3 ADVANCED GENETIC PROGRAMMING TECHNIQUES

In the first section (3.1) of this chapter we position this work. While Section 2.3.2 has explained in detail general GP concepts which are used in later chapters, Sections 3.2 onwards describe more specialist techniques, some of which are introduced into GP. In later chapters, where we first use one of these concepts, we will refer back to the appropriate subsection within Section 2.3.2 or 3.2–3.9. (There is also a glossary in Appendix B).

3.1 BACKGROUND

Almost all GP work concentrates upon the evolution of simple functions, e.g. for classification. While Koza [Koza, 1992] has performed small demonstration experiments on recursion, iteration and evolvable memory and others have investigated a fourth, evolving programs containing primitives of more than one type, these four areas are largely unexplored even today. Progress on memory and iteration or recursion (and probably on all four areas) is required if genetic programming is to be capable of evolving general (i.e. Turing complete) programs. However our book concentrates upon the issues surrounding the evolution of the use of memory within GP. (A survey of other work on Evolving memory in GP is given in Section 7.4).

Simple indexed memory read an write functions (if iteration or recursion are also included) extend GP so that the language used by evolving programs is Turing complete [Teller, 1994c]. While in theory, any computable function can be evolved using such a language, the lack of structure in the memory model makes it seem inherently difficult to produce solutions to complex problems in practice. We are motivated by the observation that even if considerable intelligence (in the form of people, either singularly or in teams) is available to guide the search for a program which solves a problem, then the search is easier and more likely to be successful if the program's use of memory is structured or controlled, rather than if random or unconstrained access to all memory is allowed. From this starting point we investigate if the same is true for genetic programming, where the unintelligent search is guided only by the fitness function. The thrust of the rest of the book is to show that this is indeed true.

3.2 TOURNAMENT SELECTION

In evolutionary algorithms there are many different techniques in use for deciding which individuals will reproduce, how many children they will have and which individuals will die (i.e. be removed from the population). The general characteristic is to reward better solutions with more offspring (and possibly also with longer life). However the question of how much to reward good individuals is important. If a single very good individual has many children then the genetic diversity of the population may fall too much. But if every individual has about the same number of children then there is little selection pressure on the population to evolve in the desired direction.

Various fitness re-scaling schemes have been used to rescale fitness values, so that the effective fitness of potential parents and so the number of children they are expected to have is within some prescribed "reasonable" range. For example, the rescaled fitness of the best member of the population might be twice that of the worst. Other schemes order potential parents by their fitness and use their position or "rank" within the population to determine how many children they will have. This can produce a prescribed reproduction pattern across the population, which is largely independent of the numerical fitness values returned by the fitness function (all that is important is whether fitness scores are bigger or smaller than others, not by how much). Arguably independence from numerical values makes the fitness function easier to produce.

The above schemes require information from the whole population. With small, centralised (i.e. not distributed), generational populations, this is not too bad a problem. However with large or distributed or dynamic (i.e. steady state, see next section) populations, maintaining global fitness data for selection becomes more onerous. Tournament selection has become increasingly popular as it performs (albeit noisy) rank selection based selection using only local information. As it does not use the whole population, tournament selection does not require global population statistics.

In tournament selection, a number of individuals (the tournament size) are chosen at random (with reselection) from the breeding population. These are compared with each other and the best of them is chosen. As the number of candidates in the tournament is small, the comparisons are not expensive. An element of noise is inherent in tournament selection due to the random selection of candidates. Many other selection schemes are also stochastic (i.e. contain an element of chance), in which case the level of "noise" they have on the selection process may be considered important. [Goldberg and Deb, 1991] and [Blickle and Thiele, 1995] compare features (including selection noise) of various commonly used selection schemes.

3.3 STEADY STATE POPULATIONS

In a traditional GA [Holland, 1992], evolution proceeds via a sequence of discrete generations. These do not overlap. An individual exists only in one generation, it can only influence later generations through its children. This is like many species of plants (and animals) which live only one year. In the spring they germinate from seeds, grow during the summer and produce their own seeds in the autumn. These survive the winter by lying dormant, but their parents die. These new individuals start growing again in the next spring. Thus the species as a whole continues through many years but no one individual lives longer than a year.

In contrast many plants and animals live many years and there is no distinct boundary between generations. In steady state GAs [Syswerda, 1989; Syswerda, 1991b] new children are continually added to the population and can immediately be selected as parents for new individuals. Usually as each new individual is added to the population an existing member of the population is removed from it. This ensures the population remains at a constant size.

To ease comparisons between steady state and generational GAs, the term *generation equivalent* is used. It means the time taken to create as many new individuals as there are in the population. Thus a generation equivalent represents the same computational effort (in terms of number of fitness evaluations) as a single generation in a traditional GA with the same sized population.

Steady state populations are increasingly popular. All the genetic programming experiments in this book use steady state populations.

3.4 INDEXED MEMORY

The indexed memory model used is based upon [Teller, 1994a]. The indexed memory consists of $2l + 1$ memory cells (numbered $-l \ldots +l$), each of which holds a single value. Attempts to access memory outside the legal range either cause the program to be aborted or the data being written to be discarded and a default value of zero returned. Details are given with each of the experiments. In contrast, Teller avoids the address range problem by reducing the address index modulo the size of the memory (which is addressed $0 \ldots m-1$). In [Teller, 1994a] 20 memory cells addressed $0 \ldots 19$ are used.

