1. Introduction

Living creatures are divided into species. Over time (which may be comparatively short or very long) species change, i.e. they evolve. There are several essential components to natural evolution. Individuals (e.g. animals or plants) within a species population are different. As a result, some live longer and are more likely to have children that survive to adulthood than others (natural selection). These animals (or plants etc.) are said to be fitter (cf. "survival of the fittest"). Sometimes the variation has a genetic component, i.e. the children may *inherit* it from their parents. Consequently after a number of generations the proportion of individuals within the species with this favourable inheritable characteristic tends to increase. That is, over time the species as a whole changes or evolves.

Each animal or plant grows from an egg or seed to a full grown adult. Its development is controlled in part by its genes. These are the genetic blueprints which specify this development process and so the final form of the individual. Of course other factors (often simply called "the environment") or just plain luck have a large impact on the animal or plant. The relative importance of genes, which each of us inherits from our parents, and "the environment" remains deeply controversial.

Even today in many species each adult produces children whose genes come only from that adult. For example bulbs not only have seeds (sex) but also form daughter bulbs which grow directly from the parent's root. They are initially physically part of the same plant but may eventually split from it. Except for random copying errors (mutations) the new plant is genetically identical to the original.

Many of the species we are used to seeing use sex to produce children. Sex means the genes in each child are a combination of genes from its two (or more) parents. The process whereby the genes of both parents are selected, manipulated and joined to produce a new set of genes for the child is called *crossover*. (The term recombination is also occasionally used).

Why Nature invented sex is by no means clear [Ridley, 1993] but notice that it has several important properties. First children are genetically different from both their parents. Second, if individuals in the population contain different sets of genes associated with high-fitness, a child of two high fitness parents may inherit both sets of genes. Thus sex provides a mechanism

whereby new beneficial combinations of existing genes can occur in a single individual plant or animal. Since this individual may itself have children, the new combination of genes can in principle spread through the population.

The process of copying genes is noisy. While Nature has evolved mechanisms to limit copy errors they do occur. These are known as *mutations*. If errors occur when copying genes in cells which are used to create children, then the mutations will be passed to the children. It is believed that most mutations are harmful but sometimes changes occur that increase the fitness of the individual. By passing these to the children the improvement may over successive generations spread through the population as a whole.

Notice that although there are no explicitly given commands or even goals, natural evolution has over time produced a great many novel and sophisticated solutions to everyday (in fact life or death) problems. The idea that blind evolution can do this has proved very seductive.

Computer scientists and engineers have been exploiting the idea of natural evolution within their computers (artificial evolution) for many years. Initially the theoretical foundations of automatic problem solving using evolutionary computation were quite weak, but in recent years the theory has advanced considerably. This book concentrates on the theoretical foundations of one such technique, genetic programming (GP) [Koza, 1992, Banzhaf *et al.*, 1998a], which uses artificial evolution within the computer to automatically create programs. However genetic programming is a generalisation of older artificial evolutionary techniques called genetic algorithms (GAs) and so many of our theoretical results actually also cover results for GAs as special cases. Therefore these advances in GP theory can also be applied to GAs. This is part of a trend towards the advancement and coming together of the theory behind the various strands of evolutionary computing.

We will defer more details of genetic programming until Section 1.2, and Section 1.3 will give an outline of the rest of this book, but first we will consider evolution (either in Nature or in the computer) as a search process.

1.1 Problem Solving as Search

Genetic programming has been applied successfully to a large number of difficult problems such as automatic design [Koza *et al.*, 1999], pattern recognition [Tackett, 1993], data mining [Wong and Leung, 2000], robotic control [Banzhaf *et al.*, 1997], synthesis of artificial neural architectures (neural networks) [Gruau, 1994a, Gruau, 1994b], bioinformatics [Handley, 1995], generating models to fit data [Whigham and Crapper, 1999], music [Spector and Alpern, 1994] and picture generation [Gritz and Hahn, 1997]. (See also Appendix A).

Throughout this book we will treat evolution as a search process [Poli and Logan, 1996]. That is, we will consider all possible animals, plants, computer programs, etc. as our search space, and view evolution as trying a few of these

possibilities, deciding how "fit" they are and then using this information to decide which others to try next. Evolution keeps on doing this until it finds an individual which solves the problem. I.e. has a high enough "fitness". In practice search spaces are huge or even infinite and evolution (even Natural evolution with a whole planet's resources) cannot try all possibilities.

Any automatic program generation technique can be viewed as searching for any program amongst all possible programs which solve our problem. Naturally there are an enormous number of possible programs. (Given certain restrictions we can calculate how many there are; see page 114). As well as being huge, program search spaces are generally not particularly benign. They are usually assumed to be neither continuous nor differentiable, and so classical numerical optimisation techniques are incapable of solving our problems. Instead heuristic search techniques, principally stochastic search techniques, like GP, have been used. In this and the following chapters we shall cast a little light on the search space that GP inhabits and how GP moves in it.

