Genetic Programming in Mathematica

by Hussein Suleman

submitted in fulfilment of the requirements for the degree of Magister Scientiae in the Department of Computer Science in the Faculty of Science at the University of Durban-Westville.

Supervisor : Dr. M. Hajek

Date Submitted : 15 January 1997

Declaration

I, Hussein Suleman, Reg. No. : 9144784,

hereby declare that the thesis entitled

Genetic Programming in Mathematica

is the result of my own investigation and research and that it has not been submitted in part or in full for any other degree or to any other University.

………………………. ……………………

Signature Date

ACKNOWLEDGEMENTS

My heartfelt thanks go to my supervisor, Dr M. Hajek, for his ever-willing assistance throughout my studies, the staff of the Department of Computer Science, my family and friends for supporting all my endeavours, and God, without whose guidance none of this would be possible.

CONTENTS

ACKNOWLEDGEMENTS II

CONTENTS III

LIST OF FIGURES VII

LIST OF TABLES IX

ABSTRACT 1

CHAPTER 1 : INTRODUCTION 2

THE EVOLUTIONARY PARADIGM OF PROGRAMMING 2

GENETIC ALGORITHMS 6

Representation of Problem 6

Population of Solutions 8

Fitness 9

Reproduction 11

Crossover 13

Mutation 14

General Algorithm 15

EVOLUTIONARY PROGRAMMING AND EVOLUTION STRATEGIES 16

GENETIC PROGRAMMING 17

Representation 17

Population of Solutions 20

Fitness 21

Reproduction 21

Crossover 22

Mutation 23

General Algorithm 24

Applications of GP 25

CHAPTER 2 : A MATHEMATICA IMPLEMENTATION 28

IMPLEMENTATION LANGUAGES 28

Lisp 28

C++ 29

Mathematica 30

INTRODUCTION TO MATHEMATICA 31

Platforms and Organisation ... *31*
Variables ... *32*
Functions ... *33*
Paradigms ... *35*
Modularization - Functions ... *36*
Modularization - Files ... *37*
SIMPLE GENETIC PROGRAMMING IMPLEMENTATION ... 38
Representation of Data ... *38*
Closure of Function Set ... *40*
Fitness ... *41*
Parameters ... *42*
Generation of Random Population ... *43*
Reproduction ... *44*
Crossover ... *46*
Mutation ... *48*
Result Designation ... *48*
Initialisation ... *49*
ApplyGen ... *50*
Automatic Recovery ... *52*
Mechanics of a Sample Implementation ... *53*

CHAPTER 3 : SYMBOLIC REGRESSION ... 54

STATISTICAL ANALYSIS TECHNIQUES ... 54
EXPERIMENT 1: SYMBOLIC REGRESSION IN MATHEMATICA ... 57
Problem Selection ... *57*
Test Data ... *58*
Platform ... *59*
Statistics ... *60*
Problem Representation and Parameters ... *60*
Experiment 1.1 ... *62*
Experiment 1.2 ... *64*
Experiment 1.3 ... *66*
Experiment 1.4 ... *68*
Experiment 1.5 ... *70*
Experiment 1.6 ... *73*
Experiment 1.7 ... *74*
Experiment 1.8 ... *75*
Conclusion ... *76*

CHAPTER 4 : PARALLEL GENETIC PROGRAMMING ... 78

INTRODUCTION 78
Suitability of Parallel Processing for GP *78*
Parallel Processing Methodologies *79*
Some Existing Implementations *81*
PARALLEL PROCESSING MODEL 81
Sub-populations and Migration *81*
General Parallel Algorithm *83*
Data Storage *84*
Approaches to Job Control *84*
Scheduling *88*
MATHEMATICA IMPLEMENTATION 90
SEQUENCE OF FUNCTION CALLS 96

CHAPTER 5 : APPLICATIONS OF PARALLEL GP 97

STATISTICAL ANALYSIS TECHNIQUES 97
EXPERIMENT 2: PARALLEL SYMBOLIC REGRESSION 104
Test Data *104*
Experiment 2.1 .... *105*
Experiment 2.2 .... *108*
Experiment 2.3 .... *110*
Conclusion *110*
EXPERIMENT 3: CSTR CONTROLLER 111
Experiment 3.1 .... *112*
Experiment 3.2 .... *114*
Conclusion *114*
EXPERIMENT 4: PID CONTROLLER 115
Experiment 4.1 .... *117*
Experiment 4.2 .... *118*
Experiment 4.3 .... *121*
Conclusion *121*
EXPERIMENT 5: THE MAGIC STAR 121
Discussion *121*
Conclusion *125*