Note, like other functions, write returns a value. We follow Teller's example and define it to return the *original* value held in the store it has just overwritten. Many of the evolved programs exploit this behaviour.

Some experiments also use a swap function. This takes two arguments which it treats as addresses within index memory of two data values. It swaps them, so they now occupy the other address in indexed memory. Table 6.4 (page 128) defines swap in detail.

3.5 SCALAR MEMORY

In addition to indexed memory the experiments make use of one or more scalar memory cells known as auxiliary variables. Depending upon the experiment there are primitives to set, read, increment and decrement them.

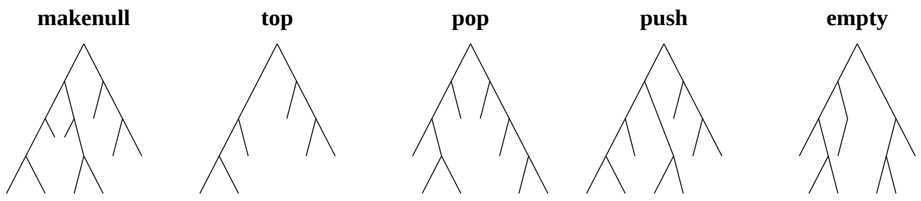


Figure 3.1. One Individual – One Program: Five Operations – Five Trees

3.6 MULTI-TREE PROGRAMS

In Chapters 4 to 6 and Section 7.3 the evolved program must perform more than one action. This is represented by allocating an evolvable tree per action. When the program is used, e.g. during its fitness testing, then the tree corresponding to the desired action is called. For example, when evolving a stack in Chapter 4 there are five different operations that a stack data structure must support. Each of these is allocated its own evolvable tree. So each individual within the population is composed of five trees, see Figure 3.1.

This multiple tree architecture was chosen so that each tree contains code which has evolved for a single purpose. It was felt that this would ease the formation of "building blocks" of useful functionality and enable crossover, or other genetic operations, to assemble working implementations of the operations from them. Similarly complete programs could be formed whilst each of its trees improved.

The genetic operations, reproduction, crossover and mutation are redefined to cope with this multi-tree architecture. While there are many different ways of doing this [Raik and Durnota, 1994], we define the genetic operations to act upon only one tree at a time. The other trees are unchanged and are copied directly from the first parent to the offspring. Genetic operations are limited to a single tree at a time in the expectation that this will reduce the extent to which they disrupts "building blocks" of useful code. Crossing like trees with like trees is similar to the crossover operator with "branch typing" used by Koza in most of his experiments involving ADFs in [Koza, 1994].

In the case of reproduction, the only action on the chosen tree is also to copy it, in other words each new individual is created by copying all trees of the parent program.

When crossing over, one type of tree is selected (at random, with equal probability, e.g. 1/5). This tree in the offspring is created by crossover between the tree in each parent of the chosen type in the normal GP way [Koza, 1992]. The new tree has the same root as the first parent (see Figure 3.2). Each mating produces a single offspring, most of whose genetic material comes from only one of its parents.

In the first set of experiments in this book, all trees have identical primitives. In later experiments, each tree has its own set of primitives from which it may be composed, see Section 5.10.

Should the offspring program exceed the maximum allowed length, the roles of the two parents are swapped, keeping the same crossover points. Given that the parents are of legal length, this ensures the offspring will be legal.

This use of multiple trees and entry points appears to have been "invented" several times. The first use of multiple trees is probably [Koza, 1992, Sections 19.7 and 19.8]

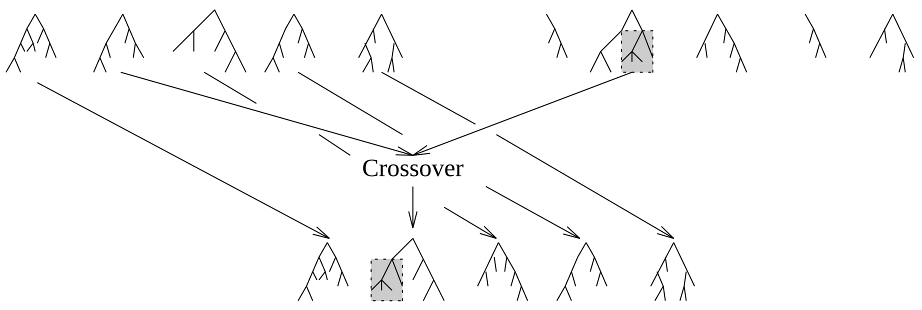


Figure 3.2. Crossover in One Tree at a time

where there are two distinct branches in the same program with different terminal sets. Unlike the trees in Figure 3.2, the branches are not equal. One branch is subservient to the other, in that it is always called before the start of the main branch, and so there is only one entry point. However the two branches used in [Andre, 1994b] are more equal, with each having its own ADFs, while [Taylor, 1995] used three separate trees and also allowed each an ADF. Multi agent programs have also been evolved using this approach, with a tree per agent [Haynes et al., 1995a]. While [Reynolds, 1994a] has a single agent, the agent has two very different behaviours. In some experiments these are forced into separate code branches. Due to the high variation between runs and the use of mutation, it is unclear if syntactically separating the behaviours is beneficial on its own. Bruce's one tree per data method has been described above in Section 7.4.8.