Often genetic algorithm and GP search is like that shown in Figure 1.1. The dark-gray area represents the search space. Each dot (circular, square or triangular) is an individual. Different symbols are used to represent individuals of different generations in a population made of five individuals: circles represent the initial population (Gen 0), squares represent generation 1 and triangles represent generation 2. The initial population is scattered randomly throughout the search space. Some of its members (such as the circle at the top left of the figure) are better than others and so they are selected to have children (produced using crossover) more often. Other individuals are selected less frequently or not at all (such as the circle at the bottom of the figure). The creation of the generation 1 program at the left of Figure 1.1 (grey square) is caused by the mixing of the genes (by crossover) of the two initial programs at the left of Figure 1.1. This crossover event is represented by the arrows leaving the two parent programs and pointing to their child program. Very often crossover will produce offspring that share some of the characteristics of the parents and can be somehow considered as being "halfway" between the parents in our idealised example. As a result of selecting fitter individuals and of crossover producing children between their parents, the new generation will be confined to a smaller region of the search space. For example, the individuals at generation 1 (squares) are all within the light-gray area, while the individuals at generation 2 (triangles) will be confined to the white area.

One important question that the theory of evolutionary algorithms wants to answer is: "How can we characterise, study and predict this search?" According to the different approaches used to try to answer this question we can identify a number of major strands in the theoretical foundations of evolutionary algorithms.

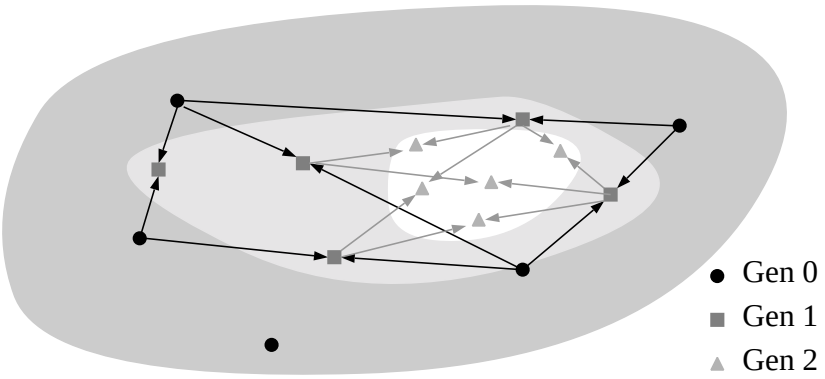

Fig. 1.1. Typical behaviour of evolutionary algorithms involving selection and crossover

1.1.1 Microscopic Dynamical System Models

These approaches are typified by Markov chain analysis which models the fine details of the individuals within the artificial population and how the genetic operations probabilistically change them over time [Nix and Vose, 1992, Davis and Principe, 1993, Rudolph, 1994, Rudolph, 1997c, Rudolph, 1997a, Rudolph, 1997b, Vose, 1999, Kellel *et al.*, 2001]. In these models the whole population is represented as a point in a multidimensional space, and we study the trajectory of this point to determine attractors, orbits, etc. The idea is illustrated in Figure 1.2. Typically these are exact models but with a huge number of parameters. Such models have been studied extensively in the genetic algorithm literature, but it is only very recently that Markov chain models for GP have been proposed [Poli *et al.*, 2001], and so, they are not discussed in this book.

1.1.2 Fitness Landscapes

In its simplest form a fitness landscape can be seen as a plot where each point in the horizontal direction represents all the genes in an individual (known as its genotype) corresponding to that point. The fitness of that individual is plotted as the height. If the genotypes can be visualised in two dimensions, the plot can be seen as a three-dimensional map, which may contain hills and valleys. Large regions of low fitness can be considered as swamps, while large regions of similar high fitness may be thought of as plateaus. Search techniques can be likened to explorers who seek the highest point. This summit corresponds to the highest-fitness genotype and so to the optimal solution (cf. Figure 1.3).