CONCLUSION 126

FUTURE DIRECTIONS 126

APPENDIX A : SERIAL ALGORITHM 128

XTRADEFS.M 128
TIME.M 128

GENPROG.M 128
STATS.M 133
HIST.M 133
RESTART.M 134

APPENDIX B : SCHEDULER 135

APPENDIX C : PARALLEL GP 146

TIME.M 146
XTRADEFS.M 146
DEFAULT.M 147
OPERATOR.M 148
INITIAL.M 150
GENMAIN.M 157
STATS.M 162

BIBLIOGRAPHY 166

LIST OF FIGURES

Figure 1.1 Bit-string GA representation 7

Figure 1.2. Conversion from bit-string to real representation 8

Figure 1.3. Initial random population 9

Figure 1.4. Selected individuals with corresponding real values and fitnesses 11

Figure 1.5. Roulette wheel individual selection 12

Figure 1.6. Crossover of two individuals in GA 13

Figure 1.7. Mutation of an individual in GA 15

Figure 1.8. Representation of individuals as trees in GP 18

Figure 1.9. Extract from population of GP trees and corresponding expression representation 20

Figure 1.10. Crossover of two individuals in GP 23

Figure 1.11. Mutation of an individual in GP 24

Figure 2.1. Representation of an expression 38

Figure 3.1. Best and worst fitnesses per generation 55

Figure 3.2. Fitness histogram for generation 8 56

Figure 3.3. Fitting of solution to sample points - Exp 1.1 63

Figure 3.4. Maximum/minimum fitness curve - Exp 1.1 64

Figure 3.5. Fitting of solution to sample points - Exp 1.2 65

Figure 3.6. Maximum/minimum fitness curve - Exp 1.2 66

Figure 3.7. Fitting of solution to sample points - Exp 1.3 67

Figure 3.8. Maximum/minimum fitness values - Exp 1.3 68

Figure 3.9. Fitting of solution to sample points - Exp 1.4 Run 1 70

Figure 3.10. Fitness histograms 72

Figure 3.11. Sample data and their fitted equations for noisy data 76

Figure 4.1. Rectangular spatial distribution of sub-populations showing migration possibilities for sub-population 13 82

Figure 4.2. Screen snapshot of scheduler 88

Figure 4.3. Matrix of migration possibilities 89

Figure 5.1. Output from GlobalCurve, displaying maximum, minimum and average fitnesses of generations 98

Figure 5.2. Typical maximum fitness histogram 100

Figure 5.3. Graph showing overall time taken vs. no of processors - Exp 2.1 107

Figure 5.4. Graph showing time taken per generation vs. no of processors - Exp 2.1 108

Figure 5.5. Control path for CSTR functions obtained by GP - Exp 3.1 113

Figure 5.6. Control path for CSTR functions obtained by GP - Exp 3.2 114

Figure 5.7. Desired control trajectory of PID controller 115

Figure 5.8. Control path for PID controller functions obtained by GP - Exp 4.1 117
Figure 5.9. Global fitness curve for PID controller - Exp 4.1 118
Figure 5.10. Control path for PID controller functions obtained by GP - Exp 4.2 Run 1 119
Figure 5.11. Global fitness curve for PID controller - Exp 4.2 Run 2 120
Figure 5.12. Six-point magic star configuration...122

