitic individuals tended to take over large portions of the population and often survived from Generation 0 until the end of the run. One possible solution to this is to calculate the fitness of the CB individuals based on their *average* performance over all the fights they are involved in, which would reduce an individual scoring highly by simply being involved in a lot of fights.

7.6.5 Digesting the results

At this point we have a number of genuinely useful individuals for each event, but no clear indication as to how they should be combined, or what priority should be associated with each event. As mentioned above, the MP individuals from the runs that produced these individuals, although serving to provide a fitness measure, are not particularly useful now.

The solution to this problem is to run the simulation a second time, in a slightly modified form. This time, the CB populations are seeded with individuals produced in the first run, and no crossover takes place in those populations. This time, only the MP population is evolving, and "interesting" individuals from these simulations would be candidates for the *best-of-run* individual, which can be selected in the same manner as before.

There was an enormous variety of individuals produced who were more than able

to hold their own in the arena. Most individuals reacted in some way to each event, the most common being as follows :

- See_event : Fire or run away.
- Hit_event : Turn to face opponent, turn slowly or run away.
- Cld_event : Turn to face opponent, turn slowly or run away.
- Ord_event : Scan the arena, run around or do nothing.

The best individual to appear was very simple, but quite aggressive, and, once it found an opponent would not stop firing until the opponent either got out of the way or died. This robot did not win every fight it was involved in, but in the tournament at the end of the run it performed the best. In the fights it did lose, it usually was because it backed into a wall, and, while being shot by some individual it couldn't see, kept trying to reverse.

(Reg_event See_event 68 (FIRE))

(Reg_event Cld_event 37 (RIGHT 5))

(Reg_event Hit_event 23 (REV 1))

(Reg_event Ord_event 1 (RIGHT 3))

A number of other individuals also appeared, using various combinations of the above strategies, with different priorities. Few of the successful robots did nothing when there were no events happening, and those that did were often struck several times before they reacted to an attack. The next best strategy was to shoot only when not under attack. Individuals adopting this strategy had a low priority for *See_event*, choosing instead to take evasive action if they came under attack.

7.7 Best-of-Chapter Results

The big question of course, is what is the best individual produced by any of the simulations? In the end it came down to a competition between those in Appendix A, the

best three individuals that appeared in each simulation being chosen to fight in a similar demetic tournament to those used in the simulations. The best four individuals (three of whom were event-driven individuals) were then put into a round-robin tournament together, with the eventual winner being the individual described in section 7.6.5.

Is this individual the best possible? Probably not. There is an enormous amount of tweaking that can be done, and a huge variety of other approaches to evolving an individual - from varying the deme size, to the fight size, to the kind of functions available to the robots. The list is endless.

7.8 Conclusion and Future Work

This chapter presents a new benchmark problem for GP. Unlike many benchmarks, there is no known optimal solution, and, because the evolved controller program is abstracted away from the robot to be controlled, there are a huge number of ways to approach the problem. Every part of the problem, from the functions to the tournament size, to crossover implementations, can be changed. These changes don't affect the main problem in any way, but could well produce interesting and viable behaviours

Due to the nature of the problem, it is easy to compare two (or more) different solutions against each other, and to test the performance of GP under varying conditions. It is even possible that sometime in the future there could be a GPRobot tournament, as discussed in the recent AAAI Symposium on Genetic Programming, [Ryan, 1995a] where individuals bred under different conditions could be tested against each other. Such is the particularly open nature of those in the GP community, that it is hoped that if such a tournament does take place, unlike competitions in games such as those mentioned earlier, *all* information about a robot be made available, including code and the method used to produce the individual. In the event of such a tournament taking place, it is likely that the individual in the previous section can be bettered - and that is the whole point of this benchmark, that individuals from various approaches can be directly compared, or even watched on screen.

This chapter also introduces the notion of teams of specialist individuals in GP and the evolution of event driven programs, where teams of individuals co-operate with

each other in competition with other teams. This method produced the best individual found, and the area warrants further investigation, particularly in the areas of evolving the conditions under which an event is deemed to have occurred, which would allow the system to become truly automatic. Potentially, event driven programming has an enormous number of applications areas. This chapter has served to show how programs requiring any number of events may be evolved.