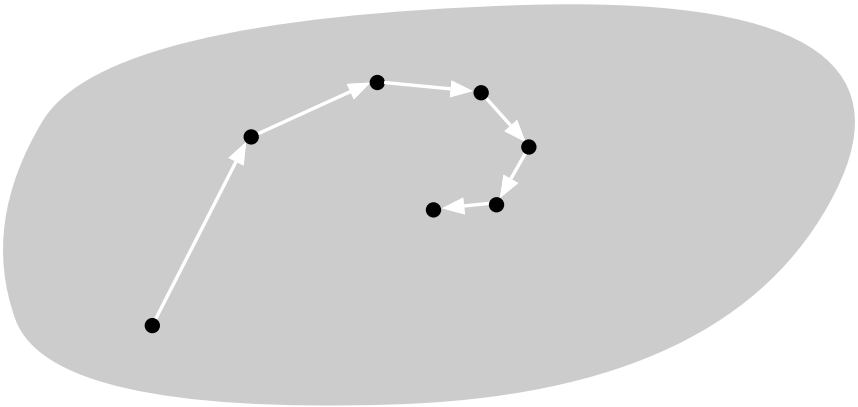

Fig. 1.2. Dynamical system model of genetic algorithms. The grey area represents the set of all possible states the population can be in. For a simple GA there are $\binom{2^N+M-1}{2^N-1}$ states (N, number of binary genes; M, size of population). The population starts in one state (represented by the lower left dot) and each generation moves (cf. white arrows) to the next.

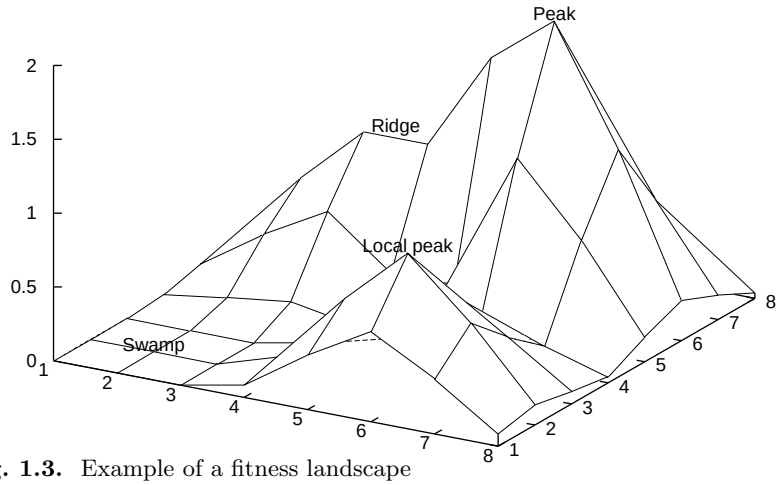

Fig. 1.3. Example of a fitness landscape

Greedy search techniques which only consider the local neighbourhood can be likened to short-sighted explorers, who just climb up the slope they are currently on. This is often a good strategy which leads to the top of the local hill, which is a local optimum. However the problem landscape may be structured so that this is not the global optimum. The local landscape may be flat (so there is no local gradient to guide the explorer) or it may lead the explorer via an unnecessarily long path to the local peak.

We can think of GAs as operating via a population of explorers scattered across such a landscape. Those that have found relatively high fitness points are rewarded by being allowed to have children. Traditionally mutation is seen as producing small changes to the genotype and so exploring a point near the parent, while crossover explores points between two parents and so may be far from either.

Notice that the fitness landscape incorporates the concept of neighbours, i.e. points that can be reached in one step from where we are now. Changing the neighbourhood changes the fitness landscape. This can have profound implications for how easy a problem is. Provided that all fitness values are different, it is theoretically possible to reorder any landscape so a local hill climber will quickly reach the global summit from any starting point. However in general it is not practical to find such a transformation due to the vast size of practical search spaces and consequently even larger number of different transformations.

Chapter 2 describes fitness landscapes in more detail. Chapter 9 maps the fitness landscape of the Santa Fe ant trail problem, while Section 8.3.1 shows the parity problem's fitness landscape is particularly awful.

1.1.3 Component Analysis

Component analysis concentrates on how genes or combinations of genes spread. Such combinations can be thought of as replicating themselves in their own right. These replicants are components of the higher level replicants represented by individual animals (or plants etc.) which reproduce and have children (who are also animals, plants etc.). [Dawkins, 1976] calls this idea the "selfish gene".

The component analysis approach focuses on the propagation of subcomponents of individuals. In genetic algorithms the individuals are represented in the computer by strings of bits. Component analysis considers what happens inside the strings and looks at single bits or groups of bits. For example, suppose the population contains just two bit strings: 0000 and 1111. Figure 1.4 shows the effects of crossing over the individuals 0000 and 1111 at their middle point to produce individuals 0011 and 1100. The component approach looks not at the effect on the individuals but at the effect on bit patterns. For example, before crossover the population contains three instances of the bit pattern 00, as shown in the upper part of Figure 1.4. (The bit string 0000 contains the bit pattern 00 in three different ways, i.e. using bits 1 and 2, 2 and 3 or 3 and 4). After crossover the population contains two instances of bit pattern 00. The deletion of one instance is represented by the cross in the lower part of Figure 1.4. The same crossover operation creates some other components. For example, the bit patterns 10 and 01. These are represented by the white dots in the lower part of Figure 1.4.

