Genetic Programming –

Computers using "Natural Selection" to generate programs

William B. Langdon, Adil Qureshi

Dept of Computer Science,
University College London.

ABSTRACT

Computers that "program themselves"; science fact or fiction? *Genetic Programming* uses novel optimisation techniques to "evolve" simple programs; mimicking the way humans construct programs by progressively re-writing them. Trial programs are repeatedly modified in the search for "better/fitter" solutions. The underlying basis is Genetic Algorithms (GAs).

Genetic Algorithms, pioneered by Holland [Hol92], Goldberg [Gol89] and others, are evolutionary search techniques inspired by natural selection (i.e survival of the fittest). GAs work with a "population" of trial solutions to a problem, frequently encoded as strings, and repeatedly select the "fitter" solutions, attempting to evolve better ones. The power of GAs is being demonstrated for an increasing range of applications; financial, imaging, VLSI circuit layout, gas pipeline control and production scheduling [Dav91]. But one of the most intriguing uses of GAs - driven by Koza [Koz92] - is automatic program generation.

Genetic Programming applies GAs to a "population" of programs - typically encoded as tree-structures. Trial programs are evaluated against a "fitness function" and the best solutions selected for modification and re-evaluation. This modification-evaluation cycle is repeated until a "correct" program is produced. GP has demonstrated its potential by evolving simple programs for medical signal filters, classifying news stories, performing optical character recognition, and for target identification.

This paper surveys the exciting field of Genetic Programming. As a basis it reviews Genetic Algorithms and automatic program generation. Next it introduces Genetic Programming, describing its history and describing the technique via a worked example in C. Then using a taxonomy that divides GP researchs into theory/techniques and applications, it surveys recent work from both of these perspectives.

Extensive bibliographies, glossaries and a resource list are included as appendices.

Keywords – Automatic Programming, Machine Learning, Genetic Algorithms, Genetic Programming.

Contents

1 Introduction

Genetic programming is a technique pioneered by John Koza which enables computers to solve problems without being explicitly programmed. It works by using John Holland's genetic algorithms to automatically generate computer programs.

Genetic algorithms were devised by Holland as a way of harnessing the power of 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 modeled by a string representing its DNA. This population is "evolved" by repeatedly selecting the "fitter" solutions and producing new solution from them (cf. "survival of the fittest"). The new solutions replacing existing solutions in the population. New individuals are created either asexually (i.e. copying the string) or sexually (i.e. creating a new string from parts of two parent strings).

In genetic programming 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 ensures that the new program is also a tree and so is also syntactically valid.

As an example, suppose we wish a genetic program to calculate $y = x^2$. Our population of programs might contain a program which calculates $y = 2x - x$ (see figure 1) and another which calculates $y = \frac{x}{\frac{\frac{x}{x}}{x-x^3}} - x$ (figure 2). Both are selected from the population because they produce answers similar to $y = x^2$ (figure 4), i.e. they are of high fitness. When a selected branch (shown shaded) is moved from the father program and inserted in the mother (displacing the existing branch, also shown shaded) a new program is produced which may have even high fitness. In this case the resulting program (figure 3) actually calculates $y = x^2$ and so this program is the output of our GP.

The remainder of this paper describes genetic algorithms in more detail, placing them in the context of search techniques, then explains genetic programming, its history, the five steps to GP, shows these steps being used in our example and gives a taxonomy of current GP research and applications. Current GP research and applications are presented in some detail.

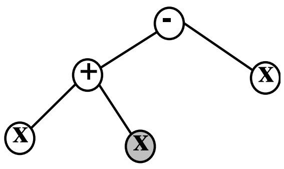

Figure 1: Mum, fitness .64286, $2x - x$

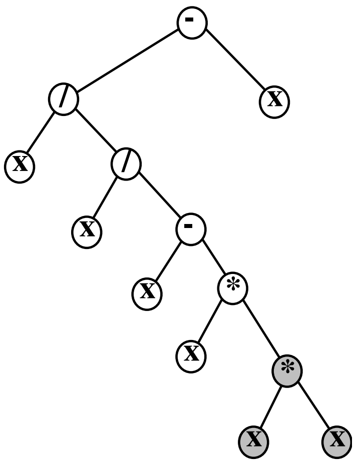

Figure 2: Dad, fitness .70588, $\frac{x}{\frac{x}{x-x^3}} - x$

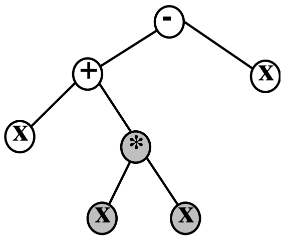

Figure 3: Correct Program, fitness 1.0, $x + x^2 - x$

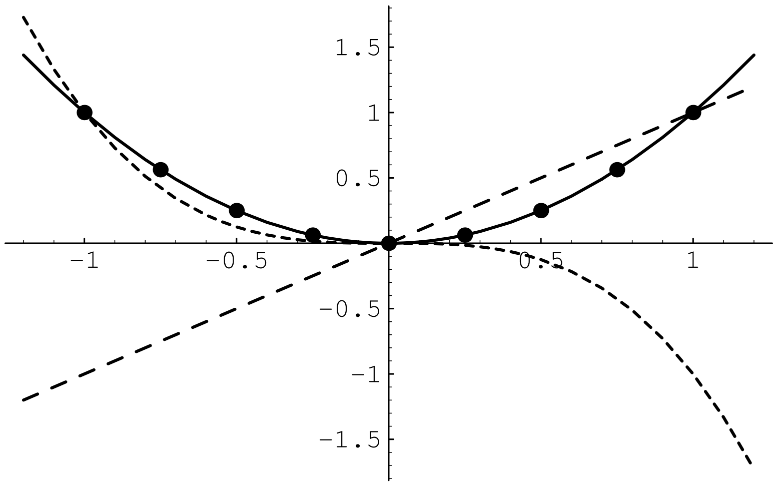

Figure 4: x^2 (solid), test points (dots), values returned by *Mum* (dashed) and *Dad* (small dashed)

1.1 Automatic Program Generation

A computer program can be thought of as a particular point within a search space of all such programs and so computer programming can be thought of as searching this space for a suitable program. Human programmers exercise their skills to direct their search so they find a suitable program as quickly as possible. There are many tools (such as high level languages and code generators) which transform the search space to make it easier for the human navigator to find his goal.

To automatically generate programs we then need an algorithm that can navigate program search spaces and find suitable programs with the minimum amount of effort. The program search space however is likely to be infinitely large, even when only considering programs that are syntactically correct. In the next section we look at a number of search techniques and discuss their suitability for navigating a program search space.

1.2 Search Techniques

There are a large number of well established search techniques in use within the information technology industry, figure 5 categorises them.

Enumerative techniques, in principle, search every possible point one point at a time. They can be simple to implement but the number of possible points may be too large for direct search. In some cases, eg game playing [UvA89], it is possible to curtail the search so that points in regions which cannot contain the solution are not checked.

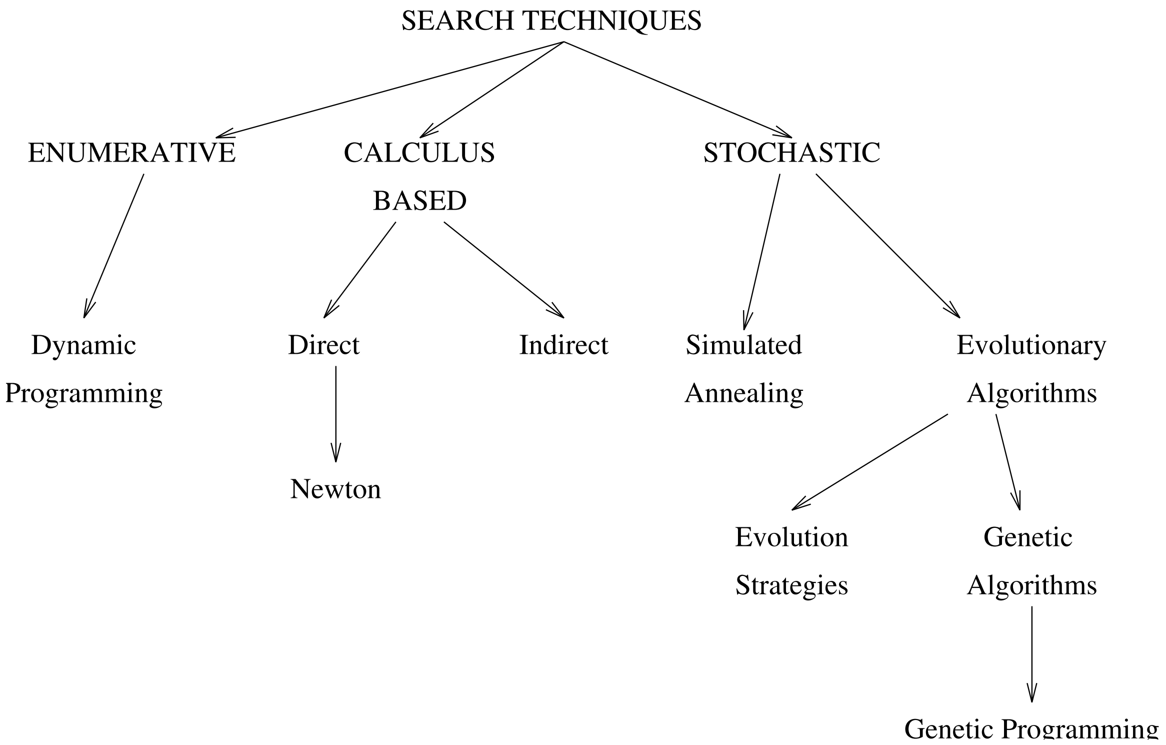

Figure 5: Search Techniques

Calculus based techniques treat the search space as a continuous multi-dimensional function and look for maxima (or minima) using its derivative. **Indirect** methods use that fact that at the extrema the function's derivative is zero. Where the function is smooth the volume of space where its derivative is zero can be a very small sample of the whole search space.

Direct calculus techniques such **Newton** use the gradient/function values to estimate the location of nearby extrema. These techniques, and others, are known as *Hill Climbing* techniques because they estimate where a maximum (i.e. hill top) lies, move to that point, make a new estimate, move to it and so on until they reach the top of the hill. These techniques can be used upon "well behaved" problems or problems which can be transformed to become "well behaved".

Stochastic search techniques use information from the search so far to guide the probabilistic choice of the next point(s) to try. They are general in their scope, being able to solve some very complex problems that are beyond the abilities of either enumerative or calculus techniques. **Simulated annealing** searches for minimum energy states using an analogy based upon the physical annealing process, where large soft low energy crystals can be formed in some metals (eg copper) by heating and then slow cooling [KGV83].

Evolutionary algorithms are based upon Darwin's Natural Selection theory of evolution, where a population is progressively improved by selectively discarding the worse and breeding new children from the better. In **Evolutionary Strategies** points in the search space are represented by a vector of real values. Each new point is created by adding random noise to the current one. If the new point is better search proceeds from it, if not the older point is retained[1]. In contrast, a **genetic algorithm** represents points in the search space by a vector

[1]Historically Evolutionary Strategies search only one point at a time but more recently they have become more like genetic algorithms by using a population of points[BHS91].

of discrete (typically) bit values. Each new child is produced by combining parts of the bit vector from each parent. This is analogous to the way chromosomes of DNA (which contains the inherited genetic material) are passed to children in natural systems.

When programming neural networks, calculus based search techniques such as back propagation are often used[2]. Such search techniques are possible because the search space has been transformed (and simplified) so that it is smooth. When the search has succeeded the neural network is said to have been trained, ie a suitable combination of connection weights has been found. Note that a program (a neural network) has been automatically generated by searching for it.

An alternative to implementing a general transformation that converts the vast discrete space of possible programs into a continuous one (which is sufficiently well behaved as to allow it to be searched by calculus based techniques) is to search the original space itself. The discrete nature of this space prevents the use of calculus based techniques and the vast number of possible programs make enumerative search infeasible, however there has been some success with some stochastic search techniques.

2 Genetic Algorithms

Genetic algorithms are perhaps the closest computation model to natural evolution. Their success at searching complex non-linear spaces and general robustness has led to their use in a number of practical problems such as scheduling, financial modeling and optimisation.

The inventor of genetic algorithms, John Holland [Hol92], took his inspiration for them from nature. Genetic algorithms contain a population of individuals, each of which has a known fitness. The population is evolved through successive generations, the individuals in each new generation are bred from the fitter individuals of the previous generation. Unlike natural evolution which continuous indefinitely, we have to decide when to stop our GA. As with the breeding of domestic animals, we choose the individuals to breed from (using a *fitness function*) to drive the population's evolution in the direction we want it to go. As with domestic animals, it may take many generations to produce individuals with the required characteristics.