The CoacH system [Raik and Durnota, 1994] allows the user to specify multiple trees within a single individual in the population. (Each individual represents a team and each tree corresponding to a team member within the team). This approach is slightly different in that it allows the user to specify how many trees participate in crossover and whether crossover must be between trees (team members) of the same type. While [Luke et al., 1997] evolves a team of 11 co-operating agents. The task is subdivided by spliting the eleven agents into four squads. Agents in the same squad share the same code. Each sqaud is composed of a main tree and (up to) two ADFs. Making a total of 12 trees, which are co-evolved together.

3.7 DIRECTED CROSSOVER

This section surveys approaches in which the standard random genetic operators have been modified to direct or bias the location within parent programs on which they act. However before we consider exotic techniques we shall explain the standard one.

[Koza, 1992, page 114] and most others (including this book) use a crude aspect of program syntax (based on differentiating between functions and terminals, i.e. internal nodes and leaf nodes) to stochastically guide the location of crossover points. Crossover is biased to increase the proportion of times it moves subtrees headed by functions, as these are larger than those headed by terminals (which contain a single leaf node). In [Koza, 1992] on average 90% of crossovers exchange functions. In this work the proportions are governed by the GP-QUICK parameter pUnRestrictWt. It determines the proportion of crossover points that can be either terminals or functions compared to those that must be functions. In large binary trees (where the number of terminals is approximately equal to the number of functions), the default value of pUnRestrictWt (70%) corresponds to $(100\% - 70\%) + 70\%/2 = 65\%$ of crossovers inserting trees larger than a single terminal. (Table 8.5 (page 197) shows the actual value can be quite close to 65% in practice).

[Angeline, 1996a, page 27] argues "that no one constant value for leaf frequency is optimal for every problem". While this seems likely to be true, we need to consider the part mutation and other non-standard techniques play in his experiments. Also determining optimal values for any problem is expensive, therefore we have retained the GP-QUICK default.

The remainder of this section describes more sophisticated techniques for guiding GP evolution. While such approaches could be used with mutation, work has concentrated upon the choice of crossover points. We start with the work in this book and then briefly consider work by others. Most experiments in this book use the standard choice of crossover points described above. However Chapters 5 and 6 contain techniques to probabilistically bias the choice of crossover points. Two methods are used; firstly ensuring offspring obey various semantic (described in Section 5.10.3) or syntactic (Section 6.4.2) restrictions. If these conditions are not met, the offspring is aborted and a replacement is generated by performing another crossover. The second approach (Section 6.6) actively drives the choice of crossover points using performance data gathered during fitness testing of the parents.

A number of papers show benefits in directing or biasing the operation of the crossover or other genetic operators. For example [Whigham, 1995a; Whigham, 1995b] uses a grammar to constrain the evolving trees but the grammar itself evolves based on the syntax of previously successful programs (in fact the best of each generation). The grammar does not become more constrictive but instead the rules within it are allocated a fitness which biases (rather than controls) the subsequent evolution of the population. [Whigham, 1996, page 231] says "recently there has been increasing interest in using formal grammars to represent bias in an evolutionary framework" and gives an overview of grammatically biased learning. LOGENPRO [Wong and Leung, 1995] and Generic Genetic Programming (GGP) [Wong and Leung, 1996] are also based upon formal grammars, while [Gruau, 1996b] argues strongly that GP workers should be forthright in using program syntax to guide the GP and shows improved GP performance by using an external grammar to define more tightly the syntax of the evolving programs.

[D'haeseleer, 1994] describes methods, based upon the syntax of the two parent programs, for biasing the choice of crossover locations so that code at similar physical locations within programs is more likely to be exchanged. The motivation is such code may be more likely to be similar than random code and so changes introduced may be smaller and so more likely to be beneficial. The assumption is that large changes are more random and so, in a complex problem, more likely to be harmful.

An approach to protect code from crossover is the use of genetic libraries ([Angeline, 1993] and [Rosca and Ballard, 1996], described in Section 2.4.5). The ETL group [Iba and de Garis, 1996] is also active in this area, work on their COAST system is reported in [Hondo et al., 1996b] and summarised in [Hondo et al., 1996a]. Also "introns" are suggested to protect code from crossover [Nordin et al., 1996] but [Andre and Teller, 1996, page 20] concludes “that introns are probably damaging”, while the EPI system [Wineberg and Oppacher, 1996] relies upon them. ([Blickle, 1996] reports explicit introns may sometimes caused performance degradation on a boolean problem). [Angeline, 1996b] advocates evolving the probability of crossover occurring at different points in the program along with the program itself. He also suggests multiple crossovers to produce an offspring. [Teller, 1995b] includes a library of callable code plus the co-evolution of “smart” crossover operators. The evolving “smart” crossover operators are free to select crossover points as they choose whilst they create offspring for parents in the main population.

