1 INTRODUCTION

In both natural evolution and human endeavour, complex problems are solved by assembling solutions to parts of the problem into a complete solution. Whilst this is highly successful, it requires limited interaction between components. The building block hypothesis [Goldberg, 1989] states the same is true for artificial evolution. While doubts concerning the building block hypothesis have been expressed in general (e.g. [Beyer, 1995]) and for genetic programming (GP) in particular [O'Reilly and Oppacher, 1995], if complex solutions are to be evolved then it must be possible to assemble complete solutions from program fragments which solve parts of the problem. Where program components have complex interactions progress is more difficult, since improvement in one aspect will affect many others in an unpredictable and so usually negative way. Global memory allows such complex interactions. In software engineering complex interactions via global memory can be tackled by controlling programmers use of memory with scoping rules and abstract data types, such as stack, queues, files etc.

The thesis is that data structures can be used within the automatic production of computer programs via artificial evolution and that appropriate data structures are beneficial.

1.1 WHAT IS GENETIC PROGRAMMING?

Genetic programming [Koza, 1992] is a technique which enables computers to solve problems without being explicitly programmed. It works by using genetic algorithms to automatically generate computer programs.

Genetic algorithms (GAs) were devised by John Holland [Holland, 1992] as a way of harnessing the power of Darwinian natural evolution for use within computers. Natural evolution has seen the development of complex organisms (e.g. plants and animals) from simpler single celled life forms. Holland's GAs are simple models of the essentials of natural evolution and inheritance.

The growth of plants and animals from seeds or eggs is primarily controlled by the genes they inherited from their parents. The genes are stored on one or more strands of DNA. In asexual reproduction the DNA is a copy of the parent's DNA, possibly with some random changes, known as *mutations*. In sexual reproduction, DNA from both parents is inherited by the new individual. Often about half of each parent's DNA is copied to the child where it joins with DNA copied from the other parent. The child's DNA is usually different from that in either parent.

Natural evolution arises as only the fittest individuals survive to reproduce and so pass on their DNA to subsequent generations. That is DNA which produces fitter individuals is likely to increase in proportion in the population. As the DNA within the population changes, the species as a whole changes, i.e. it evolves as a result of selective survival of the individuals of which it is composed.

Genetic algorithms contain a "population" of trial solutions to a problem, typically each individual in the population is modelled by a string representing its DNA. This population is "evolved" by repeatedly selecting the "fitter" solutions and producing new solutions from them (cf. "survival of the fittest"). The new solutions replace existing solutions in the population. New individuals are created either asexually (i.e. copying the string, possibly with random mutations) or sexually (i.e. creating a new string from parts of two parent strings). The power of GAs (to find optimal or near optimal solutions) is being demonstrated for an increasing range of applications; financial, imaging, VLSI circuit layout, gas pipeline control and production scheduling [Davis, 1991].

In genetic programming (GP) the individuals in the population are computer programs. To ease the process of creating new programs from two parent programs, the programs are written as trees. New programs are produced by removing branches from one tree and inserting them into another. This simple process, known as *crossover*, ensures that the new program is also a tree and so is also syntactically valid (see Figure 1.1). Thus genetic programming is fundamentally different from simply shuffling lines of Fortran or machine code.

The sequence of operations in genetic programming is given in Figure 1.2. It is fundamentally the same as other genetic algorithms. While mutation can be used in GP, see Section 2.4.6, often it is not. For example it is only used in Appendix C in this book.

GP has demonstrated its potential by evolving programs in a wide range of applications including text classification or retrieval [Masand, 1994; Dunning and Davis, 1996], performing optical character recognition [Andre, 1994c], protein classification [Handley, 1993], image processing [Daida et al., 1996], target identification [Tackett, 1993], electronic circuit design [Koza et al., 1996a] and car monitoring for pollution control [Hampo et al., 1994]. At present published applications in everyday use remain rare, however Oakley's [Oakley, 1994] use of evolved medical signal filters and the BioX modelling system [Bettenhausen et al., 1995] are practical applications.