Inside a computer an individual's fitness is usually calculated directly from its DNA (ie without the need to grow it) and so only the DNA need be represented. Usually genetic algorithms represent DNA by a fixed length vector. Where a genetic algorithm is being used for optimisation, each individual is a point in the search space and is evaluated by the fitness function to yield a number indicating how good that point is. If any point is good enough, the genetic algorithm stops and the solution is simply this point. If not then a new population, containing the next generation, is bred.

The breeding of a new generation is inspired by nature; new vectors are bred from the fitter vectors in the current generation, using either asexual or sexual reproduction. In asexual reproduction, the parent vector is simply copied (possibly with random changes, ie mutations). Figure 6 shows a child vector being created by mutating a single gene (in this case each gene is represented by a single bit). With sexual reproduction, two of the fitter vectors are chosen and the new vector is created by sequentially copying sequences alternately from each parent. Typically only two or three sequences are used, and the point(s) where the copying crosses over to the other parent is chosen at random. This is known as *crossover*. Figure 7 shows a child

[2]Simulated Annealing, Evolution Strategy and Genetic algorithms have also been used to program artificial neural networks.

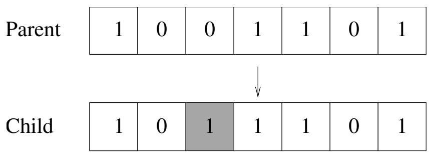

Figure 6: Genetic Algorithms - Mutation

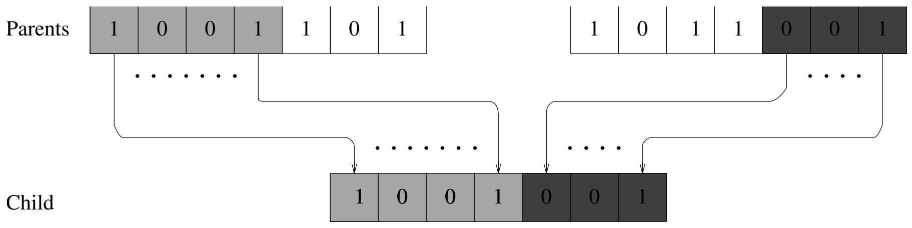

Figure 7: Genetic Algorithms - Crossover

being formed firstly by copying four genes from the left-hand parent then the three remaining genes are copied from the right-hand parent. Figure 8 shows the genetic algorithm cycle.

Holland in his paper “Genetic Algorithms and the Optimal Allocation of Trials” [Hol73] shows, via his schemata theorem, that in certain circumstances genetic algorithms make good use of information from the search so far to guide the choice of new points to search. Goldberg [Gol89] gives a less mathematical treatment of the schemata theorem.

The schemata theorem requires the vector representation and fitness function be designed so that the required solution can be composed of short fragments of vectors which, if present in a vector, give it a relatively high fitness regardless of the contents of the rest of the vector. These are known as *building blocks.* They can be thought of as collections of genes which work well together.

Given that a solution which can be decomposed into high fitness sub-solutions , i.e building blocks, exists, genetic algorithms, even starting from a random collection of vectors, can progressively select the vectors with building blocks and using the crossover operator gradually splice these together until the population contains vectors which are substantially correct.

Table 1 lists the stochastic search techniques including GAs, used with the classes of programming languages to automatically generate programs. Clearly, work has been focused on using genetic algorithms, leaving the techniques relatively untouched. The remainder of the paper concentrates on the recent success achieved by combining genetic algorithms with traditional programming. The term *genetic programming* was suggested by David Goldberg to describe the automatic production of programs by using a hierarchical genetic algorithm to search the space of possible programs.

3 Genetic Programming

3.1 History

The idea of combining genetic algorithms (GAs) and computer programs is not new, it was considered in the early days of genetic algorithms. However John Holland’s work on combining genetic algorithms and production languages, rather than tradition computer languages, was more actively pursued. This work led to classifier systems.

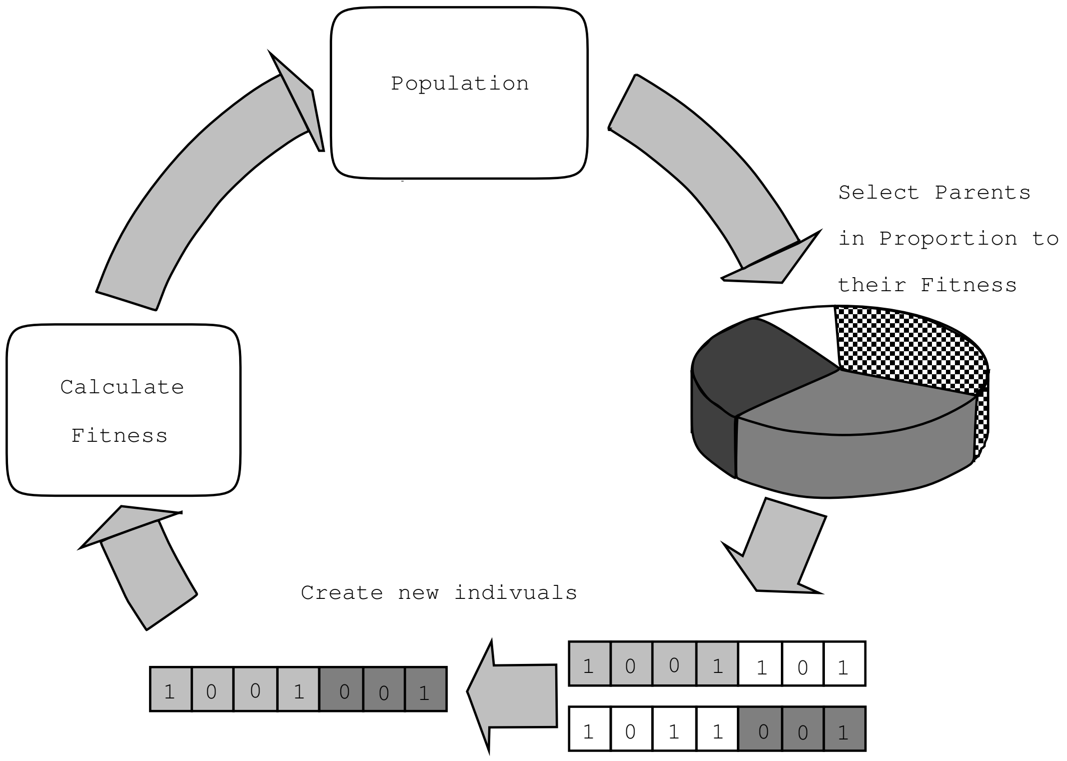

Figure 8: The Genetic Algorithm Cycle

	Class of Programming Language			
Search Technique	Traditional	Logic (eg Prolog)	Expert Systems (Rule based)	Automata
Simulated Annealing	O'Reilly[3]			
Evolution Strategy				Evolution Program
Genetic Algorithms	Genetic Programming	Nguyen[4]	BEAGLE Classifiers[5]	Self[6]

Table 1: Automatic Programming using Stochastic Search

Richard Forsyth's BEAGLE [For81] evolves programs (rules) using a (GA like) algorithm. The rules are tree structured boolean expressions, including arithmetic operations and comparisons. BEAGLE's rules use its own specific language, however Forsyth did suggest at some point in the future Lisp might be used. BEAGLE is now commercially available as an expert system (see section 5.1).

Nichael Cramer also applied genetic algorithms directly to special computer programming languages [Cra85] and Fujiki and Dickinson used genetic algorithms on a subset of LISP [FD87].

[3]Una-May O'Reilly has compared GP, Simulated Annealing and Stochastic Iterated Hill Climbing [OO94]

[4]Nguyen has demonstrated evolving programs written in Mathematica [NH94] and some work has been done on using genetic programming with Prolog [Nor94b]

[5]In Classifiers [HHNT86] a genetic algorithm, plus other techniques, is used to automatically generate both a rule base and its interconnections. Feldman [Fel93], amongst others, has used genetic algorithms to evolve aspects of Fuzzy Logic systems.

[6]Self demonstrated a genetic algorithm evolving a Turing Machine which recognised three bit even parity but had less success with other more complex problems [Sel92].

It was John Koza [Koz92] who successfully applied genetic algorithms on LISP to solve a wide range of problems. Techniques like his have come to be known as "genetic programming". Koza claims ([Koz92] page 697) genetic programming to be the most general machine learning paradigm.

A simple approach of breeding machine code and even FORTRAN source code which ignores the syntax fails because the programs produced are highly unlikely to compile or run, let alone approach the required solution [De 87]. There are countless examples of apparently minor syntax errors causing programs to misbehave in a dramatic fashion. These two facts foster the common belief that all computer programs are fragile. In fact this is not true; many programs are used and yield economic benefit despite the fact that they are subject to many minor changes, introduced during maintenance or version upgrades. Almost all programs are produced by people making progressive improvements.

Genetic programming allows the machine to emulate, to some extent, what a person does, i.e. to make progressive improvements. It does this by repeatedly combining pairs of existing programs to produce new ones, and does so in a way as to ensure the new programs are syntactically correct and executable. Progressive improvement is made by testing each change and only keeping the better changes. Again this is similar to how people program, however people exercise considerable skill and knowledge in choosing where to change a program and how; genetic programming, at present, has no such knowledge and must rely on chance and a great deal more trial and error.

Many variations of the basic genetic algorithm have been tried; in genetic programming the fixed length vectors are replaced by programs. Usually, a tree (or structured or hierarchical) representation of the program is used and individuals in the population are of different sizes. Given the new representation, the new genetic operators such as tree crossover must be used. As figure 9) shows tree crossover acts within branches of the tree to ensure that the the programs it produces are still trees and have a legal syntax. Thus genetic programming is fundamentally different from simply shuffling lines of Fortran or machine code. The sequence of operations in genetic programming is essentially as that for other genetic algorithms (see Figure 10).

3.2 Basic Choices

Koza says there are five[7] preliminary steps to solving a problem using genetic programming; choosing the terminals (1), the functions (2), the fitness function (3), the control parameters (4) and the termination criterion (5).

In Koza's terminology, the terminals (1) and the functions (2) are the components of the programs. In figure 9 functions $+$, $-$ and $*$ are used, they form the junctions in the tree. In figure 9 the only terminal is x which forms the end leafs. The connections between the terminals and functions indicate the order in which operations are to be performed. For example the top left tree in figure 9 shows a program which calculates $(x * x) + (x + (x - x))$. Note how the brackets, which denote the order of evaluation, correspond to the structure of the tree.

The choice of components of the program (ie terminals and functions) and the fitness function (3) largely determine the space which genetic programming searches, consequently how difficult that search is and ultimately how successful it will be.

[7]In his later work [Koz94a] Koza adds a sixth step; determining the architecture in terms of the program's automatically define functions (ADFs). However he also shows that it is possible (but not necessarily easy) for correct programs to be evolved even with arbitrary choices for his six steps. We shall not dwell on ADFs here.

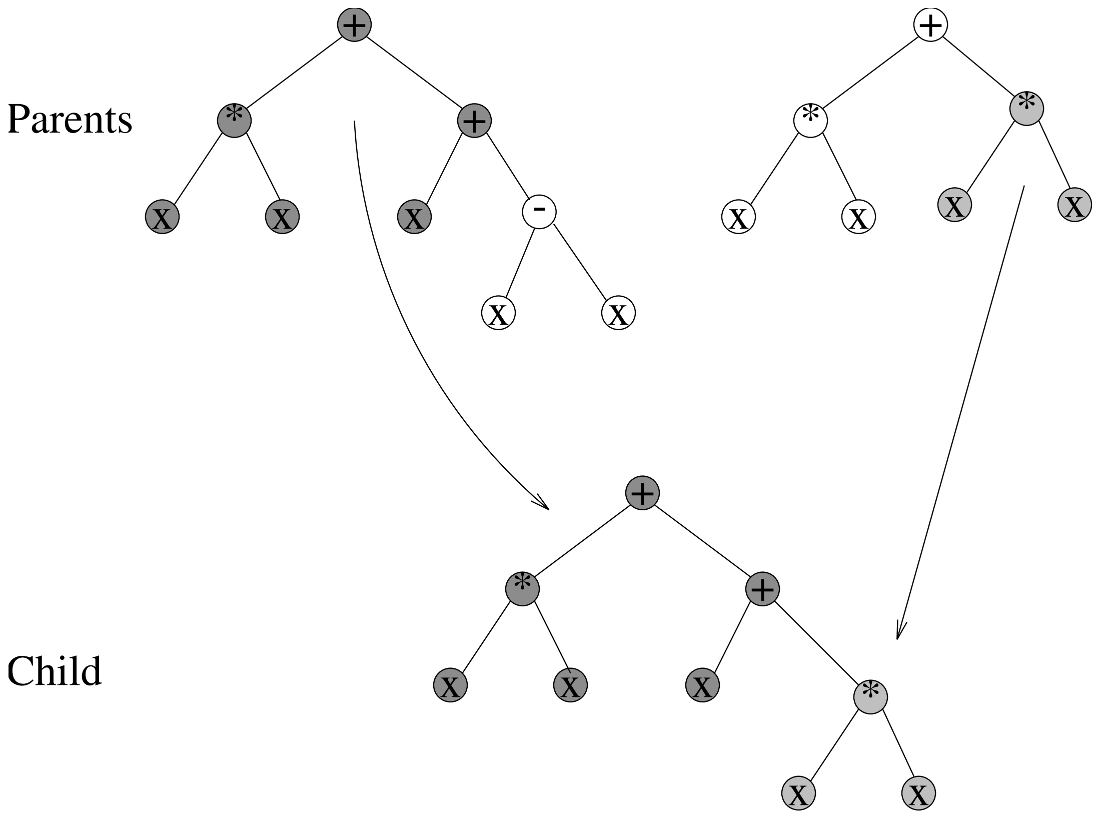

Figure 9: Genetic Programming Crossover:
$x^2 + (x + (x - x))$ crossed with $2x^2$ to produce $2x^2 + x$.