In genetic programming subcomponents could be single primitives, entire expressions, or even groups of expressions. For example, in GP we can con-

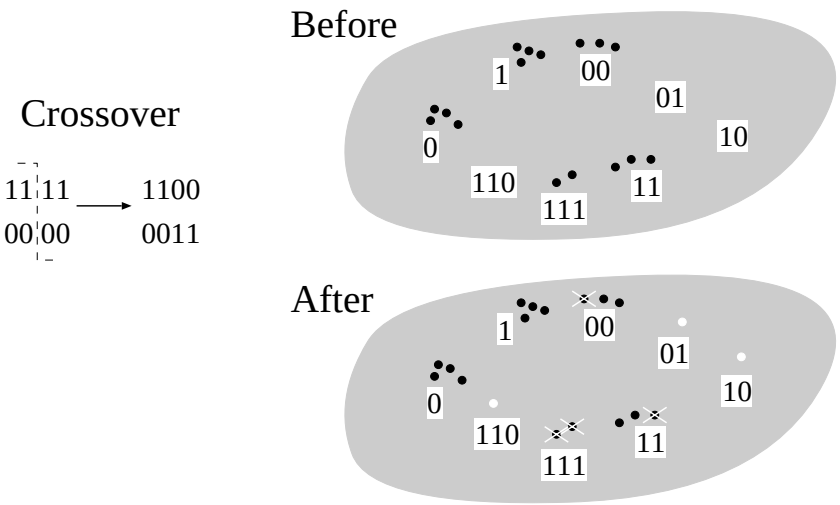

Fig. 1.4. Some of the effects of a crossover on the components of the bit strings 1111 and 0000 (left). The dots represent cases of the corresponding bit string components. After crossover (lower right) the white dots indicate instances of matching strings which did not exist before (i.e. schema creation). The crossed out dots indicate strings which no longer match (i.e. schema disruption) and black dots those that are unchanged.

sider how many copies of particular program components there are in total in the population and how this evolves. For example Chapter 10 considers how many "+" functions there are in the population within certain levels of the programs and analyses if they increase or decrease and even if they become extinct. [Langdon, 1998c, Chapter 8] gives a similar analysis for a different problem.

Chapter 3 describes various component analysis approaches in more detail.

1.1.4 Schema Theories

An alternative approach to understanding how genetic algorithms and genetic programming search, is based on the idea of dividing the search space into subspaces (called *schemata*[1]). The schemata group together all the points in the search space that have certain macroscopic quantities in common. For example all the programs which start with an IF statement could be grouped together in one schema.

Traditionally in genetic algorithms each gene is represented by a single bit. The complete set of genes is represented by a string of such bits. So,

[1] The word "schemata" is the plural for "schema".

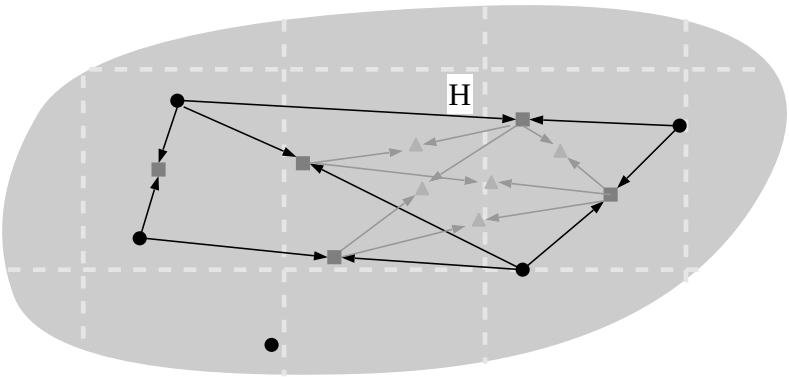

Fig. 1.5. The search space is divided into subspaces (schemata), shown as large squares. The populations are the same as in Figure 1.1 but instead of keeping track of every individual in the population we consider only how much of the population is in each schema.

as another example, we can group together every string with a particular pattern of genes in one schema. For example our schema could be everyone who has the allele 0 in locus 4 and the allele 1 in loci 5 and 6, i.e. all the strings with pattern 011 three bits from the left end. These form a subset of the whole search space (actually one-eighth of it).

Using such subsets it is possible to study how and why the individuals in the population move from one subspace to another (*schema theorems*). This idea is depicted in Figure 1.5. In the figure schemata are represented as squares. If we focus our attention on schema H, a reasonable way of considering the search might be to look at the number of individuals within H at generation t, $m(H,t)$. This number varies over time. In the initial generation $m(H,0)$ is 0. In the first generation $m(H,1)$ is 2, while $m(H,2) = 3$, etc. A *schema theorem* would model mathematically how and why $m(H,t)$ varies from one generation to the next. Chapters 3–6 will discuss the schema theories at length.