[Blickle and Thiele, 1994, Section 4] claims improved performance by marking tree edges when they are evaluated and ensuring crossover avoids unevaluated trees, however the improvement is problem dependent. In [Blickle, 1996] a deleting crossover operator which removes unevaluated trees is shown to give more parsimonious solutions on a discrete problem.

The “soft brood” approach in [Tackett, 1995a] is different, in that the genetic operator itself is not biased, instead improved offspring are produced by producing multiple offspring per parent pairing and using a (possibly simple) fitness function to ensure only the best are released into the population and so able to breed. [Crepeau, 1995, Section 2.2.1] uses a similar technique. It could also be argued that fitness functions which reward parsimony (i.e. short code) are also biasing the genetic search process. The Minimum Description Length (MDL) and Occam’s razor approaches have been described in Section 2.4.2.

There has been increasing interest in the use of “type” information to guide the creation of the initial population and its subsequent evolution via crossover since Strongly Typed Genetic Programming (STGP) was introduced by [Montana, 1993; Montana, 1994] (see Section 2.4.1). While [Montana, 1995] argues the reduction in search space is important, a more convincing explanation for the power of STGP is the use of type information to pick a better route through the search space by keeping to the narrow path of type correct programs.

3.8 DEMES

Various means to divide GA populations into subpopulations have been reported in conventional GAs [Stender, 1993; Collins, 1992] and genetic programming [Tackett, 1994; Ryan, 1994; D’haeseleer and Bluming, 1994; Koza and Andre, 1995b; Juille and Pollack, 1995]. Dividing the population limits the speed at which it converges and so may reduce the impact of premature convergence (i.e. when the population converges to a local optimum rather than the global optimum) and improve the quality of the solutions produced. (If the population is split, with very little genetic communication between its components, the population need never converge).

Demes are used in various experiments (notably in Chapters 5, 6 and 7). In this work, where direct comparisons were made, the use of a structured population, i.e. of demes, always proved to be beneficial in comparisons with simple non-demic, i.e. panmictic population. However in some cases better results were obtained by using fitness niches (see Section 3.6). Where demes are used, the model described in this section is used. In this model (which is based upon [Collins, 1992]) the whole population is treated as a rectangular grid of squares with two individuals in each square. Crossover can occur only between near neighbours, i.e. within (overlapping) demes. To avoid edge effects the grid is bent into a torus, so that each edge of the rectangle touches the opposite one.

In addition to crossover, reproduction is used. As usual two tournaments are conducted, the first chooses which individual (within the deme) to replace, and the second chooses which to copy.

Before a selection tournament occurs, the candidates for selection must be chosen. Without demes individuals are selected at random from the entire population. This leads to the population being well mixed, which is known as a panmictic population. When demes are used, all members of the selection tournament come from the same deme, i.e. a small part of the population. Figure 3.3 shows the sequence of selection events. This technique differs in detail from [Collins, 1992, Section 2.4.1] in that there are two individuals per grid square (rather than one), reverse tournament selection (rather than random) is used to select the individual to be replaced and tournament candidates are chosen with uniform probability from a square neighbourhood. [Collins, 1992] uses a random walk process to approximate a gaussian distribution centered about the individual to be replaced.

Demes have some similarities with cellular automata, in that (apart from its contents) each deme is the same as every other deme. Also the new population in a deme is related to its current population and the populations of its neighbours. This is similar to the way the next state of a cell within a cellular automata is determined by its current state and the states of its neighbours. However there are important differences: the contents of each deme is one or more programs, as the number of potential programs is huge (e.g. more than $1.4\ 10^{262}$ for the stack problem in Chapter 4), the number of states a deme may be in is also enormous. In a cellular automata usually the number of states is small. New programs are created stochastically, so given the populations in a deme and its neighbours, the new population in the deme can be one of a huge number of different possibilities. Each possible new population has in general a different probability, being given by the fitnesses of the individuals in the deme and surrounding demes. Classically cellular automata operate in parallel, while demes are updated sequentially and stochastically.

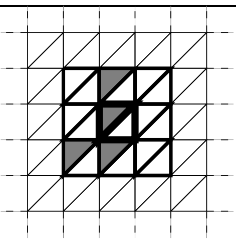
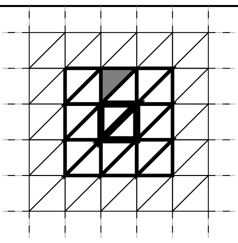
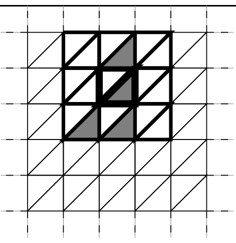
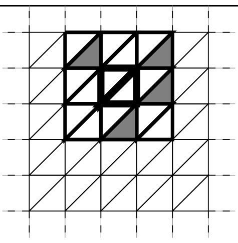
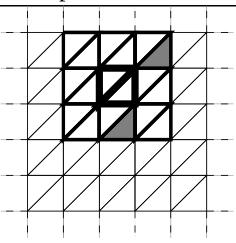
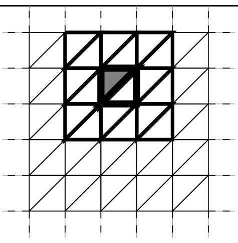
	
1 3 $\times$ 3 deme chosen at random, choose another 3 candidates for replacement at random from deme. (The centre of the deme is shown with bold edges and the four chosen individuals are shown shaded).	**2** Select worst candidate for replacement. (The upper triangles represent individuals which are being considered for deletion).
	