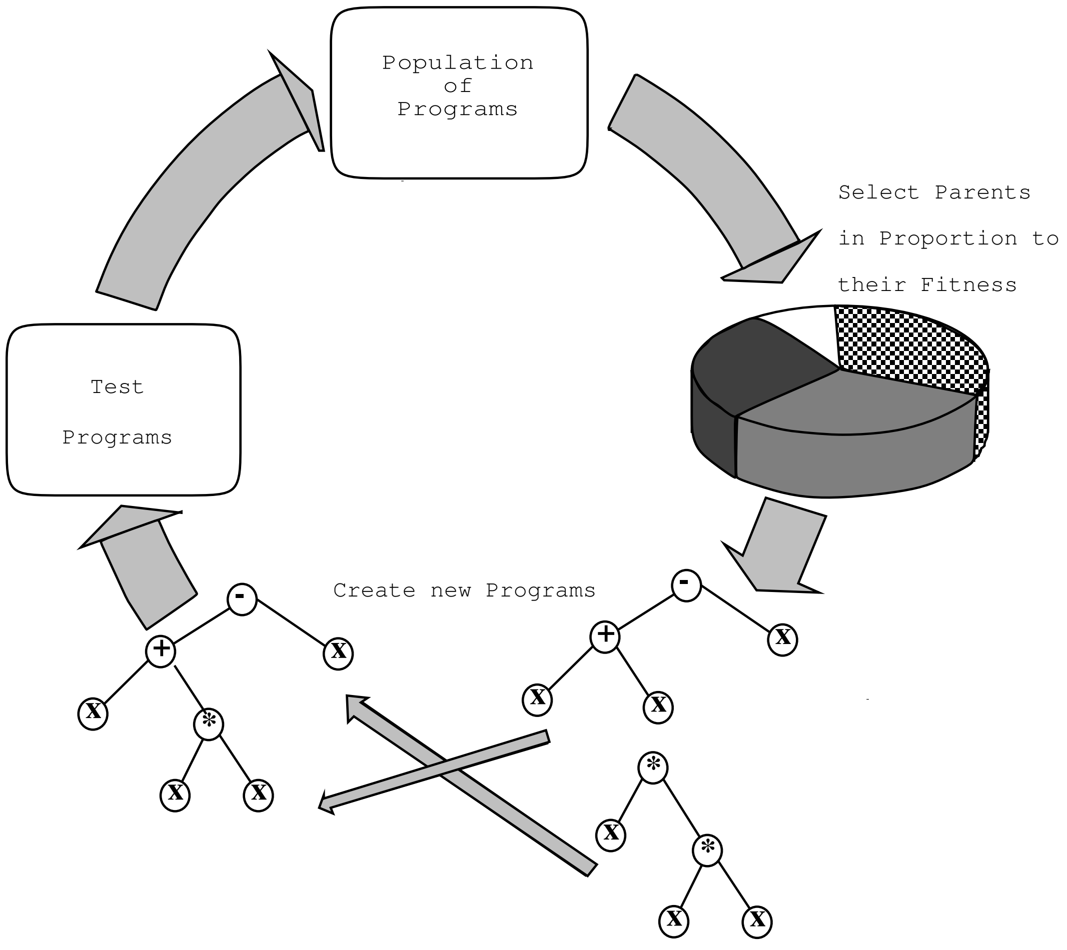

Figure 10: Genetic Programming Cycle

The control parameters (4) include the size of the population, the rate of crossover etc. The termination criterion (5) is simply a rule for stopping. Typically the rule is to stop either on finding a program which solves the problem or after a given number of generations, eg 50.

3.3 Example

In this sub-section we will outline a very simple example of genetic programming. We will use it to perform symbolic regression on a set of test values. That is we will find a formula (symbolic expression, program) whose result matches the output of the test values.

One use of symbolic regression is prediction, for once such a formula has been found, it can be used to predict the output given a new set of inputs. Eg given a history of stock prices, symbolic regression may find a formula relating the price on the next day to earlier ones, so allowing us to predict the price tomorrow.

In our very simple example the test values are related by the formula $y = x^2$. [8]

Following the five steps outlined in the previous section:

[8]The C code for this example is available via anonymous ftp from cs.ucl.ac.uk, directory genetic/gp-code/simple. The interested reader is invited to copy this code and try their own examples. Similar polynomials eg x^3, $x + x^2$, $x^4 - x^3 + x^2 - x$ may be produced by changing the test values and possibly population size.

1. The leafs (terminals) on our programming trees will be the input value, x.

2. Our program will use the four floating point arithmetic operators $+$, $-$, $\div$ and $\times$.

   Our choice is guided by the expectation that output will be a simple polynomial of x. If we guess wrongly then it may not be possible to devise a correct program given our functions and terminals. We can add others but this will increase the number of possible programs to search, which might slow down the search.

   The terminals and functions are chosen so that any function can have as any of its arguments any terminal or any function call. This allows programs to be assembled from any mixture of terminals and functions (this property is known as closure, see also section 4.1.2). In our case all terminals and functions are of type "`float`". To avoid divide by zero errors, we define divide by zero to be unity. Note as all the functions have two operands, the programs evolved will each have the form of a binary tree.

3. Fitness is calculated by executing each program with each of nine x values and comparing each answer with the corresponding y value. The nine (x, y) pairs are chosen so that $y = x^2$. They are shown as dots in figure 14. To yield a number which increases as the program's answer gets closer to the y value, 1.0 is added to the absolute difference between the programs value and the corresponding y and the result is inverted[9]. The final fitness is the mean of all nine calculations.

   If the program is within either 0.01 or $0.01 \times y$ we call this a hit, indicating that this is close enough.

4. We chose control parameters so that:

   - there are 16 individuals in the population;
   - on average 60% of new individuals are created by crossover;
   - of the other 40%, 99% are direct copies from the previous generation and 1% (ie .004) are mutated copies;
   - our programs contain no more than 25 nodes (ie no more than 12 functions and 13 terminals).

5. We stop when we have a found a program that correctly matches all nine test points (ie there are nine hits) or when we reach 50 generations.

On one run the following 100% correct code was generated:

```
float gp( float x )
{
  return ((x+(x)*(x))-(x));
}
```

By noting `x-x` is zero and removing excessive brackets it can be seen that the return statement is equivalent to `return (x*x);` ie C code to calculate x^2.

This 100% correct program is shown in Figure 13. It was produced by crossover between the two programs shown in Figures 11 and 12, both of which were of above average fitness. The subtrees affected by the crossover are shown shaded. It can be seen that the net effect of crossover is to replace the middle x with a subtree which calculates x^2 and this yields a program which is exactly equivalent to x^2.

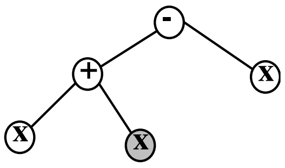

Figure 11: Mum, fitness .64286, $2x - x$

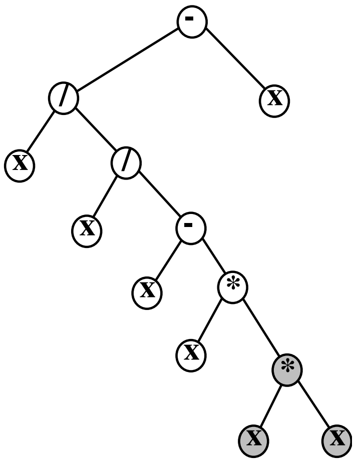

Figure 12: Dad, fitness .70588, $\frac{x}{\frac{x}{x-x^3}} - x$

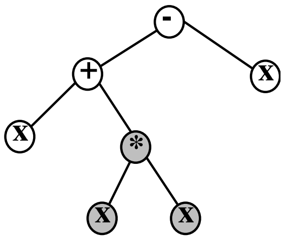

Figure 13: Correct Program: $x + x^2 - x$

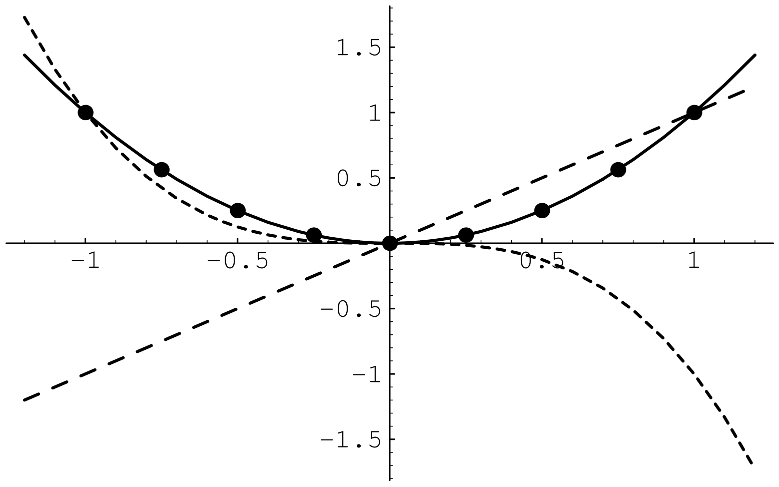

Figure 14: x^2 (solid), test points (dots), values returned by *mum* (dashed) and *dad* (small dashed)

Figure 14 shows the values calculated by the two parent programs against the test points (which their offspring calculates exactly).

Having defined in detail the concepts of GP, in the next section we look at the some of the active areas of research in Genetic Programming.

3.4 Genetic Programming Research

GP research can be broadly categorised into two groups:–

- GP Techniques and Theory
- GP Applications

The first group classifies research that is directed at the development and understanding of the theory and techniques of genetic programming. The second group concerns research which is directed mainly at the application of the GP technique. In the next two sections we look at research in each of these categories.

[9]Adding 1.0 avoids the possibility of divide by zero.

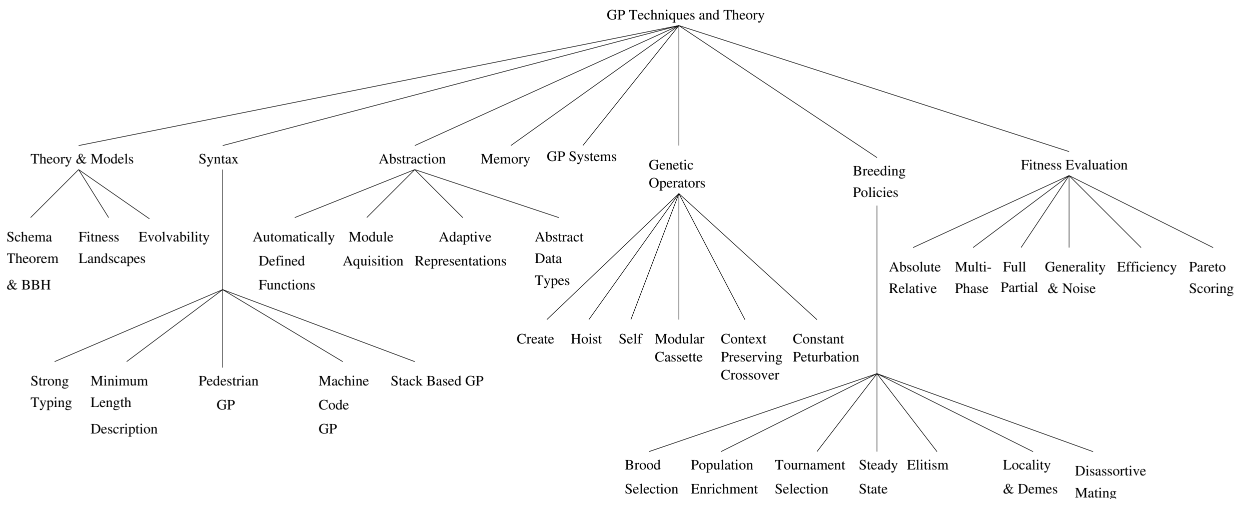

Figure 15: A Taxonomy of GP Techniques and Theory

4 GP Techniques and Theory

There is significant scope for research into the theory and techniques used in the GP paradigm. A taxonomy of the current areas of research is shown in figure 15 and detailed below. It should be noted that some of the techniques mentioned in this section draw their inspiration from work done in GAs. Table 2 lists exotic genetic algorithm techniques and indicates which have been adopted by GP researchers. As can been seen many of these have been tried, but some are yet to be explored.