The concept of schema is very general and can be used to characterise the search space itself as well as the motion of the population in the search space. For example, as we will show in Chapter 7, one can divide the search space into sets (schemata) of programs with the same number of instructions, programs with the same behaviour or having the same fitness.

1.1.5 No Free Lunch Theorems

The no free lunch theorems (often abbreviated to NFL) are a series of results that prove that no search algorithm is superior to any other algorithm

on average across all possible problems. A consequence of this is that if an algorithm, say genetic programming, performs better than random search on a class of problems, that same algorithm will perform worse than random search on a different class of problems [Wolpert and Macready, 1997]. From this one might draw the erroneous conclusion that there is no point in trying to find "better" algorithms. However, since typically we are not interested in all possible problems, this is not the case. A search algorithm can do better than random search provided that the search bias explicitly or implicitly embedded in such an algorithm matches the regularities present in the class of problems of interest.

To date no specialised result for GP has been presented, and this topic is not discussed further in the book.

1.2 What is Genetic Programming?

Figure 1.6 shows the essential iterative nature of evolutionary algorithms. In evolution in the computer, like in Nature, there is a population of individuals. The fortunate or fitter ones survive and produce children. These join the population and older individuals die and are removed from it. As in Nature, there are two ways this can happen. The whole population may be replaced by its children (the generational model). For example, in annual plants or animal species where only the eggs survive the winter, the whole population is renewed once a year. Alternatively new children are produced more or less continuously (the steady state model). In the computer, the population is usually of fixed size and each new child replaces an existing member of the population. The individual that dies may be: chosen from the relatively unfit individuals (e.g. the worst) in the population, chosen at random or selected from the child's parents.

In artificial evolution most of the computer resources are devoted to deciding which individuals will have children. This usually requires assigning a fitness value to every new individual. In genetic programming the individuals in the population are computer programs. Their fitness is usually calculated by running them one or more times with a variety of inputs (known as the test set) and seeing how close the program's output(s) are to some desired output specified by the user. Other means of assigning fitness such as coevolution are also used (e.g. [Juille and Pollack, 1996]).

In genetic programming the individual programs are usually written as syntax trees (see Figures 1.6–1.11) or as corresponding expressions in prefix notation. New programs are produced either by mutation or crossover. There are now many mutation and crossover operators: [Langdon, 1998c, pages 34–36] describes many of the mutation and crossover operators used in GP. Mutation operators make a random change to (a copy of) the parent program. For example, point mutation replaces a node within the tree by another chosen at random which has the same arity (i.e. number of arguments).

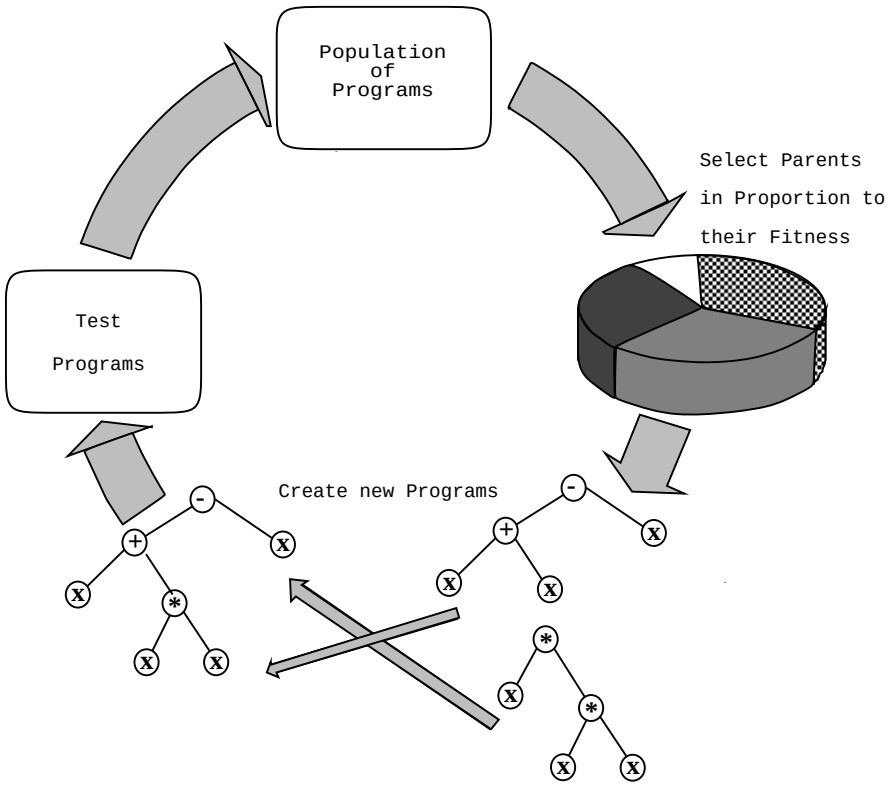

Fig. 1.6. Genetic programming cycle. The GP cycle is like every evolutionary process. New individuals (in GP's case, new programs) are created. They are tested. The fitter ones in the population succeed in creating children of their own. Unfit ones die and are removed from the population.