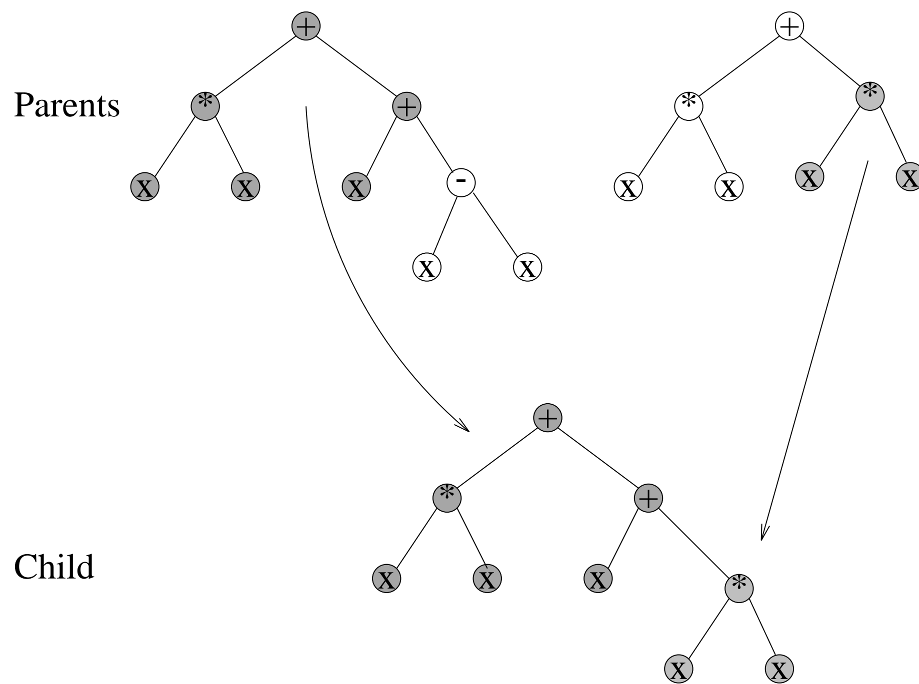

Figure 1.1. Genetic Programming Crossover:
$x^2 + (x + (x - x))$ crossed with $2x^2$ to produce $2x^2 + x$.

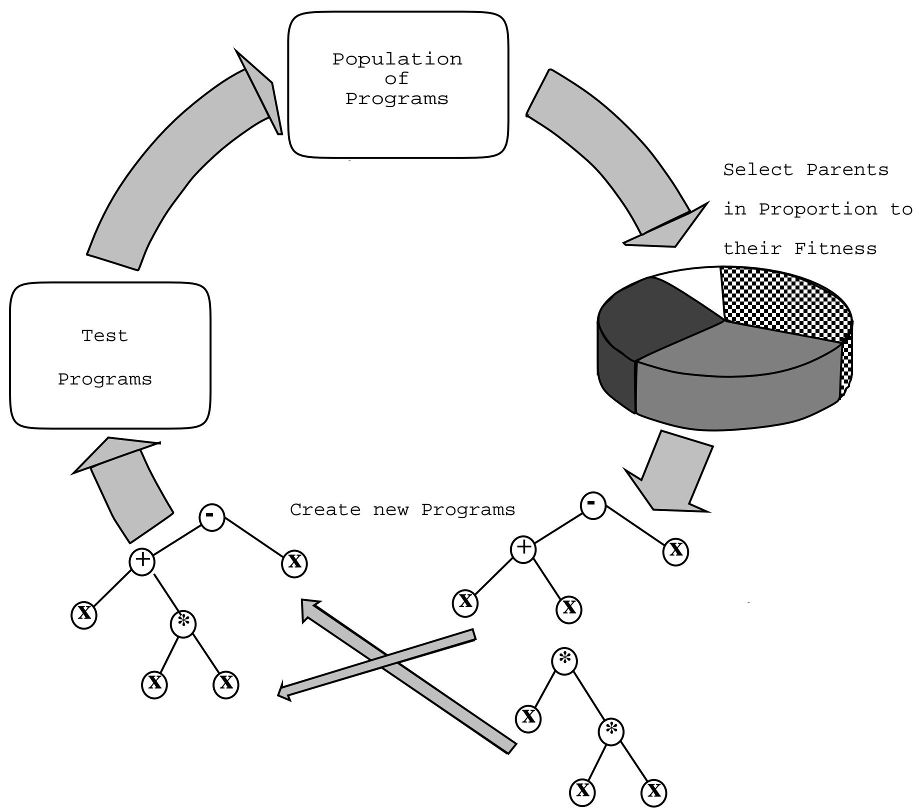

Figure 1.2. Genetic Programming Cycle

1.2 MOTIVATION

There are three main goals of this work. Firstly to show that data structures, other than simple random access indexed memory, can be used within genetic programming. Secondly to show that appropriate data structures can be beneficial when evolving programs and finally to show that appropriate data structures can be evolved as needed. As we shall see, the first two goals have been achieved. While we shall show it is is possible to evolve data structures on their own, and it is believed evolving them as needed is achievable (Section 7.5 offers some support) this has yet to be demonstrated.

We will show:

1. that abstract data types (stacks (Chapter 4), queues (Chapter 5) and lists (Chapter 6)) can be evolved using genetic programming,

2. on a number of different problems, an appropriate abstract data type is beneficial (Chapter 7),

3. GP can evolve general programs which solve the nested brackets problem (Section 7.1), recognise a Dyck context free language (Section 7.2) and evaluate Reverse Polish Notation (RPN) expressions (i.e. evolve a four function calculator, Section 7.3).