Genetic Algorithms Technique	Genetic Programming Reference
Fitness Scaling	[IdS94]
Rank and Tournament Selection	[Ang94] [Koz94a]
Co-evolution	[AP93] [Sie94][Koz92]
Steady State Populations	[Rey92] [TC94b] [Koz94a]
Parallel processors	[Sin93]
Inversion and Permutation	[Koz92]
Diplodity, Dominance and Abeyance	[Ang94]
Introns, Segregation, Translocation and Multiple Chromosomes	[Ang94]
Duplication and Deletion	
Sexual Determination and Differentiation	
Speciation, Restrictive Mating, Demes and Niches	[TC94b] [DB94] [Abb91] [Rya94]
Multiobjectives	
Hybrids	[ZM93] [Gru93]
Knowledge-augmentation	
Approximate function evaluation	[Koz92]

Table 2: Exotic GA Techniques Applied to Genetic Programming

4.1 Syntax

In this section we look at various syntactical variations to the standard GP approach.

4.1.1 Pedestrian GP

Banzhaf ([Ban93]) has devised a GP system using a traditional GA binary string representation. The mechanisms of transcription, editing and repairing are used to convert these bit strings into computer programs. Each individual in the population consists of a string of 225 bits. The 225 bits are interpreted as sequence of 5-bit words. A transcription table is used to map these 5-bit words to functions and terminals. The bitstrings are made to satisfy two requirements by a repair operator. These requirements are that the number of terminals in the resulting program be greater than the number of functions, and secondly that each sequence begins with a function call. Bit level mutation and crossover genetic operators were used to generate new individuals. Banzhaf has used this approach to successfully evolve programs that predict simple integer sequences.

4.1.2 Strongly Typed GP

In almost all genetic programming work the terminals and functions are chosen so that they can be used with each other without restriction. This is achieved by requiring that the system be closed so that all terminals are of one type and all functions can operate on this type and always yield a result also of this type. For example measures are taken to protect against divide by zero errors.

In Montana's Strongly Typed Genetic Programming (STGP) [Mon94] a genetic program can contain multiple different types simultaneously. When manipulating the program (eg using crossover) care is taken that not only are there the correct number of arguments for each function but that they are of the expected type. This leads to an increase in the complexity of the rules that must be followed when selecting crossover points. However Montana shows that tight type checking can considerably reduce the size of the search space, which he says will reduce the effort to find the solution. Generic types are allowed in order to reduce the number of functions which must be explicitly specified.

4.1.3 Minimum Description Length

One of de Jong's arguments against the combination of traditional general purpose computer programming languages and genetic algorithms [De 87] was that the behaviour of program statements written in such languages depend upon statements before it. Iba, de Garis and Sato[10] have considered the problems of symbolic regression and classification using problem specific languages which are better behaved with respect to changes in the order of statements. In one case (GMDH) they show that earlier statements cannot reduce the value of latter ones. Such languages fit well with the schema theorem if we view good sub-trees like good schema. These languages ensure that good overall individuals can be constructed of good schema, thus we should expect a genetic algorithm to perform well.

Iba etal use Quinlan's "decision trees", which are designed to classify sets of inputs. They allocate fitness based upon Quinlan's "Minimum Description Length" (MDL) which gives a

[10]Their work is spread over several publications, perhaps [IdS94] gives the most up to date summary.

natural means of basing fitness not just upon how well the program (or decision tree, in this case) performs but also on its size. Other investigators have used arbitrary combinations of program size and score to calculate a composite fitness. MDL is calculated by considering how many bits are needed to code the program and how many to code a description of its error, ie those cases where it returns the wrong answer.

$MDL = Tree\_Coding\_Length + Exception\_Coding\_Length$

where

$Tree\_Coding\_Length = (n_t + n_f) + n_t \log_2 T + n_f \log_2 F$

n_t is the total number of terminals and n_f is the total number of function calls in the tree. T is the number of terminals in the terminal set and similarly F is the size of the function set.

$Exception\_Coding\_Length = \sum_{x \in Terminals} L(n_x, w_x, n_x)$

The summation is taken over all leafs in the program. n_x is the number of cases represented by the particular leaf, x and w_x is the number of such cases which are wrongly classified. $L(n, k, b)$ is the total number of bits required to encode n bits given k are 1s and b is an upper bound on k:

Symbolic regression (as was mentioned in section 3.3) is the problem of devising a formula or program whose answer matches the output associated with a given set of inputs.

Iba etal's STROGANOFF system use an MDL based fitness together with program trees composed of GMDH primitives. The terminal GMDH primitives are the inputs on which we wish to perform symbolic regression. The GMDH functions each have a two inputs, their output is a quadratic function of the two inputs:

$G_{z_1,z_2}(z_1, z_2) = a_0 + a_1 z_1 + a_2 z_2 + a_3 z_1 z_2 + a_4 z_1^2 + a_5 z_2^2$

Good results have been obtained with both decision trees and STROGANOFF using small populations (e.g. 60).

4.1.4 Stack Based GP

In most genetic programming work the programs being evolved use a prefix tree structured syntax. For the most part the LISP language is used. This section describes experiments with other syntax.

Keith and Martin [KM94a] considered postfix and infix and hybrid (mixfix) languages but conclude that their postfix language and its interpreter are the more efficient.

Perkis [Per94a] uses a linear postfix language and claims some advantages over the more standard tree approach.

It is interesting that Cramer [Cra85] started with a linear postfix language (JB) but moved to a tree structured postfix language (TB). The move seems to have been partially due to fears that JB could form indefinite loops but also because TB avoided the implicit GOTOs of JB.

4.1.5 Machine Code GP

Nordin [Nor94a] has implemented a system using a very restricted set of machine codes and demonstrated, for his application, enormous (more than thousand fold) speed ups. However this has been at a considerable loss of generality:

1. there is no branching or looping
2. the program is of fixed length (12 instructions)
3. the program is specific to the SUN SPARC RISC instruction set
4. only a few instructions are used

It remains to be seen if such performance can be retained if a less restrictive approach is taken.

4.2 Abstraction

If we chart the development of programming languages, we see a steady trend of higher and higher levels of abstraction being applied. The earliest languages were machine code and were quickly replaced by assembly level languages. These in turn lead to the development of high level languages such as FORTRAN. The latter facilitated code reuse first in the form of subroutines and later in the more easily parameterised forms of functions and procedures. Useful functions were made available in the form of libraries which are themselves a higher form of functional abstraction. The introduction of object oriented languages which support data abstraction or abstract data types (ADTs) further increased the amount of code reuse.

The advantages of abstraction are that it allows the control of complexity and facilitates code reuse. Code reuse is particularly useful for GP because it means that a block of useful code need not be independently rediscovered at different places within the program where it is needed. In this section we look at a number of techniques that have been used to introduce abstraction into GP.

4.2.1 Automatically Defined Functions

ADFs are evolvable functions (subroutines) within an evolving genetic program, which the main routine of the program can call. Typically each ADF is a separate tree; consisting of its arguments (which are the terminals) and the same functions as the main program tree (possibly plus calls to other ADFs). If side effects are prohibited, ADFs act as functional building blocks. Crossover acting on the main program can then rearrange these building blocks within the program. ADFs can also be evolved, eg when crossover acts upon them. The overall format of the program is preserved by ensuring crossover and other genetic operations acts only within each ADF. That is code cannot be exchanged by crossover between ADFs and the main program[11]. The main program's function set is extended to allow (but not require) it to call the ADFs.

ADFs have been successfully used on problems that proved too difficult for genetic programming without ADFs.

4.2.2 Module Acquisition

The Genetic Library Builder (GLiB) [Ang94] implements encapsulation by adding a complementary pair of genetic operators, compression (encapsulation) and decompression which are

[11]Koza has demonstrated that it is possible to relax this restriction allowing crossover between program branches but the rules required to ensure the resulting offspring are still syntactically correct are much more complex.

applied to any point in the genetic program chosen at random. Compression takes (part of) the subtree at the given point and converts it into a function which is stored in a library. Those parts of the subtree not included in the function become its arguments. The original code is replaced by the function call. Decompression is the replacement of a function with its definition, ie it reverses the compression operator. Once code is encapsulated in a function it is protected from dissociation by crossover or other operators.

The functions produced may be any mixture of functions, terminals and its own arguments. Thus they need not be modular in the software engineering sense.

In one comparison [Kin94b] using EVEN-N-PARITY problems, it was found the ADF approach was superior to the standard non-ADF approach whereas no improvement was seen with encapsulation.

4.2.3 Adaptive Representations

Whereas with module aquisition, subtrees are chosen at random to become modules, Rosca and Ballard [RB94] perform an analysis of the evolutionary trace to discover building blocks. Building blocks are compact subtrees which add to the fitness of individuals. These fit blocks are parameterised and then added to the function set to extend the representation. The current population is then enriched by replacing individuals by new randomly created individuals using the new function set. In their experiments they compared the application of standard GP, GP with ADFs and GP with adaptive representations to EVEN-N-PARITY problems of difficulty ranging from N=3 to N=8. The results they report indicate superior performance compared to both of the other methods in terms of the number of generations required to find the solution and the size of the soulution found.

4.2.4 Abstract Data Types

Langdon [Lan95] used GP with index memory to successfully evolve queue and stack abstract data types (ADTs) using a chromosome structure with multiple trees. Each tree in the chromosome was used to represent one of the required ADT member functions. The results suggest that it may be possible to extend the level of abstraction in GP from pure functional abstraction as in ADFs, MAs and adaptive representations to full ADTs. This may help increase the scalability of GP.

4.3 Breeding Policies

A number of choices are determined by a breeding policy. These include the choice of parents, the number of offspring produced, the choice of which individuals in the population are to be displaced by newly created individuals. Research associated with such breeding policies are reported below.

It should be noted that many of the techniques in this section are designed to deal with a problem common to both GA and GP. This is the problem of premature convergence in which the which the population converges to a less than optimal solution as a result of loosing population diversity.

4.3.1 Brood Selection

In [TC94a] Tackett and Carmi investigate the implications of brood selection for GP. They describe brood selection as following phenomenon:-

"In nature it is common for organisms to produce many offspring and then neglect, abort, reabsorb, or eat some of them, or allow them to eat each other." The net effect of this behaviour is "the reduction of parental resource investment in offspring who are potentially less fit than others."

They then propose and show that this process can be used to reduce CPU and memory investment in an artificial genetic system. For this purpose they define a brood selection recombination operator $R_B(n)$ which they substitute for the standard GP crossover operator. The two operators are similar in that they both produce two offsprings from two parents. The difference is that internally, $R_B(n)$ produces n pairs of offspring instead of the single pair produced by the standard operator, but keeps only the two best. The choice of which pair of offspring survive is defined by a culling function F_B (brood fitness). The function F_B can either be equivalent or a simpler version of the normal fitness function

The results of their experiments (based on the "Donut Problem" [TC94b]) show that even for the case of F_B being the same as the normal fitness function, increasing the brood factor n, gives better performance than an equivalent increase in population size in standard GP, and hence significantly decreases the memory requirements. Furthermore, decreasing the cost of F_B also reduces the computational costs and surprisingly only very gradually reduces the performance of $R_B(n)$.

4.3.2 Population Enrichment

In [Per94b] Perry separates the evolution of a GP into enrichment and consolidated phases. In his experiments, the enrichment phase involved 40 runs with a population size of 2000 individuals, each processed for only 3 generations. The best of each of the 40 runs (the enriched population) were then loaded into a consolidated run with 1960 members of population generated at random. This GP population was then processed for 50 generations. The results indicate that one run of an enriched population produces better results than 3 traditional runs. In addition fewer individuals are processed, the solutions are more robust and parsimonious (concise).

4.3.3 Tournament Selection

In GA systems, individuals are chosen for reproduction from the population probablistically in proportion to their fitness. In tournament selection, a number of individuals are selected at random (dependent on the tournament size, typically 7) from the population and the individual with the best fitness is chosen for reproduction. Reverse tournament selection is sometimes used in steady state GP where, the individual with the worst fitness is chosen to be replaced by a newly reproduced individual. The tournament selection approach allows a tradeoff to be made between exploration and exploitation of the gene pool.

4.3.4 Steady State GP

In GP the population size is kept constant, so that when reproduction takes place, new individuals must replace existing individuals. If the number of individuals created is equal to the population size, then the entire population is replaced and an entire new generation is created. This approach is hence referred to as generational GP. If on the other hand, the number of individuals replaced is actually quite small, for example one, then the new population is actually a mix of new and old generations. This approach is called steady state GP(SSGP). The term “generation equivalent” is often used in SSGP and refers to the point at which an entire population has been created.