LIST OF TABLES

TABLE 2.1. SAMPLE LISP EXPRESSIONS....................................28
TABLE 3.1. 21 PAIRS OF X-Y COORDINATES USED AS TEST DATA IN EXPERIMENTS 1.4-1.7....................58
TABLE 3.2. GP PARAMETERS FOR SYMBOLIC REGRESSION - EXP 1.1-1.3....................61
TABLE 3.3. GP PARAMETERS FOR SYMBOLIC REGRESSION - EXP 1.4....................69
TABLE 3.4. TIME TAKEN FOR GP RUNS - EXP 1.4....................70
TABLE 3.5. GP PARAMETERS FOR SYMBOLIC REGRESSION - EXP 1.5....................71
TABLE 3.6. GP PARAMETERS FOR SYMBOLIC REGRESSION - EXP 1.6....................73
TABLE 3.7. MAXIMUM FITNESSES AND TIMES TAKEN - EXP 1.6....................73
TABLE 3.8. GP PARAMETERS FOR SYMBOLIC REGRESSION - EXP 1.7....................74
TABLE 3.9. TIME TAKEN FOR RUNS - EXP 1.7....................74
TABLE 5.1. SAMPLE POINTS - EXP 2....................104
TABLE 5.2. PARAMETERS FOR PARALLEL SYMBOLIC REGRESSION - EXP 2.1....................105
TABLE 5.3. TIME TAKEN TO RUN PARALLEL SYMBOLIC REGRESSION ON MULTIPLE PROCESSORS....................106
TABLE 5.4. PARAMETERS FOR PARALLEL SYMBOLIC REGRESSION - EXP 2.2....................109
TABLE 5.5. TIME TAKEN TO RUN SINGLE-POPULATION SYMBOLIC REGRESSION ON SINGLE PROCESSOR..109
TABLE 5.6. TIME TAKEN TO RUN PARALLEL SYMBOLIC REGRESSION ON 3 PROCESSORS WITH PARALLELISED MIGRATION OPERATION....................110
TABLE 5.7. GP PARAMETERS FOR CSTR....................112
TABLE 5.8. GP PARAMETERS FOR PID CONTROLLER....................116
TABLE 5.9. CRITERIA FOR PID CONTROLLERS....................120
TABLE 5.10. GP PARAMETERS FOR MAGIC STAR....................124

ABSTRACT

Genetic Programming (GP) is an implementation of evolutionary programming, where the problem-solving domain is modelled on computer and the algorithm attempts to find a solution by the process of simulated evolution, employing the biological theory of genetics and the Darwinian principle of survival of the fittest. GP is distinct from other techniques because of its tree representation and manipulation of all solutions.

GP has traditionally been implemented in LISP but there is a slow migration towards faster languages like C++. Any implementation language is dictated not only by the speed of the platform but also by the desirability of such an implementation. With a large number of scientists migrating to scientifically-biased programming languages like Mathematica, such provides an ideal testbed for GP.

In this study it was attempted to implement GP on a Mathematica platform, exploiting the advantages of Mathematica's unique capabilities. Wherever possible, optimizations have been applied to drive the GP algorithm towards realistic goals. At an early stage it was noted that the standard GP algorithm could be significantly speeded up by parallelisation and the distribution of processing. This was incorporated into the algorithm, using known techniques and Mathematica-specific knowledge.

Benchmark problems were tested on both the serial and parallel algorithms to assess the ability of the implementation to effectively solve problems using GP. Mostly known problems were used since it was desired to test the implementation and not the capabilities of the algorithm itself.

Mathematica has been found to be suitable for the implementation of GP in cases where the problem domain has been modelled already in this environment. Although Mathematica is not an optimal environment for the execution of a GP, it is highly adaptable to different problem domains, thus promoting the implementation of problem-solving techniques like GP.

CHAPTER 1 : INTRODUCTION

The Evolutionary Paradigm of Programming

Computer Science had its beginnings when scientists built the first computers and realised that these machines needed to be constantly tended. This tending took the form of writing programs and thereafter maintaining these programs and their data. At first it was a rather haphazard process, with programmers writing code on the spur of the moment and then changing their programs to suit changes in the environment or the requirements. As time passed, this disorderly process caused more problems than solutions and Computer Science began to turn its head towards the formal specification of programming.

The programming of computers can be considered as the focus of research in Computer Science. In recent years, people have been asking very pertinent questions regarding the speed and size of programs. There has been a quest to write programs that run faster and use less memory and storage. Also, some programs are sought simply for parsimony or the ability to prove correctness mathematically. But, like any other scientific field, the thrust of work is not on efficiency but on new developments. Problems from all aspects of life are modelled on computer and new solutions are being constantly sought.

People from varied disciplines implement their problem-solving methodologies on computer. In many cases an existing sequence of steps is known and this simply needs to be converted into a computer program. In other situations, only raw data is available and this then needs to be processed to generate useful information. Both scenarios require that computer programs be written, whether by the user or an external party.