Increasing the number of events / functions available to a program while still ensuring that specialisation can take place helps GP move ever closer to producing large-scale programs. If desired, an extra criterion can be imposed on a population of main programs simply by creating an extra sub-population. This allows the main program to make use of this extra function, but without having to maintain the genes necessary.

The event driven approach can even be used in applications that do not employ events, Appendix A shows some individuals rewritten in this way. The only difference in individuals rewritten is that they no longer employ pre-emption.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

This thesis has presented a number of improvements to EAs by taking further examples from nature. The basic approach has been to identify a problem and examine how natural biology would solve it. In some cases, example has been taken from nature itself, while in others, example was taken from methods used by biologists.

The performance of EAs is subject to a number of parameters, many of which must be chosen with little knowledge of how they will affect the populations. It has been shown that the more control that is left to evolution, the better the population as a whole will perform. A number of different problems were examined in the course of this thesis, and all the approaches described have been adopted almost directly from nature. In all cases examined in this thesis, the newer, naturally-inspired methods, outperformed a number of existing methods.

A single population is not always sufficient to solve a problem, so many of the simulations in this thesis used multiple populations. The degree of co-operation between these populations varied. In some cases each population concentrated on a particular part of the problem, before communicating its knowledge with the other population. In other cases, each population avoided the others, and sought out its own environmental niche. In the final case, each population concentrated on solving a single part of the problem, but its true worth could be appreciated only when it co-operated with individuals from another population.

Chapter 3 introduced the notion of *ecotypes* in the Pygmy Algorithm. Two behaviourally different, yet biologically compatible, groups of individuals were maintained, each with a different fitness function. Between the two groups, sufficient selection pressure was generated to force the population as a whole to evolve and produce individuals who excelled at both fitness functions.

The benefits of inbreeding were demonstrated also in Chapter 3. By allowing evolution itself to control the level of inbreeding through a meta-GA, the performance of the Pygmy Algorithm was improved still further.

Chapter 4 examined the problem of multi-modal function optimisation. The Pygmy Algorithm was modified to become the Races Genetic Algorithm. RGA used several races to spread a population across a search landscape, with each race concentrating on a different part of the search space and subject to its own fitness function. RGA was shown to be superior to several other methods for a number of one and two dimensional problems.

Genetic Programming was applied to a new problem area in Chapter 5. The Paragen system is a new method for the automatic parallelisation of serial programs. The Paragen system was built using the techniques developed in earlier chapters. By allowing different parts of the population concentrate on various parts of the problem, for example, the generation of parallel code and the generation of code functionally identical to the original program, a parallel version of an initial, serial program could be generated.

Chapter 6 examined the genetic structure used in Evolutionary Algorithms. Diploidy has not enjoyed much use in EAs because of difficulties associated with its implementation, both for binary genes and particularly with high level structures. This is unfortunate because diploidy would allow populations to adapt more quickly, even in cases where the environment or training set can change. Two models, the Degree of Oneness, and the Degree of *N*ness were presented, along with the notion of polygenic inheritence. These diploid models were shown to outperform previous approaches to diploidy. The implementations of diploidy presented in this chapter were all adopted from those observed in natural biology.

A new benchmark problem for GP was described in Chapter 7. This benchmark has no known optimal strategy and its competitive nature permits the direct comparison of two or more individuals. In evolving a solution to the problem, GP was applied to the generation of event driven programs. This involved dividing a problem up to be solved by

co-operating teams of individuals from specialist populations.

This thesis has presented several extensions to traditional Evolutionary Algorithms, and successfully applied it to two new, real world areas. Practically all the improvements presented either occur in natural biology, or were inspired by natural phenomena.

8.2 Future Directions

The use of multiple populations to solve a single problem has been shown to greatly enhance the power of an EA. Using multiple populations on a massively multimodal function[Goldberg, 1992] where the populations could nomadically travel the landscape searching for peaks could allow small populations to be applied to large problems.