4.3.5 Elitism

Elitism refers to a breeding policies which try to preserve the best individuals in the population. The implication is that it is possible for highly fit individuals to never be replaced. SSGP is an elitist breeding policy since only the individuals with poor fitness are replaced by new individuals. Elitism is also sometimes implemented in generational GP.

4.3.6 Locality/Demes

The standard GP breeding policy is described as panmictic since it potentially allows any individual in the population to breed with any other individual. An alternative is to divide the population into subpopulations called “demes”, or islands. These demes are typically positioned on a two dimensional grid, and breeding is restricted so that any two parents have a much higher probability of being selected from within the same deme or from geographically close demes.

This approach has been investigated both by D’Haeseleer and Bluming in [DB94], and by Tackett and Carmi in [TC94b]. Tackett and Carmi use explicitly defined demes of different sizes for a problem with tunable difficulty (the donut problem). Their results suggest that use of demes significantly improves performance. In contrast D’Haeseleer and Bluming instead of using explicit demes, introduce locality in the form of geographic neighbourhoods, and select parents from the same neighbourhood and also place the child into the same neighbourhood. They have detected the spontaneous emergence of deme like structures in their results and report that the introduction of locality significantly improves population diversity which in conjunction with slightly increased generality of individuals, yields a substantial improvement in the generality of the population as a whole.

One of the advantages of the demic approach is that it is easy to implement using parallel or distributed architectures. Koza [KA95] has developed a next generation GP system using a network of transputers to exploit the parallelism inherent in the demic model. The results of his experiments with EVEN-N-PARITY problems indicated that more than linear speed-up in solving the problem using GP was obtained.

4.3.7 Disassortative Mating

In an effort to reduce premature convergence in the presence of elitism, Ryan ([Rya94]) introduces an algorithm that he calls the pygmy algorithm. Elitism has the benefit that it speeds the convergence to a solution, but at the cost of reducing the population diversity. This in turn can lead to premature convergence. The pygmy algorithm uses disassortative mating to maintain population diversity in the presence of the powerful selection pressure of elitism.

Disassortative mating is breeding policy that uses phenotypically different parents. In the pygmy algorithm, two lists of individuals, each list with a different fitness function and each sorted by fitness value are maintained. The individuals in one list are referred to as civil servants, and the individuals in the other are referred to a pymies. The list containing civil servants uses a fitness function equivalent to the performance of the individual, with ties resolved by rating parsimonious individuals higher. A new individual is added to this list if it's performance is higher than a certain threshold. If it is lower an attempt is made to add it to the pygmys list. The pygmys list uses a fitness function that includes the performance of the individual plus a weight for the length. In the pygmy list, a shorter individual is considered better.

During breeding, one parent is selected from the pymies list and the other from the civil servants list. The two lists are analogous to different genders, with the exception that unlike nature, gender is determined after fitness evaluation. The aim is to promote children which have the best attributes of the individuals in each of the two lists; high fitness and parsimony. The results of experiments performed by Ryan in which minimal sorting networks were evolved indicated that this approach was superior to both standard GP and GP extended to include a secondary feature in the fitness function. Their conclusion was that pygmy algorithm helps to maintain population diversity in the face of an elitist breeding policy and low population size.

4.4 Memory/State

Most computer programs make extensive use of storage, yet in almost all genetic programming examples, each genetic program is a function of its inputs and almost all storage requirements are dealt with by the framework used to support the genetic programs. Some of Koza's examples include limited storage which is used via side effects. This is in contrast with Cramer's work where a small number of global integers within the evolving program form the framework for input and output but are also available for storage.

Teller [Tel94] includes explicitly 20 small integer storage cells. The programs he evolves may freely use them. This problem, a mobile robot simulation problem, was chosen because it has not been solved (either by genetic or human programming) without some form of storage. He states that this allows genetic programming to tackle all cases of machine learning which are turing computable. Teller's read and write primitives have also been used by Andre [And94b] and Jannink [Jan94].

4.5 Fitness Evaluation

There are a number of issues associated with the fitness function. These issues range from the way that the fitness function is defined, to how it is evaluated. Below we look at research directed at these issues.

4.5.1 Absolute and Relative Fitness

In the majority of GP applications the fitness of an individual is an absolute measure determined by running the individual against a set of test cases of fixed difficulty. In co-evolution, the test cases themselves are evolving programs which may be members of the same population or of different populations. Since the test cases are evolving too, the difficulty of the test cases would be changing from generation to generation (hopefully getting progressively more difficult). This has the effect that the fitness value of both the test program and the tested program are mutually dependent and hence relative as opposed to absolute. This in turn means that knowledge of the optimal strategy is not required.

Angeline and Pollack [AP93] used absolute and competitive fitness to evolve players for the game of Tic Tac Toe (TTT). They found that co-eveolved players developed better strategies than players using absolute fitness measures.

Similar results were obtained by Reynolds [Rey94a] when he used competitive fitness for the evolution of vehicle steering control programs used for the game of tag. He reports that near optimal solutions were found solely from direct competition between individuals in the population.

4.5.2 Multi-Phase Fitness

Andre [And94b] used GP to evolve programs that are capable of storing a representation of their environment (map-making), and then using that representation to control the actions of a simulated robot. He used GP to effectively co-evolve two programs (each making use of ADFs) for each individual. He split the fitness evaluation of each individual into two phases hence the term multi-phasic fitness. In the first phase, the first program is run to extract and store features from the environment. Only the first program is provided with the set of functions to probe the environment. In the second phase, the second program is executed to use the stored representation to control the actions of a robot. The fitness value was awarded only for the success of this second phase. He found that using this approach, he was able to successfully evolve programs that solve simple problems by examining their environment, storing a representation of it, and then using the representation to control action.

4.5.3 Full and Partial Fitness

Genetic programming determines the fitness of an individual by running it. This makes it, compared to other genetic algorithms, particularly susceptible to badly behaving individuals. Badly behaved programs can be produced where there are recursive or iterative features, as these promote very long or even infinite loops. The presence of only one such program in the population effectively halts the whole genetic programming system. Where a program may be expected not to halt (or take a very long time) Koza [Koz92] implements ad-hoc loop limits which time out badly behaving individuals and give them poor fitness scores.

A more general mechanism to ensure programs which loop infinitely do not hang up the system has been suggested by Maxwell [Max94]. This applies an external time limit after which programs are interrupted if they are still running, thus allowing other members of the population to be run. The interrupted program is not aborted but is given a partial fitness. The partial fitness is used when selecting which programs to breed from and which are to be replaced by the new programs created. Programs which remain in the population are allowed to run for another time interval. Looping programs are given low partial fitness and so are eventually removed from the population, as new programs are bred. Maxwell claims his technique “effectively removes arbitrary limits on execution time (eg iteration limits) and yet still produces solutions in finite time” and often it will require “less effort” and produce solutions of “greater efficiency”.

4.5.4 Generality and Noise

Many examples attach no importance to the efficiency or size (parsimony) of the programs produced, the only important factor is how close the result given by the program is to being correct. Perhaps as a consequence many of the evolved programs are large (ie non-parsimonious). However in some cases, such as classification, the size of the program is important in itself, rather than as an indicator of wasteful coding. In such cases the size of the program gives an indication of lack of generality or overfitting training data, section 4.1.3 presents one approach to this problem.

Koza [Koz92] and Kinnear [Kin93] have combined program size with how well it works to yield a composite fitness. In Koza's example a functionally correct program was evolved of near theoretically minimum size. However more effort, that is more fitness evaluations, were required to evolve the correct program than when program size was ignored. Kinnear found he could evolve more general solutions by adding a term inversely proportional to program length to its fitness.

Reynolds [Rey94b] introduced noise into fitness tests used to evolve programs to steer a simulated 2D vehicle through connected corridors at various angles. He concluded that the addition of noise into fitness tests discouraged solutions that were brittle, opportunistic, or overly complicated, so that the discovered solutions were simple and robust.

4.5.5 Evaluation Efficiency

Individuals in a GP population are usually represented as a forest of trees. Handley [Han94a] represents the entire population of parse trees as a single directed acyclic graph (DAG). This approach brings immediate benefits in term of memory requirements since identical subtrees do not need to be duplicated. In addition if the value computed for each subtree is cached, it does not need to be re-evaluated for each each instantiation, even if it occurs in a different generation. The important restrictions in this approach are that all functions must have no side effects and the fitness cases must be bounded and fixed for all individuals and all generations. With this approach Handley reports a 15-28 fold reduction in the number of nodes per generation and an 11-20 fold reduction in the number of nodes evaluated per run.

4.5.6 Pareto Scoring

Most genetic programming work uses a single fitness criterion namely the functionality of the program, ignoring issues such as size and run time. There has been some interest in using Pareto scoring [Gol89], [Sin94a], [Lan95], which allows members of the population to be scored or compared using multiple criterion, such as functionality and runtime. Scoring systems which combine several criteria (such as functionality and efficiency) to yield a single fitness score have been tried with some success.

4.6 Genetic Operators

A number of alternatives to the standard GP crossover and mutation genetic operators have been proposed. Some of these operators [Kin94b] are introduced below.

4.6.1 Hoist

The hoist operator creates a new individual entirely from a randomly chosen subtree of an existing individual. The operator is useful for promoting parsimony.

4.6.2 Create

The create operator is unique in that it does not require any existing individuals. It generates an entirely new individual in the same way that an individual in the initial population is generated. This operator is similar to Hoist in that helps to reduce the size of program trees.

4.6.3 Self-Crossover

The self-crossover operator uses a single individual to represent both parents. The single individual is itself can be selected using the standard fitness proportional selection or tournament selection methods. Kinnear reports that no performance advantage was obtained using this operator.

4.6.4 Modular/Cassette-Crossover

One of the restrictions of the standard crossover operator is that is not possible to swap blocks that occur in the middle of a tree path. The standard crossover operator only allows entire subtrees to be swapped. One of the reasons for the success of ADFs might be due to an effective relaxation of this restriction. This possibility lead Kinnear and Altenberg to develop the Modular or Cassette crossover operator. One can best view the operator as a module swap between two individuals. A module is defined in the program tree of the first individual, and another in the program tree of the second individual. The module in the first individual is then replaced by the one in the second individual. The new module is then expanded. Clearly there is problem created by the possible mismatch of the arguments passed to the module (actual parameters) and the formal parameters defined by the module. There are two such possibilities, either the number of formal parameters are greater than the number of actual parameters or vice-versa. The former is resolved by extending the existing actual parameters with random chosen copies of themselves. The latter is resolved by choosing at random a subset of the existing actual parameters. Kinnear reports performance gains when any form of modular crossover is used with the Basic (non-ADF) GP approach, even with modular self-crossover.

4.6.5 Context Preserving Crossover (SCPC/WCPC)

D'Haeseleer [D'h94] also inspired by the success of ADFs has suggested alternative genetic operators called Strong Context Preserving Crossover (SCPC) and Weak Context Preserving Crossover (WCPC). Both of these operators require a system of coordinates for identifying node positions in a tree. SCPC allows two subtrees to be exchanged during crossover between two parents only if the points of crossover have the same coordinates. WCPC relaxes this rule slightly by allowing crossover of any subtree of the equivalent node in the other parent. Experiments were performed by D'Haeseleer using these operators on four different problems taken from [Koz92]; the obstacle avoiding robot, the boolean 11-multiplexer, the obstacle avoiding robot (iterated form) and the central place food foraging problem. The results of experiments using these operators suggests that when SCPC is combined with regular crossover, the results are for most cases superior to using just regular crossover. WCPC however did not show any benefits.

4.6.6 Constant Perturbation

Spencer [Spe94] describes a genetic operator that he calls the Constant Perturbation operator. This operator only affects special terminals called ephemeral random constants which are used to introduce constants into a GP individual [Koz92]. The constant perturbation operator as used by Handley multiplies all constants in a GP individual by a random number between 0.9 and 1.1 (i.e. perturbs the constants by up to 10% of their original value). Handley reports two positive effects of the operator for the experiments that he performed. Firstly it allowed "fine-tuning" of the coefficients in programs which is otherwise difficult to achieve. Without constant perturbation, the only way of modifying the constant is to replace it via mutation or crossover, which tends have a more coarse effect. Secondly, constant perturbation enriches the set of terminals by adding new constants. Handley believes this operator to be an improvement over the ephemeral random constants constants used by Koza. Some of the experiments that he performed with symbolic regression showed dramatic improvements.

4.7 GP Theory

One of the most important deficiency in GP at the moment is the lack of sound theoretical foundations. Some work has however commenced to address this need.