3 Choose 4 candidates for 1st parent from new deme centered on the individual to be replaced.	**4** Choose 2nd tournament group of 4 candidates for 2nd parent from same deme.
	
5 Create new program by crossing over tournament winners.	**6** Replace loser from first tournament with new program.

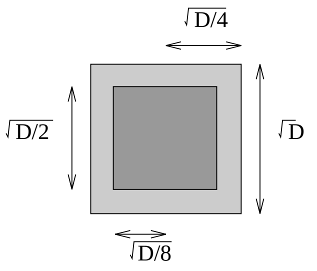
Figure 3.3. Selecting Parents and Individual to be Replaced in a Demic Population

Limiting Convergence

Using this deme structure a fit individual's influence within the population is limited by how fast it can move through the population. The following analysis shows this depends upon how much better it is than its neighbours.

If a program is consistently better than its neighbours (and so too are its offspring), then its influence (i.e. its offspring) is expected to spread at a high rate across the population (NB this means its number grows quadratically, rather than exponentially). Each time it produces a new offspring, the new offspring will be about $1/2\sqrt{D/2}$ from its parent (where there are D grid points in the deme, see Figure 3.4). When considering how the individuals spread we need only consider those at the edge. When these reproduce only about 50% of their offspring will be outside the previously occupied area.

In a rectangular torrodial population of size M with each grid point containing P individuals, each deme is within $1/2\sqrt{\frac{MR}{P}}$ of every other (see Figure 3.5). (R denotes the ratio of the rectangle's sides). The influence of a program that is consistently of above average fitness can be expected to spread throughout the whole population in about $2\frac{1/2\sqrt{\frac{MR}{P}}}{1/2\sqrt{D/2}} = 2\sqrt{2MR/DP}$ time steps.



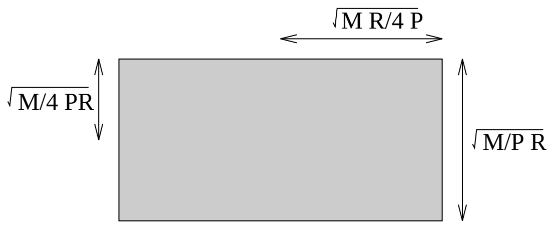


Figure 3.4. In a square deme containing D grid points, 50% points lie within $\frac{1}{2}\sqrt{\frac{D}{2}}$ of the center

Figure 3.5. In a rectangular population of M individuals, aspect ratio R and P individuals per grid point, each individual is within $\frac{1}{2}\sqrt{\frac{MR}{P}}$ of any other

The time taken to dominate the whole population is proportional to the program's reproduction rate, which in turn depends upon how much fitter it is than its neighbours. With tournament selection the fittest individual in a deme will win every tournament it is selected to be a candidate in. In demes on the edge of the program's influence, i.e. demes that don't yet contain many descendants of the individual, their chance of winning a selection tournament (of size t) is approximately t bigger than that of the average individual (see Section 8.4.2 page 188). With a crossover rate of p_c there are on average $1 + p_c$ tournaments per offspring created. Thus the maximum reproduction rate of an individual is about $t(1+p_c)/M$. However in GP, crossover is asymmetric, with one parent usually contributing more genetic material than the other (see Figure 3.2). If we consider only those parents, the maximum reproduction rate is t/M. Thus the shortest time for a very fit individual to dominate the whole population is $\approx \frac{2}{t}\sqrt{2MR/DP}$ generation equivalents. (If $M = 10,000$, $D = 9$, $P = R = 2$, $t = 4$, this is approximately 24 generation equivalents).

If a program is only slightly better than its neighbours, it can be expected to have about one offspring per generation equivalent. This will be placed within the same deme as its parent, but in a random direction from it. Thus the original program's influence will diffuse through the population using a "random walk", with a step size of about $1/2\sqrt{D/2}$ (see Figure 3.4). The absolute distance travelled by a random walk is expected to be step size $\times$ $\sqrt{\text{no. steps}}$. Thus the number of time steps required is (no. of steps required) squared. The number of generation equivalents it can be expected to take to spread through the whole population is $2\frac{MR}{DP}$ (If $M = 10,000$, $D = 9$, $P = R = 2$, then this is approximately 2200 generation equivalents).

Where selection is from the whole population, the chance of a program being selected to crossover with itself, is very small. However when each random selection is equally likely to be any member of a small (3 × 3) deme, the chance of any program being selected more than once is quite high. Possibly the increased chance of crossover between the same or similar programs may also be beneficial.

3.9 PARETO OPTIMALITY

Existing GPs (and indeed genetic algorithms in general and other search techniques) use a scalar fitness function where each individual is given a single measure of its usefulness. An alternative, explored later chapters of this book, is to use a multi-dimensional fitness measure where each fitness dimension refers to a different aspect of the trial solution.