Thus a leaf (which occurs at the end of a branch, and has arity 0) is replaced by another leaf chosen at random and a binary function (e.g. $+$) is replaced by another binary function (e.g. $\times$).

Crossover works by removing code from (a copy of) one parent and inserting it into (a copy of) the other. In [Koza, 1992] subtree crossover removes one branch from one tree and inserts it into another. This simple process ensures that the new program is syntactically valid (see Figure 1.7).

1.2.1 Tree-based Genetic Programming

As a very simple example of genetic programming, suppose we wish a genetic program to calculate $y = x^2$. Our population of programs might contain a program that calculates $y = 2x - x$ (see Figure 1.8) and another that calculates $y = \frac{x}{\frac{x}{x-x^3}} - x$ (Figure 1.9). Both are selected from the population

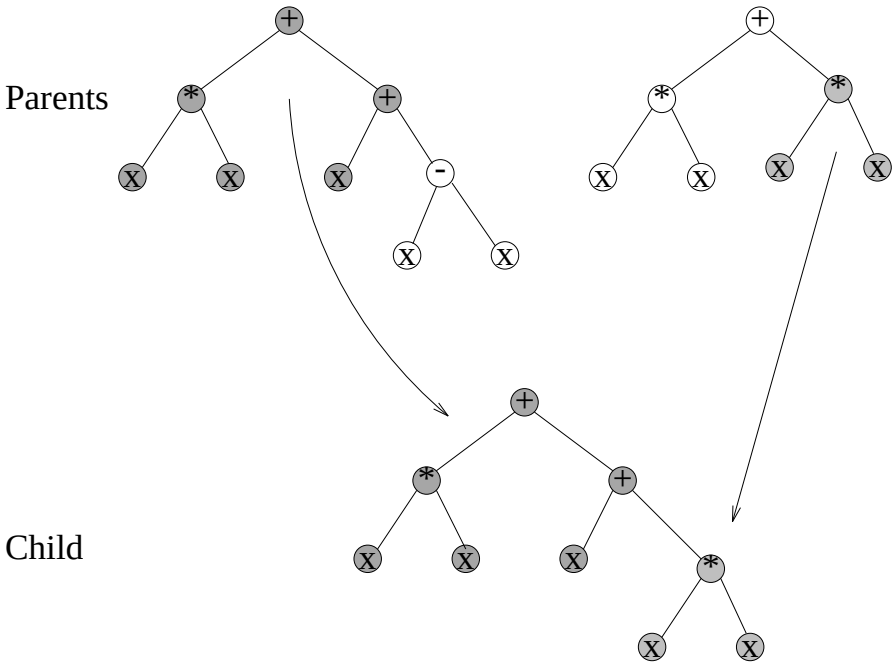

Fig. 1.7. Genetic programming subtree crossover: $x^2 + (x + (x - x))$ crossed with $2x^2$ to produce $2x^2 + x$. The right hand subtree $(\times\, x\, x)$ is copied from the right hand parent and inserted into a copy of the left hand parent, replacing the subtree $(-\, x\, x)$ to yield the child program $(+\, (\times\, x\, x)\, (+\, x\, (\times\, x\, x)))$.

because they produce answers similar to $y = x^2$ (Figure 1.11), i.e. they are of high fitness. When a selected branch (shown shaded) is moved from the father program and inserted into the mother (displacing the existing branch, also shown shaded) a new program is produced which may have even higher fitness. In this case the resulting program (Figure 1.10) actually calculates $y = x^2$ and so this program is the output of our GP. The C code for this example is available via anonymous ftp from `ftp://ftp.cs.bham.ac.uk/pub/authors/W.B.Langdon/gp-code/simple`

1.2.2 Modular and Multiple Tree Genetic Programming

Many variations on Koza's tree based genetic programming have been proposed. Of these perhaps Koza's own Automatically Defined Functions (ADFs) [Koza, 1994] are the most widely used. With ADFs the program is split into a main program tree and one or more separate trees which take arguments and can be called by the main program or each other. Evolution is free to decide how, if at all, the main program will use the ADFs as well as what they do. In more recent work even the number of ADFs and their parameters can be left to GP to decide [Koza *et al.*, 1999].