4.7.1 Fitness Landscapes

In the traditional GA individuals in a population are represented as fixed length bitstrings. The total number of possible individuals is equal to the total number permutations of the bitstring, or 2^n where n is the of bits in the bitstring. Suppose that we plot these individuals as points on a horizontal plane such that the distant between individuals is equal to the number of bit changes required to get from the one individual to the other (the hamming distance). Suppose then that we give these points a vertical component proportional to the fitness values of the individuals that they represent. What we end up with is called a fitness landscape, where the peaks represent location of high fitness, and the valleys locations of poor fitness. The actual ruggedness of this landscape would depend on the difficulty of the fitness function. A particularly difficult function might result in a fitness landscape which is flat and virtually featureless, with the exception of a few widely distributed spires (figure 16). Conversely a very easy fitness function might consist of a "mountain" with a single peak (figure 17).

When we generate an initial random population of individuals, we are effectively placing a mesh over the fitness landscape, simultaneously sampling the fitness function at different places. Genetic operators such as mutation and crossover then provide a means for exploring this landscape. Single bit mutation allows us to move one point at a time across the landscape. Crossover and multi-bit mutation allow us to take giant leaps across the landscape. An assumption of the GA is that if we select individuals of higher fitness, then by applying the genetic operators, we should end up on the landscape at a point of high fitness. Thus there should be some correlation between parents and children, This notion is called auto-correlation.

The fitness landscape provides a powerful tool for visualising the difficulty of a problem. However, if we try to use this tool in GP, we run into one major difficulty; how to decide

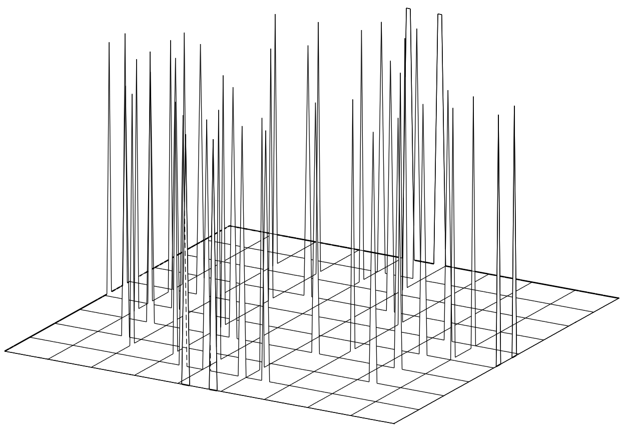

Figure 16: A difficult to search fitness landscape.

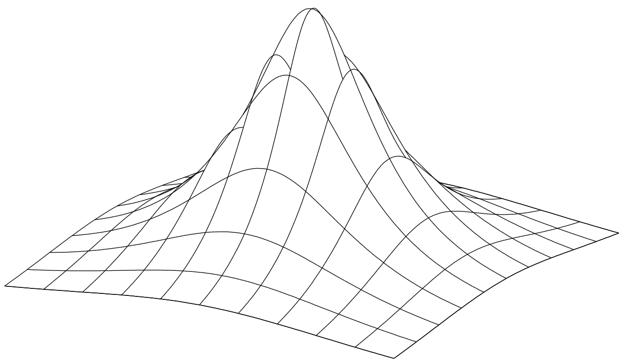

Figure 17: An easy to search fitness landscape.

which points are adjacent to each other. In the fixed-length standard GA we use the hamming distance, but this is not possible in the GP representation which is neither bit oriented nor is it of fixed length. The individuals in GP are program trees which must be compared in some way to determine how far apart they are to be plotted on the fitness landscape. The latter is not very obvious. Kinnear [Kin94c] examines the structure of the fitness landscape on which GP operates, and analyses the landscapes of a range of problems of known difficulty in order to determine the correlation between landscape measures and problem difficulty. The landscape analysis techniques that he employs include adaptive walks, and the autocorrelation of the fitness values of random walks. His results indicate that former shows better correlation with the problem difficulty than the latter for GP.

4.7.2 GP Schema Theorem and Building Block Hypothesis

O'Reilly and Oppacher [OO92] extend Koza's schema theorem for GP. The particular weakness that they identify with Koza's schema theorem is that no schema is defined by an incompletely specified S-Expression such as (+ # 2) where '#' is a wild card. They go on to form a more

Name	Author	Language		Notes
		Implementation	Evolved	
	Koza	Lisp	Lisp	Code from [Koz92] & [Koz94a], widely copied
SGPC	Tackett and Carmi	C	own	Can import/export Lisp
GPSRegress	Nguyen	Mathematica	Mathematica	Package to do symbolic regression
CEREBRUM	Dudey	Lisp	ANN (in Lisp)	Framework for genetic programming of neural networks
gpc++-0.4	Fraser	C++	own	
gepetto	Glowacki	C	own	Similar to SGPC but with enhancements
GPQUICK	Singleton	C++	own	Described in [Sin94b]
GPEIST	White	Smalltalk 80		

Table 3: Some Public Domain Genetic Programming Implementations

general schema theorem that encorporates schemas of the form mentioned. They then use this to attempt to formulate a building block hypothesis (BBH) for GP, but conclude that "the GP BBH is not forthcoming without untenable assumptions". In their analysis, they are also critical of the GA BBH.

4.7.3 Evolvability

Altenberg [Alt94] introduces evolvability as a performance measure for genetic algorithms. He defines evolvability in this context as "the ability of the genetic operator/representation scheme to produce offspring that are fitter than their parents". He describes evolvability as a local or fine grained measure of a genetic algorithms performance compared to the production of fitter offspring during one or more runs of a GA which is a more global or large scale performance measure.

His theoretical analysis of the dynamics of GP predicts the existence of an emergent selection phenomenon that he calls the evolution of evolvability. He explains that this phenomenon is cause by the proliferation within programs of blocks of code that have a higher a chance of increasing fitness when added to programs. He proposes new selection techniques and genetic operators (some of which have been introduced in previous sections) to give better control over the evolution of evolvability and and improve evolutionary performance.

4.8 GP Development Systems

Almost all of the genetic programming experiments described in this paper have used implementations either written or adapted by the experimenter. Table 3 lists the general purpose genetic programming implementations that have been placed in the public domain, Table 4 list others that have been described.

Name	Author	Language		Notes
		Implementation	Evolved	
STGP	Montana	C++	Ada/Lisp	Section 4.1.2
STROGANOFF	Iba, de Garis, Sato		Decision Trees GMDH	Section 4.1.3
GLiB	Angeline			Section 4.2.2
SSGP	Craig Reynolds			Steady State GP Section 4.3.4
GP-GIM	Andrew Singleton	C++		486 distributed network desk top supercomputing

Table 4: Some other Genetic Programming Implementations

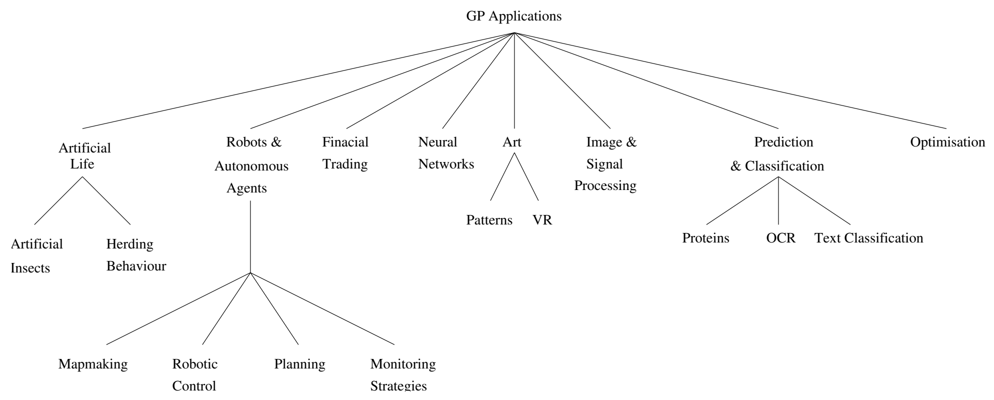

Figure 18: A Taxonomy of GP Applications

5 GP Applications

This section briefly lists real applications where genetic programming has been tried. Although these are real world applications the claims for genetic programming should not be over stated, ie the success of genetic programming should not be taken as implying there are no better techniques available for the particular application. Figure 18 shows a taxonomy of the GP applications reported below.

5.1 Prediction and Classification

Although developed before the term genetic programming was coined, BEAGLE [For81] may be classified as a genetic programming system in that it evolves tree structured programs. BEAGLE is a rule-finder program that uses a database of case histories to guide the evolution of a set of decision rules (programs) for classifying those examples. Once found the rule base (knowledge base) can be used to classify new examples. BEAGLE is commercially available and has been widely applied; eg in insurance, weather forecasting, finance and forensic science [JRFT94].

Handley [Han93] uses genetic programming to predict the shape of proteins. He was able to evolve programs which, using the protein's chemical composition, were able to predict whether each part of a protein would have a particular geometric shape (an α-helix) or not. Genetic programming was able to do this broadly as well as other techniques but all suffered from the fact that the structure depends upon more than local composition. Koza [Koz94a] reports similar success on other protein geometry problems.

Iba etal [IKdS93] use genetic programming to fit chaotic time series data, see section 4.1.3.

Masand [Mas94] applied genetic programming to find a program that gives confidence values to automatically classified news stories. Automatic text classification using multiple keywords results in multiple scores (one score for each keyword). The problem is to decide how to combine the scores to obtain an overall confidence value for the classification. This confidence value is used to decide whether or not the story needs to be referred to a human editor. On a defined set of test news stories, the best genetically produced program was better than the best human coded program.

Perry [Per94b] used GP to predict winning horses using horse racing past performance data. He divided the data into training and test sets, using only the former to evolve the prediction program, and the latter to check the generality of the best evolved program. The results of the best program showed profit levels of 181.2% during the training phase and 111.1% during the test phase.

Andre has successfully used genetic programming in optical character recognition problems (OCR). In [And94a] he combines genetic programming with a two dimensional genetic algorithm to produce an OCR program from scratch. In [And94c] he shows genetic programming can be used to maintain existing hand coded programs. He shows genetic programming automatically extending an existing manually written OCR program so that it can be used with an additional font.

5.2 Image and Signal Processing

Tackett describes [Tac93] the use of genetic programming to extract targets from low contrast noisy pictures. Various (≈20) standard metrics are abstracted from the image using standard techniques, which are then processed by a genetic program to yield target details. In his experiments, Tackett compared GP with a neural network and a binary tree classifier system. He reports better results from GP than both of the other techniques. Furthermore, the ability to analyse the evolved program enabled him to get insight into the features that assist in pattern discrimination, which would not have been possible with the "more opaque methods of neural learning."

Oakley describes [Oak94] obtaining blood flow rates within human toes using laser Doppler measurement. These measurements are both noisy and chaotic. He compares the effectiveness (at removing noise but preserving the underlying signal) of special filters evolved by genetic programming and various standard filters. He concludes that a combination of genetic programming and heuristics is the most effective.

Genetic programming has been used [HBM94] to construct a program which detects engine misfires in cars. A GP approach was found to be superior to one based on neural networks. A further advantage claimed for genetic programming in this application is that the program produced was more easily integrated into the existing (software based) engine management system than would have been the case with other approaches, such as the neural network. This may be the case in many embedded systems.

5.3 Optimisation

Nguyen and Huang [NH94] have used genetic programming to evolve 3-D jet aircraft models, however the determination of which models are fitter is done manually. This work is similar to that in section 5.8; however the aircraft models are more complex.

5.4 Financial Trading

Andrews and Prager [AP94] used genetic programming to create strategies which have traded in simulated commodity and futures markets (the double auction tournaments held by the Santa-Fe Institute, Arizona, USA). Their automatically evolved strategies have proved superior to many hand-coded strategies. A number of commercial firms are also active in this area.

5.5 Robots and Autonomous Agents

Andre [And94b] used GP to evolve programs that are capable of storing a representation of their environment (map-making), and then using that representation to control a simulated robot in that environment. In his experiments, he was able to evolve solutions yielding 100% fitness.

Handley [Han94b] used GP to generate plans for a simulated mobile robot. The plans were represented as programs, and the success of the plans were evaluated by executing them. Handley describes the genetic planner as different from traditional methods in that it does not require a logical model of the agent's world, nor does it depend on any reasoning about the effect of any generated plans on that world.

The process of monitoring the environment can be expensive for an agent, but so too can be the cost of working with inaccurate or out of date information. An agent therefore needs to establish strategies that make tradeoffs between the two costs. Atkin and Cohen [AC94] first used a GA to evolve monitoring strategies and spurred by their success applied GP to solve problems where monitoring as well as behaviour is required. Their work is probably the first example of applying GP to a real-time application.

5.6 Artificial Life

Artificial Life is the study of natural life by computer simulation. There is a ready connection to autonomous robots. For example, evolving a program to control a robot's leg may be considered either as an engineering problem or as a simulation of a real insect's leg.

Spencer [Spe93] used GP to generate programs that enable a "six-legged insect" to walk in a simulated environment. He reports that in every experiment, GP evolved programs capable of efficient walking and furthermore that the performance of these programs were at least as good and often better than that of hand-generated programs.