EAs are being applied to increasingly complex problems, often involving enormous amounts of testcases. The new diploidy schemes introduced in Chapter 6 effectively give a genetic memory to a population. This memory allows the varying of testcases without the population losing gene combinations that were created to cope with earlier testcases. This opens up a new avenues of interest, that of efficiently varying the testcases.

The Paragen system is currently undergoing further development[Walsh, 1996]. Genetic Programming has been shown to be extremely suitable to the difficult problem of automatic parallelisation. The eventual goal is to use Genetic Programming to produce parallel programs from scratch, in a similar manner to the way it is now used to produce serial programs.

GPRobots presents a new benchmark for GP. There are certainly other viable strategies for robots, and, with the release of the code for GPRobots in to the public domain, it is hoped that other researchers will produce interesting individuals through different methods.

This thesis has shown how EAs can be improved by looking to nature. EAs are still in their infancy, while nature and biology have been studied for centuries. As such, there is an enormous wealth of information readily available. If computer scientests are willing to use this existing knowledge, the combined effort should be of mutual benefit.

Appendix A

Sample Individuals from GPRobots

A.1 Best-of-Chapter Candidates

The following individuals were considered when choosing the Best-of-Paper individual.

A.1.1 From the initial simulations

Spinners :

- **(IF-See (FIRE) (LEFT 2))**

- **(IF-See (FIRE) (LEFT (IF-Missile (bearing_of_enemy) 5)))**

Panicker

- **(IF-Hit (AHEA (RIGHT 5)) 1)**

A.1.2 Event Driven Individuals

- **(Reg_event See_event 96 (FIRE))**

(Reg_event Cld_event 73 (PN3 (LEFT 5) (RIGHT 5) (AHEA (RIGHT 5))))

(Reg_event Hit_event 23 (AHEA (LEFT (AHEA (LEFT 2)))))

(Reg_event Ord_event 1 (LEFT 3))

- **(Reg_event See_event 68 (FIRE))**

(Reg_event Cld_event 37 (RIGHT 5))

(Reg_event Hit_event 23 (REV 1))

(Reg_event Ord_event 1 (RIGHT 3))

- **(Reg_event See_event 34 (PN3 (REV (FIRE)) (FIRE) (REV 1)))**

(Reg_event Cld_event 23 (RIGHT 3))

(Reg_event Hit_event 82 (REV 1))

(Reg_event Ord_event 17 (RIGHT 3))

A.2 Event Driven Individuals rewritten without events

The Event Driven individuals shown in the previous are rewritten below. Functionally, they are almost identical in either form, in that the priorities remain intact, but it is no longer possible for pre-emption to take place.

```
(IF-See (FIRE)
    (IF-Cld (PN3 (LEFT 5) (RIGHT 5) (AHEA (RIGHT 5) ) )
        (IF-Missile (AHEA (LEFT (AHEA (LEFT 2) ) ) )
            (LEFT 3) ) ) )

(IF-See (FIRE)
    (IF-Cld (RIGHT 5)
        (IF-Missile (REV 1)
            (RIGHT 3) ) ) )

(IF-Missile (REV 1)
    (IF-Hit (PN3 (REV (FIRE)) (FIRE) (REV 1))
        (IF-Cld (RIGHT 3)
            (RIGHT 3) ) ) )
```

Appendix B

Published Work

A number of papers describing work in this thesis have been published, their full references are below:

Ryan C. (1994) Pygmies and Civil Servants. In *Advances in Genetic Programing.* Ed: K. Kinnear. MIT Press.

Ryan C. (1994) The Degree of Oneness. In the proceedings of the European Conference of Artificial Intelligence Workshop on Genetic Algorithms.

Ryan C. (1994) Racial Harmony in Genetic Algorithms. In the prodeedings of KI workshop on Genetic Algorithms.

Ryan C. (1995) Niche and Species Formation in Genetic Algorithms. In *Practical Handbook of Genetic Algorithms.* Ed: L Chambers. CRC Press.

Ryan C. (1995) Racial Harmony and Function Optimization in Genetic Algorithms - The Races Genetic Algorithm. In *The Proceedings of Evolutionary Programming.* The MIT Press.