Programming, by its very creative nature, is an intuitive process that cannot be broken down into finite determinate steps. Many people argue for and against this standpoint. Software engineers argue quite strongly that software can be created using a pre-defined series of steps in a determinate manner [Schach, 1992]. But they also agree

that innovations in programming cannot follow this same process. Ultimately, a program has to be written and that program cannot always be created in a definite manner. This implies that a programmer will have to intuitively devise a new algorithm, using and incorporating existing algorithms. Being a creative process, it takes an unknown amount of time and resources to accomplish. Also, the programmer never knows for certain whether the problem will be solved (except for some cases where this is proven mathematically in advance) by the program. Some problems do not even lend themselves to a program, although most of these are ferreted out by the experienced programmer.

Whatever the case may be, an experienced programmer has to devote an unknown amount of time in order to solve any moderately complex problem. This in itself is a problem worthy of study. How can this programming task be made easier ? Classical computer science has proposed many techniques to ease programming by modularising the data and programs e.g. object-orientation. Artificial intelligence suggests different approaches which consider computer programs as simply "black boxes" which convert input into the appropriate output.

Neural networks are a popular strategy for problem solving nowadays. Using this approach, a computer model of the human brain is created and this then learns the relationship between the input and output. Information is stored internally in the form of a matrix of weights, where each weight refers to the relative ability of one neuron to fire another one. This "connectionist" approach is used widely because of its ability to simulate the learning and recollection process of human thought. However, it does have some disadvantages, namely the requirement that the inter-neuron connections be seeded before learning can begin (in back-propagation learning). This initial state has to be determined experimentally and this makes it somewhat similar to the classical program because an expert needs to set up the neural network.

The "non-connectionist" school of artificial intelligence has tried to implement the black-box computer component by modelling it on existing systems other than the human brain. One of the most popular approaches is to model the computer on nature. Nature has succeeded in solving a rather complex problem, that of creating and sustaining life. In order to do this, simple living organisms were first introduced into

the environment. Then these organisms underwent a transformation process through evolution, lasting many millions of years. The current set of organisms that inhabits the world is far stronger and better adapted to its environment than its predecessors. For example, the ratio of diameters of blood vessels in the human body allows for better flow according to modern fluid dynamics [Hietkotter, 1995]. But this was a result of evolution and not some individual's calculations. So if problem-solving is modelled on evolution, it may be possible to discover solutions that are optimal or better than the analytical ones.

Evolution was a theory proposed by Darwin [Darwin, 1959] to explain the creation of life. He proposed that the nature of living creatures changed over the years to result in stronger specimens, better suited to the environment, being formed. The better specimens would then dominate and the lesser individuals would eventually cease to exist. This is commonly known as "survival of the fittest". This does not preclude the evolutionary process creating individuals that are less fit than their predecessors. In such cases, the new generation individuals would simply perish and their ancestors would continue to thrive, until they can generate better specimens.

This does not suggest that evolutionary techniques are the solution to all our problems. Evolution itself does not guarantee the creation of fitter individuals. It does however, explore many possibilities that may lead to stronger individuals. There is no ultimate goal or problem that must be solved by natural evolution. Instead organisms are constantly changed to suit the environment, which changes just as rapidly. Similarly, in an artificial environment of simulated evolution, solutions can be gradually adapted to satisfy the problem specification with greater accuracy.

According to modern theory of genetics, the fabric of our being is stored as a set of attributes in our DNA (genes). An individual's genes are like a blueprint to create that individual, since it is a complete description. When two parents mate to produce offspring, the children receive some genetic material from each parent. This crossing over of the genetic material allows nature to create individuals different from either parent.

For example, consider a monkey population where long tails are desired and long noses are not. If one parent with a long tail and short nose mates with another with a short tail and long nose, the offspring could have any combination of these features. If the child has a long nose and short tail, that child would not be very strong since it cannot hang from branches and its nose would always get in the way - it would probably not reproduce since none of the other monkeys would be attracted to a weak individual. On the other hand, a child with a long tail and short nose would be ideally suited to the monkey's environment. This child would be the fitter of the two and would propagate its genes in future generations.