One aspect of Artificial Life is the study of natural evolution using computer simulation. Early simulations [Ray91] relied heavily upon mutation but genetic algorithms (eg [Sim94] and genetic programming using crossover have also been used more recently. For example Reynolds [Rey92] describes a simulation of herding behaviour based upon genetic programming[12].

[12][Rey93] and [Rey94c] gives more recent results.

5.7 Neural Networks

The normal use of the term genetic programming implies the data structures being evolved can be run as computer programs. The term has also been applied to using a genetic algorithm to evolve data structures which must be translated into another form before they can be executed. Both linear and tree (or hierarchical) data structures have been used. This process can be likened to an individual (the executable program) growing according to the instructions held in its DNA (the genetic algorithm's data structures).

The final form is often a neural network. Many ways of representing a network using a genetic algorithm have been used [PU93] [Rom93] [ZM93] [HHC93]. Kodjabachian and Meyer [KM94b] summarise various approaches to evolving artificial nervous systems.

One such is cellular encoding; Gruau [Gru93] has used cellular encoding, in conjunction with a hierarchical genetic algorithm, to produce complex neural networks with a high degree of modularity.

5.8 Artistic

There have been a number of uses of genetic programming, perhaps inspired by Dawkins' morphs [Daw86] or Karl Sims' panspermia, which generate patterns on a computer display. Singleton, with his Doodle Garden, has taken this as far as a commercial product, where the patterns grown can be used as pleasing screen savers.

Das etal [DFP^{+}94] uses genetic programming to generate sounds and three dimensional shapes. Virtual reality techniques are used to present these to the user. As in section 5.3, there is a manual fitness function, with the user indicating a preference between the four objects presented to him.

6 Conclusions

This paper has briefly surveyed recent work on the technique of automatic program generation known as genetic programming. It has presented program generation as the task of searching the space of possible programs for one that is suitable. This search space is vast and poorly behaved, which is the sort of search for which genetic algorithms are best suited. It is therefore reasonable to apply genetic algorithms to this search and, as this paper shows, this has had a measure of success.

Genetic programming has been demonstrated in the arena of classification (section 5.1), albeit not under the name genetic programming; with at least one commercial package available. It is as a general technique that genetic programming is a particularly new and emerging research area. It has solved a number of problems from a wide range of different areas. Genetic programming has also been successfully applied to real world applications, such as optical character recognition (OCR) and signal processing (section 5.2).

It is expected that use of genetic programming in the production of commercial software will become more widespread, perhaps first in signal processing and control (section 5.2). Current research directed at refining GP and making it more scalable (such as parallel implementations and abstraction) should greatly improve the range of applications.

7 Acknowledgements

This survey was written in University College, London. A. Qureshi is funded by the Engineering and Physical Sciences Research Council (EPSRC), W. B. Langdon is funded both by the EPSRC and National Grid Plc.

We would like to thank Tom Westerdale, Howard Oakley, Suran Goonatilake, Philip Treleaven, Mark Levene and Mauro Manela for their help in reading this document and suggesting improvements.

A Resources

A.1 Books and Videos

For anyone starting out in GP, the following books are highly recommended:-

The Blind Watchmaker Both GAs and GP take their inspiration from natural evolution. This book [Daw86] provides a fascinating introduction to this subject.

Genetic Algorithms in Search, Optimisation and Machine Learning Much of the techniques used in GP originate from work in GAs. This book [Gol89] written by one of the pioneers of GAs therefore provides an excellent background to those interested GP.

Genetic Programming: On the Programming of Computers by Natural Selection This book [Koz94a] (referred to as GP I by GPers) is considered the bible for GP and provides complete introduction and tutorial. It has many worked examples.

Genetic Programming: The Movie This video [KR92] is an excellent companion to the above book. GP concepts are explained and solutions to example problems used in the book can be seen evolving.

Genetic Programming II: Automatic Discovery of Reusable Programs This sequel [Koz94a] extends the concepts introduced in the original with the notion of ADFs. Worked examples again are extensively used.

Genetic Programming: The Next Generation This video [Koz94b] does for GP II what "GP the Movie" did for GP I.

Advances in GP This book [Kin94a] contains an excellent collection of GP papers.

Advances in GP II This sequel to the above book is due to published soon and will contain a new set of papers.

A.2 Internet Resources

The **GP home page** is located at University College London (UCL) and has the URL "http://www.cs.ucl.ac.uk/research/genprog/". The home page includes a GP Frequently Asked Questions (FAQ) list with answers, and links to many other GP related resources.

The FTP site at UCL (cs.ucl.ac.uk) has a directory (/genetic) archiving GP related resources. The directory contains a complete mirror of the GP archive at "ftp.io.com" and an additional collection of papers, code and an extensive GP bibliography. A subdirectory called "incoming" allows resources to be submitted to this archive. Bill Langdon has put together simple example programs for both GAs and GP, which are available in the ga-demo and gp-code subdirectories respectively.

Adil Qureshi maintains both the GP home page and the FTP site at UCL, and can be contacted at the email address "A.Qureshi@cs.ucl.ac.uk".

There is a GP Mailing List which is used by GP researchers to exchange informations and obtain help. To join the mailing list, send a message to:-

"genetic-programming-REQUEST@cs.stanford.edu"

The body of the message should consist of a single line with the words "subscribe genetic-programming" or "subscribe genetic-programming <my-address>", depending on whether you want to be subscribed using the address you mailed the message from, or an alternative address. The FAQ at the GP home page (see above) gives further details regarding the use of the mailing list.

A References

References

[Abb91] R. J. Abbott. Niches as a GA divide-and-conquer strategy. In Art Chapman and Leonard Myers, editors, *Proceedings of the Second Annual AI Symposium for the California State University*. California State University, 1991.

[AC94] M. Atkin and P. R. Cohen. Learning monitoring strategies: A difficult genetic programming application. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*. IEEE Press, 1994.

[Alt94] Lee Altenberg. The evolution of evolvability in genetic programming. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*. MIT Press, 1994.

[And94a] David Andre. Automatically defined features: The simultaneous evolution of 2-dimensional feature detectors and an algorithm for using them. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 23. MIT Press, 1994.

[And94b] David Andre. Evolution of mapmaking ability: Strategies for the evolution of learning, planning, and memory using genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1. IEEE Press, June 1994.

[And94c] David Andre. Learning and upgrading rules for an OCR system using genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*. IEEE Press, 1994.

[Ang94] Peter John Angeline. Genetic programming and emergent intelligence. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 4. MIT Press, 1994.

[AP93] Peter J. Angeline and Jordan B. Pollack. Competitive environments evolve better solutions for complex tasks. In *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 264–270. Morgan Kaufmann, 1993.

[AP94] Martin Andrews and Richard Prager. Genetic programming for the acquisition of double auction market strategies. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 16, pages 355–368. MIT Press, 1994.

[Ban93] Wolfgang Banzhaf. Genetic programming for pedestrians. In *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*. Morgan Kaufmann, 1993.

[BHS91] Thomas Back, Frank Hoffmeister, and Hans-Paul Schwefel. A survey of evolution strategies. In *Proceedings of fourth International Conference on Genetic Algorithms*, pages 2–10, 1991.

[Cra85] Nichael Lynn Cramer. A representation for the adaptive generation of simple sequential programs. In John J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and the Applications*, pages 183–187, 1985.

[Dav91] Lawrence Davis, editor. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York, 1991.

[Daw86] Richard Dawkins. *The blind Watchmaker*. Harlow : Longman Scientific and Technical, 1986.

[DB94] Patrik D'haeseleer and Jason Bluming. Effects of locality in individual and population evolution. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 8, pages 177–198. MIT Press, 1994.

[De 87] Kenneth De Jong. On using genetic algorithms to search program spaces. In John J. Grefenstette, editor, *Genetic Algorithms and their Applications: Proceedings of the second international conference on Genetic Algorithms*, pages 210–216, George Mason University, July 1987. Lawrence Erlbaum Associates.

[DFP$^+$94] Sumit Das, Terry Franguidakis, Michael Papka, Thomas A. DeFanti, and Daniel J. Sandin. A genetic programming application in virtual reality. In *Proceedings of the first IEEE Conference on Evolutionary Computation*, volume 1, pages 480–484. IEEE Press, June 1994. Part of 1994 IEEE World Congress on Computational Intelligence, Orlando, Florida.

[D'h94] P. D'haeseleer. Context preserving crossover in genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 256–261. IEEE Press, 1994.

[FD87] Cory Fujiki and John Dickinson. Using the genetic algorithm to generate lisp source code to solve the prisoner's dilemma. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 236–240. Lawrence Erlbaum Associates, July 1987.

[Fel93] David S. Feldman. Fuzzy network synthesis with genetic algorithms. In *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 312–317. Morgan Kaufmann, 1993.

[For81] Richard Forsyth. BEAGLE A dawinian approach to pattern recognition. *Kybernetes*, 10:159–166, 1981.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimisation and Machine Learning*. Addison-Wesley, Reading, MA, 1989.

[Gru93] Frederic Gruau. Genetic synthesis of modular neural networks. In *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 318–325. Morgan Kaufmann, 1993.

[Han93] Simon Handley. Automatic learning of a detector for alpha-helices in protein sequences via genetic programming. In *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 271–278. Morgan Kaufmann, 1993.

[Han94a] S. Handley. On the use of a directed acyclic graph to represent a population of computer programs. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, pages 154–159. IEEE Press, 1994.

[Han94b] Simon G. Handley. The automatic generations of plans for a mobile robot via genetic programming with automatically defined functions. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 18. MIT Press, 1994.

[HHC93] Inman Harvey, Philip Husbands, and Dave Cliff. Genetic convergence in a species of evolved robot control architectures. In *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, page 636. Morgan Kaufmann, 1993.

[HHNT86] John H. Holland, Keith J. Holyoak, Richard E. Nisbett, and Paul R. Thagard. *Induction Processes of Inference, Learning, and Discovery.* MIT Press, 1986. Section 3.1 and Chapter 4.

[Hol73] John H Holland. Genetic algorithms and the optimal allocation of trials. *SIAM Journal on Computation*, 2:88–105, 1973.

[Hol92] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* MIT Press, 1992. First Published by University of Michigan Press 1975.

[IdS94] Hitoshi Iba, Hugo de Garis, and Taisuke Sato. Genetic programming using a minimum description length principle. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 12, pages 265–284. MIT Press, 1994.

[IKdS93] Hitoshi Iba, Takio Karita, Hugo de Garis, and Taisuke Sato. System identification using structured genetic algorithms. In *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 279–286. Morgan Kaufmann, 1993.

[Jan94] Jan Jannink. Cracking and co-evolving randomizers. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 20, pages 425–443. MIT Press, 1994.

[JRFT94] C. Alippi J. R. Filho and P. Treleaven. Genetic algorithm programming environments. *IEEE Computer Journal*, Jun 1994.

[KA95] John R. Koza and David Andre. Parallel genetic programming on a network of transputers. Technical Report CS-TR-95-1542, Stanford University, Department of Computer Science, January 1995.

[KGV83] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983. 13th.

[Kin93] Kenneth E. Kinnear Jr. Generality and difficulty in genetic programming: Evolving a sort. In *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 287–294. Morgan Kaufmann, 1993.

[Kin94a] Kenneth E. Kinnear, Jr., editor. *Advances in Genetic Programming*. MIT Press, Cambridge, MA, 1994.

[Kin94b] Kenneth E. Kinnear, Jr. Alternatives in automatic function definition: A comparison of performance. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 6. MIT Press, 1994.

[Kin94c] Kenneth E. Kinnear, Jr. Fitness landscapes and difficulty in genetic programming. In *Proceedings of the 1994 IEEE World Conference on Computational Intelligence*, volume 1, pages 142–147. IEEE Press, 1994.

[KM94a] Mike J. Keith and Martin C. Martin. Genetic programming in C++: Implementation issues. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 13. MIT Press, 1994.

[KM94b] Jerome Kodjabachian and Jean-Arcady Meyer. Development, learning and evolution in animats. In P. Gaussier and J-D Nicoud, editors, *Perceptions to Action*, pages 96–109, Lausanne Switzerland, Sep 1994. IEEE Computer Society Press.

[Koz92] John R. Koza. *Genetic Programming: On the Programming of Computers by Natural Selection*. MIT press, Cambridge, MA, 1992.

[Koz94a] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, May 1994.

[Koz94b] John R. Koza. *Genetic Programming II Videotape: The next generation*. The MIT Press, 55 Hayward Street, Cambridge, Massachusetts, release 1.01 edition, 1994.

[KR92] John R. Koza and James P. Rice. *Genetic Programming:The Movie*. The MIT Press, Cambridge, MA, 1992.

[Lan95] W. B. Langdon. Evolving data structures using genetic programming. Research Note RN/1/95, UCL, Gower Street, London, WC1E 6BT, January 1995. Accepted for presentation at ICGA-95.