In several experiments there is more than one task which the evolved program is to perform. For example when evolving a data structure there are multiple operations that the data structure must support. In the first experiments (Chapter 4) a single fitness measure is produced by combining the performance of the individual operations. Later work (particularly in Chapters 5, 6 and Section 7.3) separates the performance of each operation, each contributing a dimension to the overall multi-dimensional fitness measure. In some cases penalties for excessive CPU or memory usage also contribute a dimension to the fitness. (Since fitness testing often requires more than one operation to be active, e.g. when testing that operations work together, a total separation between fitness dimensions is not possible. Nevertheless a multi-objective fitness function does allow some measure of which parts of the program are working well to be recorded).

Pareto optimality [Goldberg, 1989, page 197] offers a way of comparing individuals within the population using multiple criteria without introducing an arbitrary means of combining them into a single fitness. Evidence for the effectiveness of Pareto Tournament selection is given in [Fonseca and Fleming, 1993] and [Louis and Rawlins, 1993]. [Fonseca and Fleming, 1995] contains a review of multi-objective optimisation techniques used inconjunction with various evolutionary computing algorithms. In a Pareto approach fitness values are compared dimension by dimension. If a fitness

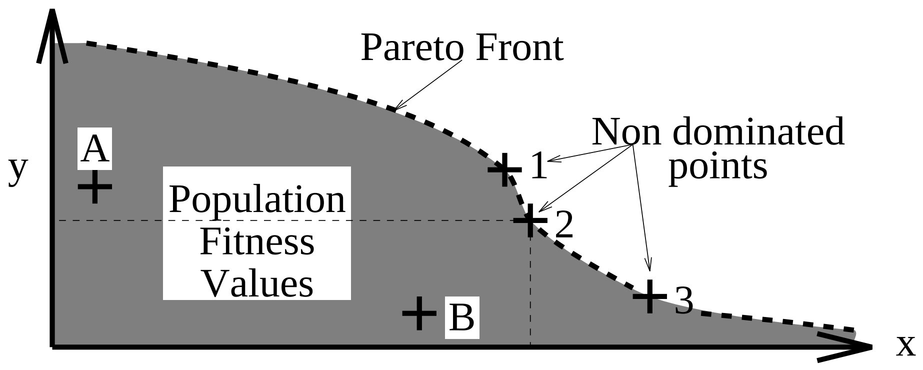


Figure 3.6. Two Dimensional Example of Pareto Optimality and the Pareto Front

value is no worse than the other in every dimension and better in at least one dimension then it is said to dominate the other. For example in Figure 3.6, point 2 dominates B but does not dominate A.

Pareto scoring means individuals which make an improvement on any part of the problem tend to be preferred, whereas a scalar fitness will tend to require each improvement to match or exceed any deterioration in all other parts of the problem. Whether an improvement is more important than a deterioration is given by scaling parameters within the fitness function. Consequently setting them is complex and must be done with care. To some extent Pareto fitness avoids this problem.

With “fitness sharing” [Goldberg, 1989], the fitness of individuals which are “close” to each other is reduced in proportion to the number of individuals. This creates a dispersive pressure in the population, which counteracts the GAs tendency to converge on the best fitness value in the population. So the number of occupied niches remains high.

An alternative approach is to impose a fixed number of niches. [Yang and Flockton, 1995] describes dynamic niches, containing clusters of individuals which move across the representation space as the population evolves. Hill climbing, via mutation, provides effective local search within each niche. However keeping track of such niches is computationally expensive [page 196].

In the case of a multi-objective fitness measure there is also a tendency for the number of niches to fall however fitness sharing can again be used to create a dispersive pressure and keep the number of occupied niches high [Horn et al., 1993]. [Horn et al., 1993] suggests a method of fitness sharing based upon estimating an individual’s Pareto rank by comparing it with a random sample of the whole population, known as the comparison set. (The implementation used in this work is described in the next subsection).

Figure 3.7 shows the evolution of the number of occupied points on the Pareto optimal surface in two runs starting from the same initial population. (The two runs were preliminary experiments on the list problem, cf. Chapter 6). We see the use of a comparison set leads to the retention of a large number of occupied niches, each of

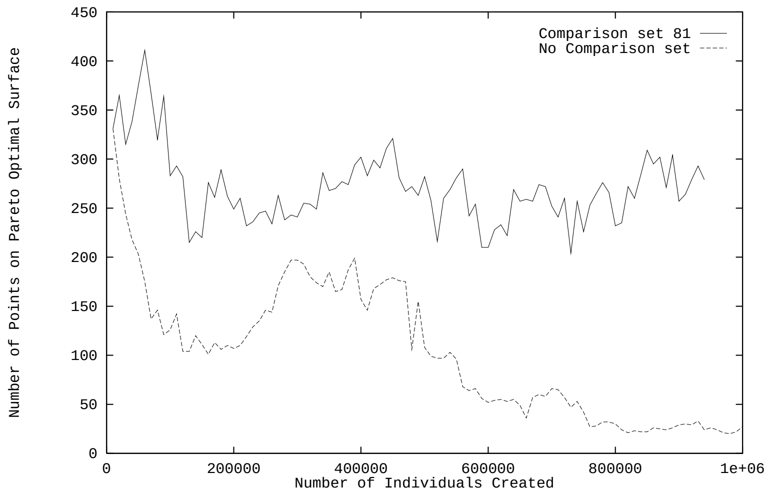


Figure 3.7. Number of different non dominated fitness values in a list population (Chapter 6) with and without a comparison set (no niche sharing, not elitist, pop=10,000, no demes)

which is the best on some combination of criteria. However without it, the number of niches falls.