Computer programs modelled on nature, normally associate possible solutions with the populations of individuals from nature. Then these solutions undergo a simulated evolution to attempt to produce better individuals. Just like nature, this process is quasi-random and solutions generated can be either better or worse than their parents. However, the probability of producing better solutions in this way is much higher than a blind random search through the solution space [Koza, 1992]. There exist many different approaches to this modelling, the most common being Genetic Algorithms, Evolutionary Programming, Evolution Strategies and Genetic Programming [Kinnear, 1994]. Collectively these are known as Evolutionary Algorithms. An evolutionary algorithm has the following general structure :

```
initialise a random generation of individuals
Pop = initpopulation (G)

evaluate the fitnesses of individuals in the population
evaluate (G)

while not done do
// select couples for reproduction
Pop1 = select (Pop);

// apply genetic operations to genes
Pop1 = genetic operations (Pop1);

// evaluate fitnesses of new population
evaluate (Pop1);

// merge new individuals into the existing population
Pop = merge (Pop1);
```

Genetic Algorithms

In order to understand Genetic Programming, it is first vital to consider the alternative approaches to evolutionary programming that led to its creation. Most discussions on genetic programming begin with an explanation of genetic algorithms, being the direct predecessor of genetic programming [Koza, 1992; Andre, 1994].

Genetic Algorithms (GAs) are evolutionary programs that manipulate a population of individuals represented by fixed-format strings of information. Their acceptance as a means to solve real-world optimization problems is readily attributable to the theory of artificial adaptation discussed in the ground-breaking work of Holland [Holland, 1992]. An initial population of individuals (solutions) is generated for the problem domain and these then undergo evolution by means of reproduction, crossover and mutation of individuals until an acceptable solution is found.

Genetic algorithms, like most other evolutionary computation techniques, require that only the parameters for the problem be specified. Thereafter the algorithm applied to search for a solution is mostly problem-independent .

As an inheritance from its biological counterpart, in genetic algorithms each character in the individual's data string is called a gene. Each possible value that the gene can take on is called an allele. These concepts are elaborated upon in numerous texts on biological genetics e.g. Hartl [Hartl, 1988].

For the purposes of the following discussion of genetic algorithms, the problem being solved is finding the square root of 2.

Representation of Problem

The representation of the problem domain is one of the most important factors when designing a genetic algorithm. Genetic algorithms usually represent all solutions in the form of fixed length character strings, analogous to the DNA that is found in living organisms. There are a few genetic algorithm implementations that make use of variable-length strings and other representations [Michalewicz, 1992] but these are not common. The reason for the fixed length character strings is to allow easier manipulation, storage, modelling and implementation of the genetic algorithm.

Consider the example of finding the square root of two. The first step would be to identify a possible range of solutions. Assuming no knowledge of the solution, it would be possible to deduce that the solution lies between zero and the number itself (in this case 2). Since it is known that the square of 1 is one, all numbers less than one can be removed. Also, the square of two will produce 4 so that can be eliminated as well. Thus the range is reduced to numbers greater than 1 and less than 2 - no solution to this problem can lie outside of this range. Of course, negative numbers can also produce the same results but since negative numbers are only different in sign, only the positive numbers need be considered. The next step is to represent all numbers between 1 and 2 with a fixed length character string. Binary numbers are usually utilised for numerical computations such as this. The reasons for this are outlined below. Binary numbers also allow for easy conversion to and from the exact solution. However, since there are obviously infinitely many real numbers between 1 and 2, fixed-length strings pose an additional problem for the programmer. To solve this, the real number range must be discretized into a finite number of constituent real number segments, corresponding to each binary number used in the character string. Suppose that the character strings have a length of n=10. Then the possible values for the character string would be from 0000000000 to 1111111111.

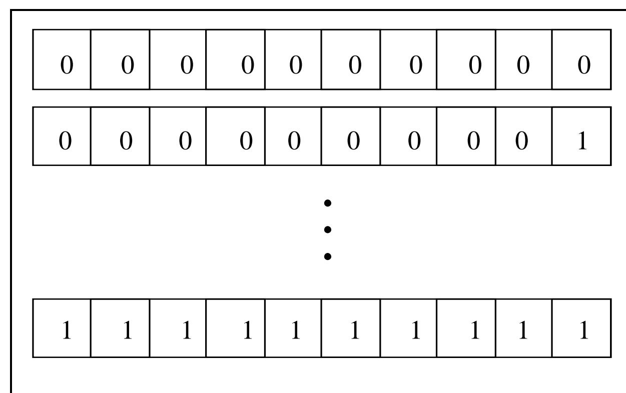

Figure 1.1 Bit-string GA representation

These binary numbers must be mapped onto the range of possible solutions, viz. the numbers between 1 and 2. There are 1024 (2^n) distinct numbers in the binary range, hence the numbers start from 0 and end at 1023 (2^n -1). The 1 (solution space) is

mapped onto the 0 (binary) and the 2 (solution space) is mapped onto the 1023 (binary). All other binary numbers are mapped linearly onto the real solution range.