[Mas94] Brij Masand. Optimising confidence of text classification by evolution of symbolic expressions. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 21. MIT Press, 1994.

[Max94] Sidney R. Maxwell III. Experiments with a coroutine model for genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence, Orlando, Florida, USA*, volume 1, pages 413–417a. IEEE Press, June 1994.

[Mon94] David J. Montana. Strongly typed genetic programming. BBN Technical Report #7866, Bolt Beranek and Newman, Inc., 10 Moulton Street, Cambridge, MA 02138, USA, March 1994.

[NH94] Thang Nguyen and Thomas Huang. Evolvable 3D modeling for model-based object recognition systems. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 22, pages 459–475. MIT Press, 1994.

[Nor94a] Peter Nordin. A compiling genetic programming system that directly manipulates the machine code. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 14. MIT Press, 1994.

[Nor94b] Peter Nordin. Two stage genetic programming using prolog. Electronic Correspondence, 1994.

[Oak94] Howard Oakley. Two scientific applications of genetic programming: Stack filters and non-linear equation fitting to chaotic data. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 17. MIT Press, 1994.

[OO92] U. M. O'Reilly and F. Oppacher. The troubling aspects of a building block hypothesis for genetic programming. Working Paper 94-02-001, Santa Fe Institute, 1992.

[OO94] Una-May O'Reilly and Franz Oppacher. Program search with a hierarchical variable length representation: Genetic programming, simulated annealing and hill climbing. Technical Report 94-04-021, Santa Fe Institute, 1994.

[Per94a] Tim Perkis. Stack-based genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, pages 148–153. IEEE Press, 1994.

[Per94b] J. E. Perry. The effect of population enrichment in genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, pages 456–461. IEEE Press, 1994.

[PU93] Daniel Polani and Thomas Uthmann. Training kohonen feature maps in different topologies: an analysis using genetic algorithms. In *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 326–333. Morgan Kaufmann, 1993.

[Ray91] Thomas S. Ray. Is it alive or is it GA. In Richard K. Belew and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 527–534, San Diego, California, July 1991. Morgan Kaufmann.

[RB94] J. P. Rosca and D. H. Ballard. Learning by adapting representations in genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*. IEEE Press, 1994.

[Rey92] Craig W. Reynolds. An evolved, vision-based behavioral model of coordinated group motion. In Meyer and Wilson, editors, *From Animals to Animats (the proceedings of Simulation of Adaptive Behaviour*. MIT Press, 1992.

[Rey93] Craig W. Reynolds. An evolved, vision-based behavioral model of obstacle avoidance behaviour. In Christopher G. Langton, editor, *Artificial Life III*, volume 16 of *SFI Studies in the Sciences of Complexity*. Addison-Wesley, 1993.

[Rey94a] Craig W. Reynolds. Competition, coevolution and the game of tag. In Rodney A. Brooks and Pattie Maes, editors, *Proceedings of the Fourth Internationa Workshop on the Synthesis and Simulation of Living Systems*, pages 59–69. MIT Press, 1994.

[Rey94b] Craig W. Reynolds. Evolution of corridor following behavior in a noisy world. In *SAB-94*, 1994.

[Rey94c] Craig W. Reynolds. Evolution of obstacle avoidance behaviour:using noise to promote robust solutions. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 10. MIT Press, 1994.

[Rom93] Steve G. Romaniak. Evolutionary growth perceptrons. In *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 334–341. Morgan Kaufmann, 1993.

[Rya94] Conor Ryan. Pygmies and civil servants. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 11. MIT Press, 1994.

[Sel92] Steven Self. On the origin of effective procedures by means of artificial selection. Master's thesis, Birkbeck College, University of London, September 1992.

[Sie94] Eric V. Siegel. Competitively evolving decision trees against fixed training cases for natural language processing. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 19. MIT Press, 1994.

[Sim94] Karl Sims. Evolving 3d morpholgy and behaviour by competition. In R. Brooks and P. Maes, editors, *Artificial Life IV Proceedings*, pages 28–39, 245 First Street, Cambridge, MA 02142, USA, 1994. MIT Press.

[Sin93] Andrew Singleton. Meta GA, desktop supercomputing and object-orientated GP. Notes from Genetic Programming Workshop at ICGA-93,, 1993.

[Sin94a] Andrew Singleton. Tournament selection. Electronic communication, 1994.

[Sin94b] Andy Singleton. Genetic programming with C++. *BYTE*, pages 171–176, February 1994.

[Spe93] Graham F. Spencer. Automatic generation of programs for crawling and walking. In *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, page 654, Stanford, 1993. Morgan Kaufmann.

[Spe94] Graham F. Spencer. Automatic generation of programs for crawling and walking. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 15, pages 335–353. MIT Press, 1994.

[Tac93] Walter Alden Tackett. Genetic programming for feature discovery and image discrimination. In *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*. Morgan Kaufmann, 1993.

[TC94a] W. A. Tackett and A. Carmi. The unique implications of brood selection for genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*. IEEE Press, 1994.

[TC94b] Walter Alden Tackett and Aviram Carmi. The donut problem: Scalability and generalization in genetic programming. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 7. MIT Press, 1994.

[Tel94] Astro Teller. The evolution of mental models. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 9. MIT Press, 1994.

[UvA89] J. W. H. M. Uiterwijk, H. J. van den Herik, and L. V. Allis. A knowledge-based approach to connect-four. In David Levy and Don Beals, editors, *Heuristic Programming in Artificial Intelligence: The First Computer Olympiad*. Ellis Harwood; John Wiley, 1989.

[ZM93] Byoung-Tak Zhang and Heinz Muhlenbein. Genetic programming of minimal neural nets using occam's razor. In Stephanie Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 342–349. Morgan Kaufmann, 1993.

B Glossary

Building Block A pattern of genes in a contiguous section of a chromosome which, if present, confers a high fitness to the individual. According to the building block hypothesis, a complete solution can be constructed by crossover joining together in a single individual many building blocks which were originally spread throughout the population.

Cellular Automata A regular array of identical finite state automata whose next state is determined solely by their current state and the state of their neighbours. The most widely seen is the game of *Life* in which complex patterns emerge from a (supposedly infinite) square lattice of simple two state automata whose next state is determined solely by the current states of its four closes neighbours and itself.

Classifiers An extension of genetic algorithms in which the population consists of a co-operating set of rules (ie a rulebase) which are to learn to solve a problem given a number of test cases. Between each generation the population as a whole is evaluated and a fitness is assigned to each rule using the bucket-brigade algorithm or other credit sharing scheme (eg the Pitt scheme). These credit sharing schemes aim to reward or punish rules which contribute to a test case according to how good the total solution is by adjusting the individual rules fitness.

At the end of the test data a new generation is created using a genetic algorithm as if each rule were independent using its own fitness (measures may be taken are taken to ensure a given rule only appears once in the new population).

Co-evolution Two or more populations are evolved at the same time. Often the separate populations compete against each other.

Convergence Tendency of members of the population to be the same. May be used to mean either their representation or behaviour are identical. Loosely a genetic algorithm solution has been reached.

Chromosome Normally, in genetic algorithms the bit string which represents the individual. In genetic programming the individual and its representation are the same, both being the program parse tree. In nature many species store their genetic information on more than one chromosome.

Crossover Creating a new individual's representation from parts of its parents' representations.

Deme A separately evolving subset of the whole population. The subsets may be evolved on different computers. Emigration between subsets may be used (see Panmixia).

Elitist An elitist genetic algorithm is one that always retains in the population the best individual found so far.

Evolution Programming A population containing a number of trial solutions each of which is evaluated to yield an error. Typically, at the end of each generation, the best half of the population is retained and a new solution is produced from each survivor. The process is continued with the aim that the population should evolve to contain an acceptable solution.

Evolutionary programming like Evolution Strategy produces new children by mutating at random from a single parent solution. The analogue components (eg the connection weights when applied to artificial neural networks) are changed by a Gaussian function whose standard deviation is given by a function of the parent's error called its temperature. Digital components (eg presence of a hidden node) are created and destroyed at random.

Evolution Strategy *or Evolutionsstrategie* A search technique first developed in Berlin. Each point in the search space is represented by a vector of real values. In the original Evolution Strategy, $(1 + 1)$-ES, the next point to search is given by adding gaussian random noise to the current search point. The new point is evaluated and if better the search continues from it. If not the search continues from the original point. The level of noise is adjusted so that on average $\frac{1}{5}$ of new points are accepted.

Evolutionary Strategies can be thought of as like an analogue version of genetic algorithms. In $(1+1)$-ES, 1 parent is used to create 1 offspring. In $(\mu+\lambda)$-ES and (μ,λ)-ES μ parents are used to create λ children (perhaps using crossover).

Finite State Automata (FSA) *or Finite State Machine (FSM)* A machine which can be totally described by a finite set of states, it being in one these at any one time, plus a set of rules which determine when it moves from one state to another.

Fitness Function A function that defines the fitness of an individual as a solution for for the required problem. In most cases the goal is to find an individual with the maximum (or minimum) fitness.

Function Set The set of operators used in a genetic program, eg $+ - \times \div$. These act as the branch points in the parse tree, linking other functions or terminals. See also non-terminals.

Generation When the children of one population replace their parents in that population. Where some part of the original population is retained, as in steady state GAs, generation typically refers to the interval during which the number of new individuals created is equal to the population size.

Genetic Algorithm A population containing a number of trial solutions each of which is evaluated (to yield a fitness) and a new generation is created from the better of them. The process is continued through a number of generations with the aim that the population should evolve to contain an acceptable solution.

GAs are characterised by representing the solution as an (often fixed length) string of digital symbols, selecting parents from the current population in proportion to their fitness (or some approximation of this) and the use of crossover as the dominate means of creating new members of the population. The initial population may be created at random or from some known starting point.

Genetic Operator An operator in a genetic algorithm or genetic programming, which acts upon the chromosome to produce a new individual. Examples are mutation and crossover.

Genetic Program A program produced by genetic programming.

Genetic Programming A subset of genetic algorithms. The members of the population are the parse trees of computer programs whose fitness is evaluated by running them. The reproduction operators (eg crossover) are refined to ensure that the child is syntactically correct (some protection may be given against semantic errors too). This is achieved by acting upon subtrees.

Genetic programming is most easily implemented where the computer language is tree structured so there is no need to explicitly evaluated its parse tree. This is one of the reasons why Lisp is often used for genetic programming.

This is the common usage of the term *genetic programming*, however the term has also been used to refer to the programming of cellular automata and neural networks using a genetic algorithm.

Hits The number of hits an individual scores is the number of test cases for which it returns the correct answer (or close enough to it). This may or may not be a component of the fitness function. When an individual gains the maximum number of hits this may terminate the run.

Infix Notation Notation in which the operator appears between the operands. Eg $(a + b) \times c$. Infix notation requires the use of brackets to specify the order of evaluation, unlike either prefix or postfix notations.

Non-Terminal Functions used to link parse tree together. This name may be used to avoid confusion with functions with no parameters which can only act as end points of the parse tree (ie leafs) and so are part of the terminal set.

Mutation Arbitrary change to representation, often at random. In genetic programming, a subtree is replaced by another, some or all of which is created at random.

Panmixia When a population is split into a number of separately evolving populations (demes) but the level of emigration is sufficiently high that they continue to evolve as if a single population.

Parsimony Brevity. In GP, this is measured by counting the nodes in the tree. The smaller the program, the smaller the tree, the lower the count and the more parsimonious it is.

Postfix Notation *Reverse Polish Notation or Suffix Notation* Notation in which the operator follows its operands. Eg $a + b \times c$ represented as $abc \times +$.

Prefix Notation *Polish Notation* Notation in which the operator comes before its operands. Eg $a + b$ represented as $+ab$.

Premature Convergence When a genetic algorithm's population converges to something which is not the solution you wanted.

Recombination as crossover.

Reproduction Production of new member of population from existing members. May be used to mean an exact copy of the original member.

Simulated Annealing Search technique where a single trial solution is modified at random. An *energy* is defined which represents how good the solution is. The goal is to find the best solution by minimising the energy. Changes which lead to a lower energy are always accepted; an increase is probabilistically accepted. The probability is given by $\exp(-\Delta E/k_B T)$. Where ΔE is the change in energy, k_B is a constant and T is the *Temperature*. Initially the temperature is high corresponding to a liquid or molten state where large changes are possible and it is progressively reduced using a *cooling schedule* so allowing smaller changes until the system *solidifies* at a low energy solution.

Stochastic Random or probabilistic but with some direction. For example the arrival of people at a post office might be random but average properties (such as the queue length) can be predicted.

Terminal Set A set from which all end (leaf) nodes in the parse trees representing the programs must be drawn. A terminal might be a variable, a constant or a function with no arguments.

Tournament Selection A mechanism for choosing individuals from a population. A group (typically between 2 and 7 individuals) are selected at random from the population and the best (normally only one, but possibly more) is chosen.