[Oei et al., 1991] considered linear GAs with complete generational replacement and state the "naive" use of tournament selection to spread a GA population across equally fit niches will cause the population to behave chaotically. They predict [page 5] the number of niches, n, within a population of equally fit individuals will fall, being given by the formula:

$$n = \frac{1}{\frac{1}{n_0} + \frac{2G}{\pi^2 M}} \qquad (3.1)$$

where n_0 is the initial number of niches, G number of generations and M is the population size.

It can be seen that Equation 3.1 does not fit the lower curve in Figure 3.7 well. The derivation of Equation 3.1 made several assumptions that don't hold for Figure 3.7, perhaps the most important is that the fitness niches are static, whereas in Figure 3.7 each fitness value ceases to be a niche when a new solution which dominates is found. Nevertheless Figure 3.7 shows a general downward trend in the number of different non-dominated fitness values in the population (i.e. niches) after the initial high rate of fitness improvement slows and the population becomes more stable. The loss of variation with time in finite populations of equally fit individuals is also known as "genetic drift".

Both fitness sharing and demes encourage retention of genetic diversity. Demes promote breeding of nearby individuals (which are likely to be genetically similar) while fitness sharing retains a large fitness variation in the population. As the population has a large fitness diversity, it may also have a large genetic diversity, so fitness sharing promotes breeding between genetically diverse individuals.

Fitness Sharing Pareto Tournament Selection

Pareto optimality can be readily combined with tournament selection. In all the GP experiments described in this book a small number (4) of programs are compared and the best (or worst) is selected. With scalar fitness this is done by comparing each program in the tournament group with the current best. If it is better, then it becomes the new best. With Pareto selection, we need to consider the case where neither program is better than the other.

With Pareto ranking, instead of maintaining a unique "best so far" individual in the tournament, a list of "best so far" individuals is kept. Each member of the tournament group is compared with each member of the best so far list. If it is better than a "best so far" individual, that individual is removed from the list. If it is worse, then it is discarded. If after comparing with the whole list, it has not been discarded, it is added to the list. After comparing all candidates, the winner is taken from the "best so far" list. If there is more than one individual in the list and fitness sharing is not being used then the tournament winner is chosen at random from those in the list.

To reduce the size of the "best so far" list and so the number of comparisons, if a candidate has identical fitness to a member of the list, the candidate is discarded. This introduces a bias away from programs with identical scores, as it prevents them increasing their chances of selection by appearing multiple times in the "best so far" list.

Where a tournament group contains two, or more, non-dominated individuals (i.e. the "best so far" list contains more than one individual at the end of the tournament) the remainder of the population can be used to rank them. Thus an individual which is dominated by few others will be ranked higher than one dominated by many. NB this exerts a divergent selection pressure on the population as individuals are preferred if there are few others that dominate them. Following [Horn et al., 1993] the pareto rank is estimated by comparison with a sample of the population rather than all of it. Typically, in this work, a sample of up to 81 individuals is used.

Elitism. Using a conventional scalar fitness function and tournament selection a steady state population is *elitist*. That is the best individual in the population is very unlikely to be lost from the population. This is because it is very unlikely to be selected for deletion. This could only happen if the best individual was selected to be every member of a deletion tournament. The chance of this is M^{-k} (where M is the population size and k is the kill tournament size). If there is always a unique best individual then the chance of ever deleting it is $1-(1-M^{-k})^g$ where g is the number of individuals deleted. Assuming $M^k \gg g$ then we can approximate this with gM^{-k}. If G is the number of generation equivalents, then this becomes $GM^{-(k-1)}$. With $M = 10,000$, $k = 4$ and $G = 100$, the chance of deleting the unique best member of the population is $\leq 10^{-10}$. If there are multiple individuals with the highest fitness score then the chance of deleting any one of them is much higher, but then there will be at least one more individual in the population with the same high score.

Where the population is separated into demes the chance of deleting the unique best member of the population is much higher, however the best member will reproduce rapidly and so is unlikely to remain unique for long. The chance of selecting a deme containing the best individual is $1/M \times$ No. overlapping demes $= D/M$ (due to the implementation of overlapping demes, the number of individuals at each grid point need not be considered). The chance of selecting the best individual to be a candidate in a selection tournament is $1/D$ but for it to be deleted, it needs to be the only candidate, i.e. it needs to be selected k times. The chance of this is D^{-k}. Thus the chance the best individual will be deleted by any one kill tournament is $D/M \times D^{-k}$ $= D^{1-k}M^{-1}$. (If $M = 10,000$, $D = 9$, $k = 4$, this is approximately 0.14 10^{-6}). If there is always a unique best in the population (which need not always be the same individual) the chance of it ever being lost from the population is $1 - (1 - D^{1-k}M^{-1})^g$ $= 1 - \exp(g \log(1 - D^{1-k}M^{-1})) \approx 1 - \exp(-gD^{1-k}M^{-1}) = 1 - \exp(-G/D^{k-1})$. (If $G = 100$, $D = 9$, $k = 4$, this is approximately 0.13. I.e. in a steady state demic population in the worst case (where there is always only one copy of the best individual in the population) there is a small chance of deleting the best member of the population).