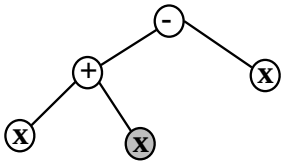

Fig. 1.8. Mum, fitness .64286, $2x - x$

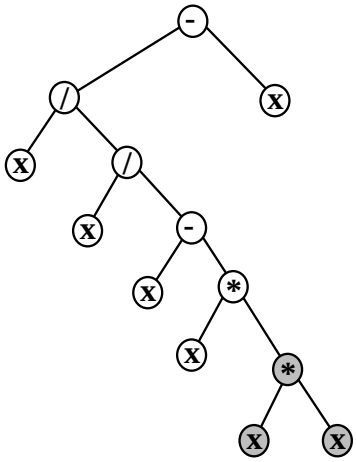

Fig. 1.9. Dad, fitness .70588, $\frac{x}{\frac{x}{x-x^3}} - x$

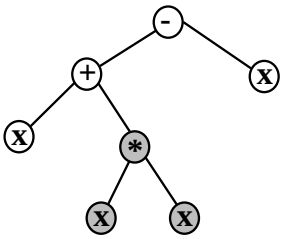

Fig. 1.10. Correct program, fitness 1.0, $x + x^2 - x$

Angeline's genetic library [Angeline, 1994] and Rosca's Adaptive Representation with Learning (ARL) [Rosca and Ballard, 1996] are other extensions to GP. In these, evolved code fragments are automatically extracted from successful program trees and are frozen and held in a library. They are then available to the evolving code via library calls to new primitives in the program tree. Sometimes code can be extracted from the library and returned to the program tree where it can continue to evolve.

Another approach, popular with multi-agent systems, is to have multiple trees, each of which has a defined purpose [Langdon, 1998c]. This can be combined with the ADF approach. So each individual may contain code for a number of agents or purposes. These may call other trees as functions (i.e. as ADFs). The ADFs may be specific to each purpose or shared by the agents or purposes.

Several authors have taken the tree program concept further. Broadly instead of the tree being the program it becomes a program to create another

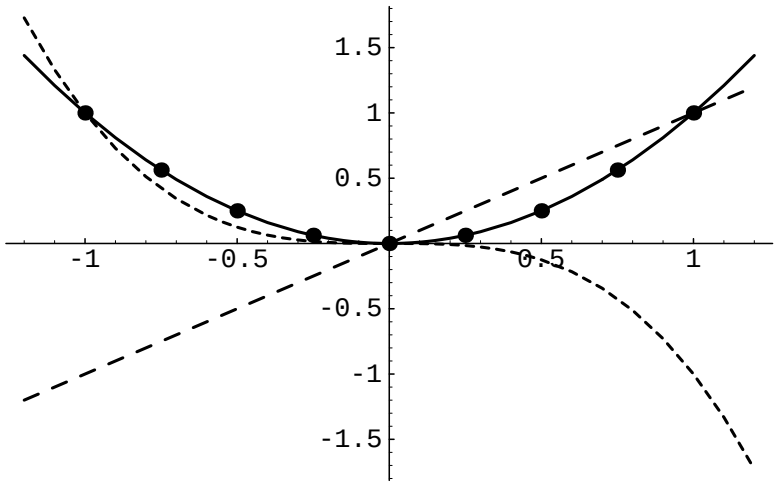

Fig. 1.11. x^2 (solid), test points (dots), values returned by mum ($2x - x$, dashed line) and dad ($\frac{x}{\frac{x}{x-x^3}} - x$, small-dashed line)

program. In [Gruau, 1994b]'s cellular encoding technique the evolved trees become programs to describe artificial neural networks or electrical networks [Koza and Bennett III, 1999] etc. Some draw inspiration from the development of embryos to further separate the phenotype from the genotype. For example [Jacob, 1996]'s GP uses Lindenmayer systems (L-Systems) as intermediates for evolving artificial flowers, while [Whigham and Crapper, 1999] treat the tree as defining a path through a formal grammar. The approach taken in [O'Neill and Ryan, 1999] is similar, but uses an initially linear program to navigate the grammar yielding a computer program written in the user-specified language.

1.2.3 Linear Genetic Programming

Except for Section 8.1 we shall not say much about linear GP systems. Briefly there are several linear GP systems. They all share the characteristic that the tree representation is replaced by a linear chromosome. This has some sim-

ilarities with conventional genetic algorithms. However, unlike conventional GAs, typically the chromosome length is also allowed to evolve and so the population will generally contain individual programs of different sizes. Typically each chromosome is a list of program instructions which are executed in sequence starting at one end. Linear GP systems can be divided into three groups: stack based, register based and machine code.