Ryan C. and Walsh P. (1995) Automatic conversion of programs from serial to

parallel - the Paragen System. In *The Proceedings of ParCo' 95*. Elsevier Press.

Ryan C. (1995) GPRobots and GPTeams - Competition, co-evolution and co-operation in Genetic Programming. In *Working notes from the AAAI Fall Symposium series on Genetic Programming.*

Bibliography

[Angeline, 1993] Angeline, P. (1993). *Evolutionary Algorithms and Emergent Intelligence.* PhD thesis, Ohio State University.

[Baeck, 1992] Baeck, T. (1992). Self-adaptation in genetic algorithms. In *First European Conference on Artificial Life.*

[Banarjee, 1993] Banarjee, U. (1993). Automatic program parallelization. *Proceedings of IEEE.*

[Beasley, 1993] Beasley, D. (1993). A sequential niche technique for multimodal function optmisation. *Evolutionary Computation*, 2:101–125.

[Blume, 1994] Blume, W. (1994). Automatic detection of parallelism. *IEEE Parallel and Distributed Technology.*

[Braunl, 1993] Braunl, T. (1993). *Automatic Parallelisation and Vectorisation.* Prentice-Hall.

[Brindle, 1981] Brindle, A. (1981). *Genetic Algorithms for function optimization.* PhD thesis, University of Alberta.

[Collins, 1992] Collins, R. (1992). *Studies in Artificial Life.* PhD thesis, University of California, Los Angeles.

[Darwin, 1859] Darwin, C. (1859). *On the Origin of the Species.* Murray.

[Deb, 1989] Deb, K. (1989). Genetic algorithms in multimodal function optimization. Master's thesis, University of Alabama.

[Deb and Goldberg, 1989] Deb, K. and Goldberg, D. (1989). An investigation of niche and species formation. In *ICGA3*.

[DeJong, 1975] DeJong, K. (1975). *An analysis of the behaviour of a class of genetic adaptive systems*. PhD thesis, University of Michigan.

[Dewdney, 1984] Dewdney, A. (1984). In a game called core wars hostile programs engage in a battle of bits. *Scientific American*, pages 14–23.

[D'Haeseleer, 1994] D'Haeseleer, P. (1994). Effects of locality in individual and population evolution. In *Advances in Genetic Programming*. MIT Press.

[Eshelman, 1991] Eshelman, L. (1991). Preventing premature convergence in genetic algorithms by preventing incest. In *ICGA4*.

[Fogel and Atmar, 1992] Fogel, D. and Atmar, J., editors (1992). *Proceedings of the first annual conference on evolutionary programming*.

[Fogel et al., 1966] Fogel, L., Owens, J., and Walsh, M. (1966). *Artificial Intelligence through simulated evolution*. Wiley and Sons.

[Goldberg, 1989a] Goldberg, D. (1989a). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley.

[Goldberg, 1989b] Goldberg, D. (1989b). Zen and the art of genetic algorithms. In *ICGA3*.

[Goldberg, 1992] Goldberg, D. (1992). Massive multimodality, deception and genetic algorithms. Technical Report 92005, University of Illinois at Urbana-Champaign.

[Goldberg and Richardson, 1987] Goldberg, D. and Richardson, J. (1987). Genetic algorithms with sharing for multimodal function optmisation. In *ICGA2*.

[Goldberg and Smith, 1987] Goldberg, D. and Smith, R. (1987). Nonstationary function optimization using genetic algorithms with dominance and diploidy. In *ICGA2*.

[Grefenstette, 1986] Grefenstette, J. (1986). Optimisation of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man and Cybernetics*.

[Hillis, 1989] Hillis, D. (1989). Coevolving parasites improves simulated evolution as an optimisation procedure. In *ALIFE II*.

[Holland, 1975] Holland, J. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press.

[Horn et al., 1994] Horn, J., Nafpliotis, N., and Goldberg, D. (1994). A niched pareto genetic algorithm for multi-objective optmization. In *First IEEE Conference on Evolutionary Computation*.