The combination of Pareto fitness and tournament selection is no longer elitist. This is because with the introduction of Pareto scoring there may be more than one, indeed many, individuals within the population which are the “best”, in the sense that there is none better. The population will tend to converge to these “best” individuals, i.e. their numbers will grow. It is possible for the whole of the tournament group to consist entirely of “best” individuals in which case one of them must be selected for deletion. With Pareto scoring, these need not have identical scores, only scores that were not dominated by the deleted program. In this way programs which appear to have the best score so far can be lost from the population. Figure 5.26 (page 116) shows several cases where the program with the highest total score (i.e. the total number of tests passed) is lost from the population, resulting in a fall in the simple sum of five of the six fitness measures.

Figures 3.8 and 3.9 show the advance of the “best in the population” Pareto front to higher fitness as the population evolved (albeit projected onto just two dimensions). The two populations come from runs of the queue problem (Section 5.10). Note these graphs only plot scores on two of the criteria (dequeue and front), the other criteria account for some concavities in the front.

3.10 CONCLUSIONS

This chapter has described briefly described early work on evolving programs’ use of memory and described in more detail the advanced GP techniques (tournament selection, overlapping generations, indexed and scalar memory, multi-tree programs, directed crossover, partitioning the population using demes and multi-objective fitness functions). These will be used in the experimental chapters, which follow immediately.

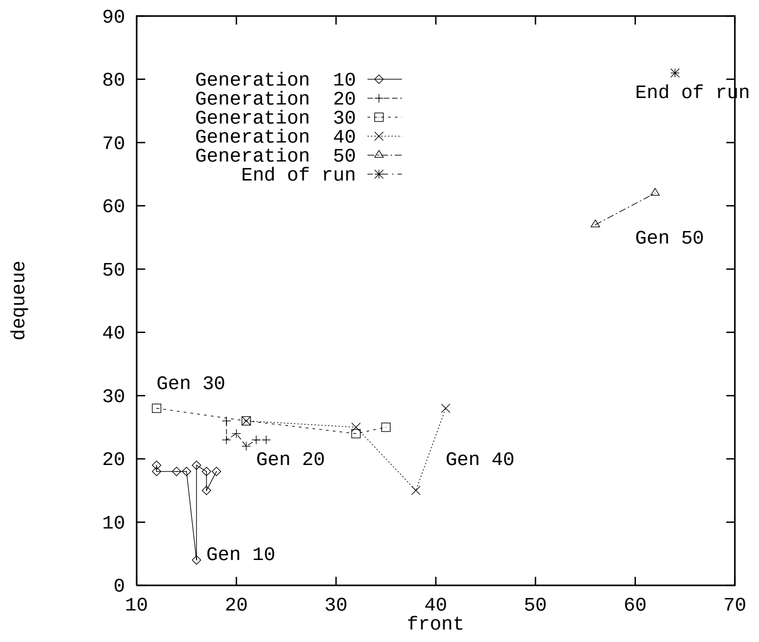


Figure 3.8. Evolution of the Pareto front (successful run of queue problem, cf. Section 5.10)

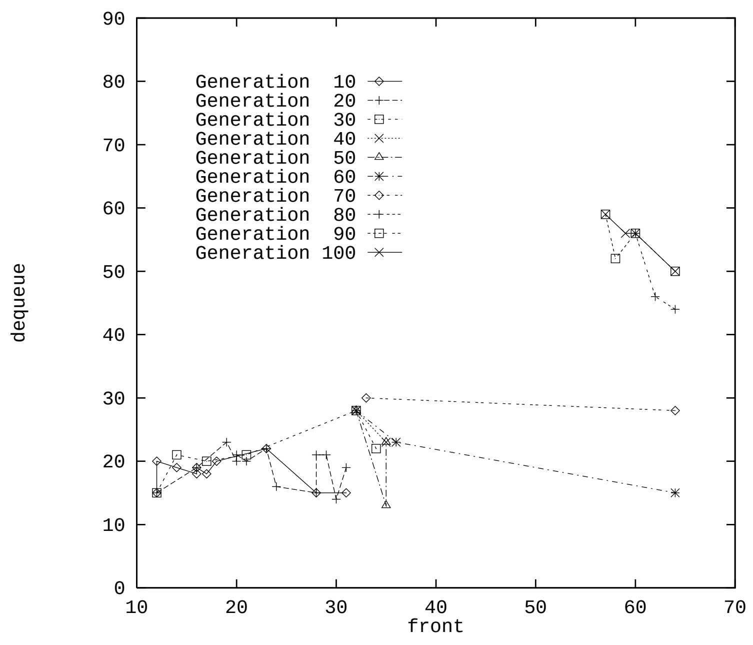


Figure 3.9. Evolution of the Pareto front (typical run of queue problem, cf. Section 5.10)