4. Chapter 2 contains a survey of GP, while a critical review of experiments with evolving memory is presented in Section 7.4).

5. Finally Appendix C describes investigations of real world electrical network maintenance scheduling problems that demonstrate that Genetic Algorithms can find low cost viable solutions to such problems.

1.3 OUTLINE

Following this introductory chapter, Chapter 2 describes in general terms the genetic programming technique and then Chapter 3 covers in some detail the specific techniques used in the remainder of the book. The next four chapters describe experiments. The knowledgeable reader may wish to commence with the experimental chapters, i.e. Chapter 4, and follow the references back to sections within Chapters 2 or 3 as necessary.

Chapter 4 describes in detail an experiment which shows it is possible to automatically generate programs which implement general stack data structures for integers. The programs are evolved using genetic programming guided only by how well candidate solutions perform. NB no knowledge of the internal operation of the programs or comparison with an ideal implementation is used. The two trees per individual in the population introduced by [Koza, 1992, Sections 19.7 and 19.8] is extended to five trees, one per stack operation. Chapters 6 further extends it to ten trees plus shared automatically defined functions (ADFs). Chapter 4 concludes by considering the size of the test case (in terms of its information content in the [Shannon and Weaver, 1964] sense) and the size of the evolved programs. The general solutions evolved are smaller than the test case, i.e. they have compressed the test case.

Chapter 5 describes a series of experiments which show genetic programming can similarly automatically evolve programs which implement a circular "First-In First-Out" (FIFO) queue. Initially memory hungry general solutions evolved but later experiments show that adding resource consumption as a component of the fitness function enables memory efficient solutions to be evolved. The final set of experiments show FIFO queues can be evolved from basic primitives but considerably more machine resources are required. Mechanisms are also introduced to constrain the GP search by requiring evolving functions (ADFs) to obey what a software engineer would consider sensible rules.

In Chapter 6 the last data structure, an integer list, is evolved. A list is a generalisation of both a stack and a queue but more complex than either. A controlled iteration loop and syntax rules are introduced. The evolution of the list proves to be the most machine resource intensive of the successful experiments in our book. Chapter 6 also describes a model for the automatic maintenance of software produced by GP. In one experiment considerable saving of machine resources is shown.

Chapter 7 is the crux of the book. It shows in three cases GP can beneficially use appropriate data structures in comparison to using random access memory. The three problems are the balanced bracket problem, a Dyck language (i.e. balanced bracket problem but with multiple types of brackets) and evolving a reverse polish expression calculator.

Chapter 8 stands back from the experiments and considers in some detail the dynamics of GP populations using the runs from Chapter 4 as an example. Chapter 8 starts by considering the application of results from theoretical biology. It concludes Price's theorem of selection and covariance can, in general, be applied to genetic algorithms and genetic programming but the standard interpretation of Fisher's fundamental theorem of natural selection cannot. The remainder of Chapter 8 investigates the reasons behind the small proportion of successful runs in the stack problem. It concludes the presence of easily found "deceptive" partial solutions acts in many cases via fitness based selection to prevent the discovery of complete solutions. Partial solutions based upon use of memory are readily disrupted by language primitives which act via side-effects on the same memory. This leads to selection acting against these primitives, which in most cases causes their complete removal from the population. However where complete solutions are found, they require these primitives and thus in most runs complete solutions are prevented from evolving by the loss of essential primitives from the population. While the details of the mechanism are specific to the stack problem, the problem of "deceptive" fitness functions and language primitives with side-effects may be general.

The stack populations are also at variance with published GP results which show variety in GP populations is usually high (in contrast to bit string genetic algorithm populations which often show convergence). With the stack populations in many cases there are multiple identical copies within the population. This is due to the discovery of high fitness individuals early in the GP run which contain short trees. With short trees many crossover operations produce offspring which are identical to their parents and these tend to dominate the population so reducing variety. This effect may be expected in any GP population where high fitness solutions contain short trees but are fragile, in that most of their offspring have a lower fitness. The presence of code within

the trees which does not affect the trees performance (variously called “fluff”, “bloat” or “introns”) may conceal this effect as trees need not be short and many offspring may be functionally identical to their parents (and so have the same fitness) but not be genetically identical. Should these dominate the population then it will have high variety even though many individuals within it are functionally the same.

The concluding chapter, Chapter 9, is followed by an extensive bibliography and then appendices. Appendix A tabulates the resources consumed in terms of number of trial solutions processed by the previous experiments. Appendix B contains a glossary of evolutionary computation terms. This is followed by Appendix C which details experiments using a permutation based genetic algorithm and others using genetic programming, to produce low cost schedules for preventive maintenance of the high voltage electrical power transmission network in England and Wales (the National Grid). The final appendix contains notes on the code implementation and network addresses from which it may be obtained.