[Jannick, 1994] Jannick, J. (1994). Cracking and co-evolving randomizers. In *Advances in Genetic Programming*. MIT Press.

[Kinnear, 1993] Kinnear, K. (1993). Generality and difficulty in gp : Evolving a sort. In *ICGA 5*.

[Knuth, 1973] Knuth, D. (1973). *The art of Computer Programming*. Addison Wesley.

[Koza, 1992] Koza, J. (1992). *Genetic Programming*. MIT Press.

[Langdon, 1995] Langdon, W. (1995). Pareto, population partitioning, price and genetic programming. Technical report, University College London.

[Pai, 1985] Pai, A. (1985). *Foundations of Genetics : A science for society*. McGraw-Hill.

[Parkins, 1979] Parkins, D. (1979). *An introduction to evolutionary genetics*. Edward Arnold, London.

[Rechenberg, 1973] Rechenberg, I. (1973). *Evoltionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Fromman-Holzboog.

[Reynolds, 1994a] Reynolds, C. (1994a). Competition, coevolution and the game of tag. In *ALife IV*.

[Reynolds, 1994b] Reynolds, C. (1994b). Evolution of obstacle avoidance behaviour : using noise to promote robust solutions. In *Advances in Genetic Programming*. MIT Press.

[Rognlie, 1995] Rognlie, R. (1995). C++ robots. Available through anonymous ftp from ftp.netcom.com.

[Ryan, 1994a] Ryan, C. (1994a). The degree of oneness. In *ECAI94 Workshop on Genetic Algorithms*.

[Ryan, 1994b] Ryan, C. (1994b). Pygmies and civil servants. In *Advances in Genetic Programming*, chapter 11, pages 243–264. MIT Press.

[Ryan, 1994c] Ryan, C. (1994c). Racial harmony in genetic algorithms. In *KI'94 Workshop on Genetic Algorithms*.

[Ryan, 1995a] Ryan, C. (1995a). Gprobots and gpteams - competition, co-evolution and co-operation in genetic programming. In *AAAI Fall Symposium Series on Genetic Programming Working Notes*.

[Ryan, 1995b] Ryan, C. (1995b). Niche and species formation. In *Practical Handbook of Genetic Algorithms*, chapter 2, pages 57–74. CRC Press.

[Ryan, 1995c] Ryan, C. (1995c). Racial harmony and function optimization in genetic algorithms - the races genetic algorithm. In *EP95*.

[Ryan and Walsh, 1995] Ryan, C. and Walsh, P. (1995). Automatic conversion of programs from serial to parallel using genetic programming. In *Parallel Computing*.

[Schaffer, 1989] Schaffer, J. (1989). A study of control parameters affecting online performance of genetic algorithms for function optimisation. In *ICGA3*.

[Schick, 1995] Schick, B. (1995). Robowars. Available through anonymous ftp from psycfrnd.interaccess.com.

[Siegel, 1994] Siegel, E. (1994). Competitively evolving decision trees against fixed training cases for natural language processing. In *Advances in Genetic Programming*, chapter 19, pages 409–424. MIT PRess.

[Spears, 1995] Spears, W. (1995). Simple subpopulation schemes. In *EP95*.

[Strickberger, 1990] Strickberger, M. (1990). *Genetics*. Macmillan publishing company, 3rd edition.

[Suzuki, 1989] Suzuki, D. (1989). *Introduction to Genetics*. Freeman and Co., 4th edition.

[Syswerda, 1989] Syswerda, G. (1989). Uniform crossover in genetic algorithms. In *ICGA3*.

[Timin, 1995] Timin, M. (1995). Robot auto racing simulation. Available through anonymous ftp from magdonaz.mcaffee.com.

[Todd and Miller, 1991] Todd, P. and Miller, J. (1991). On the sympatric origin of the species : Mercurial mating in the quicksilver model. In *ICGA4*.

[Walsh, 1996] Walsh, P. (1996). Untitled phd thesis.

[Zima, 1990] Zima, H. (1990). *Supercompilers for parallel and vector computers.* ACM Press.