In stack-based GP [Perkis, 1994] each program instruction takes its arguments from a stack, performs its calculation and then pushes the result back onto the stack. For example when executing a + instruction, the top two items are both popped from the stack, added together and then the result is pushed back onto the stack. Typically the program's inputs are pushed onto the stack before it is executed and its results are popped from the stack afterwards. To guard against random programs misbehaving, checks may be made for both stack overflow and underflow. The programs may be allowed to continue execution if either happens, e.g. by supplying a default value on underflow and discarding data on overflow [Perkis, 1994].

Register-based and machine-code GP [Nordin, 1997] are essentially similar. In both cases data is available in a small number of registers. Each instruction reads its data from one or more registers and writes its output to a register. The program inputs are written to registers (which the evolved code may be prevented from over writing) before the program is executed and its answer is given by the final contents of one or more registers. In machine-code GP the instructions are real hardware machine instructions and the program is executed directly. In contrast register-based (and all other GP) programs are interpreted, i.e. executed indirectly or compiled before execution. Consequently machine code GP is typically at least ten or twenty times faster than traditional implementations.

1.2.4 Graphical Genetic Programming

While we do not deal explicitly with the theory of other types of GP, the reader may wish to note that programs can be represented in forms other than linear or tree-like. So the chromosomes of GP individuals can also be of these forms and genetic operators can be defined on them. For example PDGP (Parallel Distributed Genetic Programming) represents programs as graphs, see Figure 1.12. PDGP defines a number of graph mutation operators and crossover operations between graphs. The graph edges are directed and are interpreted as data-flow connections between processing nodes. While PDGP defines a fixed layout for the nodes, both the connections between them and the operations they perform are evolved. In a number of benchmark problems PDGP has been shown to produce better performance than conventional tree GP [Poli, 1997, Poli, 1999a].

Another example of graph programs is Teller's PADO (Parallel Architecture Discovery and Orchestration) [Teller and Veloso, 1996, Teller, 1998]. PADO is primarily designed for solving the problem of recognising objects

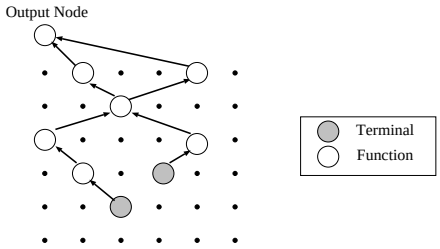

Fig. 1.12. Grid-based representation of graphs representing programs in PDGP (for simplicity inactive nodes and links are not shown).

in computer vision or signal analysis. Again there are a number of mutation operators and crossover operators and the graph edges are directed but they are control-flow connections between processing nodes. While PDGP defines a fixed layout for the nodes, PADO has a looser structure in which the number and location of processing nodes as well as the connections between them and the operations they perform are evolved.

1.3 Outline of the Book

Chapter 2 introduces the notion of fitness landscape and discusses its usefulness in explaining the dynamics of evolutionary search algorithms.

Chapter 3 describes ways of analysing evolution by looking at the propagation of components within the programs rather than the individual programs themselves. This can be likened to [Dawkins, 1976]'s "Selfish Gene" theory of evolution. Early genetic programming schema theorems, such as those due to Koza, Altenberg, O'Reilly and Whigham, fall into this category, which makes them different from the traditional genetic algorithm schema theory [Holland, 1975].

Schema theorems make predictions for the average behaviour of the population at the next generation. Chapter 4 looks at GP schema theorems (such as those due to Poli and Rosca) which provide lower bounds on schema proportions. Like Holland's schema theorem, these do not include the fact that genetic operators can create new instances of schemata as well as deleting (disrupting) them. Therefore we have referred to them as "pessimistic" schema theories. However Chapter 5 describes "exact" schema theories which do make allowance for schema creation. Chapter 6 concludes the presentation of the GP schema theorems by bringing together their implications.

Chapter 7 takes a different view point, by considering the GP search space and how it varies with the size and shape of programs. Some widely applicable results are presented. The formal proofs are deferred to Chapter 8.

We finish in Chapters 9, 10 and 11 with a detailed analysis of two benchmark GP problems and by considering bloat within GP. The analysis uses the

component analysis tools of Chapter 3, the schema theorems of Chapters 4–6 and the search space analysis of Chapter 7.

Appendix A gives some pointers to other sources of information on genetic programming, especially those available via the Internet, and is followed by the bibliography, a list of special symbols and a glossary of terms.

Finally, while each chapter finishes with a summary of the results contained within it, Chapter 12 gives our overall conclusions and indications of how the theoretical *foundations of genetic programming* may continue to develop.