binary	***binary value***	***real equivalent***
0 0 0 0 0 0 0 0 0 0	0	1 + (0/1023) = 1
0 0 0 0 0 0 0 0 0 1	1	1 + (1/1023)
0 0 0 0 0 0 0 0 1 0	2	1 + (2/1023)
⋮	⋮	⋮
1 1 1 1 1 1 1 1 1 1	1023	1 + (1023/1023) = 2

Figure 1.2. Conversion from bit-string to real representation

One of the reasons for using binary numbers is to disallow incorrectly formatted solutions automatically. Every combination of 1's and 0's corresponds to a possible solution. Decimal numbers can be used but since the solution range is between 1 and 2, a remapping process would have to be carried out to exclude the numbers greater than 2 or less than 1. In binary, it is easier to visualise some characteristics being present (by a 1) or absent (by a 0). This is more applicable to non-numeric problem domains. In addition, there are only two possible binary values (1 and 0). This means that all possible binary values can be generated by these two values. Thus the binary individuals 0000000000 and 1111111111 contain all the genetic material possible i.e. they span the solution space. With representations of a larger order (e.g. decimal), the number of individuals needed to span the solution space is much larger and this has repercussions on the speed at which the genetic algorithm finds a solution and the size of the parameters needed.

Population of Solutions

A collection of possible solutions is kept throughout the life cycle of the genetic algorithm. This collection is generally known as the population since it is analogous to a population of living organisms. The population can be either of fixed or variable size but fixed size populations are used more often so that the exact amount of computer

resources can be pre-determined. The population of solutions is stored in main memory or on secondary storage, depending on the type of genetic algorithm and computer resources available.

At the very beginning of the algorithm, a population of solutions is generated randomly. In the case of the square root problem, a fixed number of 10 character binary strings are generated randomly.

individual no	*random individuals*									
1	0	0	1	0	0	1	0	1	1	0
2	0	1	1	0	1	1	0	0	1	1
3	1	1	0	1	0	0	1	1	1	0
⋮	⋮									
100	1	0	1	0	0	1	0	0	0	1

Figure 1.3. Initial random population

This population is then modified through the mechanisms of evolution to result eventually in individuals that are closer to the solution than these initial random ones.

Fitness

Darwinian evolution of a population implies that the strongest individuals will survive. To implement such a principle necessitates a means of evaluating the relative strength, or fitness, of each individual. In terms of the genetic algorithm, the fitness of an individual is a numerical assessment of that individual's ability to solve the problem at hand - it is the ability of the individual to satisfy the requirements of the environment.

In terms of the square root problem, the perfect individual is the numerical value approximated by 1.414213562373. This can therefore be regarded as the fittest solution. Since fitness is quantified numerically, maximum and minimum fitness values of 1 and 0 are normally used. According to this scale, the perfect solution

above represents a fitness of 1. The minimum fitness must be the absolutely worst solution possible, to ensure that all solutions are in the range 0-1. In the square root problem, the worst solution is “2”, hence the fitness of the solution “2” would be 0. Although it is possible to find distinct best and worst case values in this problem it is not possible for all problem domains. However, every possible individual in the solution space must be restricted to the fitness range 0-1.

Fitness is normally defined as a function that takes as its single parameter the individual and returns a real number representing the fitness value of that individual. Fitness cannot be calculated by comparing the perfect solution with the individual simply because the perfect solution is not known at the time of calculation. Thus it has to be calculated from other information in the specification. In the case of the square root problem, the fitness of an individual can be calculated by squaring its numerical value and then comparing this to 2. The results can then be scaled to fit in the range 0 to 1. The following fitness function satisfies these criteria.

$$Fitness(x) = \frac{Abs(x^2 - 2)}{2} \quad \ldots\ldots\ldots\ldots\ldots\ldots \quad (1.1)$$

In addition to assigning the boundary values, the fitness function must also be able to assign values to every other solution in the solution space. The intuitively better solutions must be allocated better fitnesses than the worse solutions. This is necessary so that the better solution can be selected over the worse one when comparisons are being made. For numerical calculations the fitness function is chosen as a relative error (as is done above in *Equation 1.1*) to achieve this aim. In economic problems, the profit can be used to generate a fitness function - greater profit tends towards a perfect solution while lesser profit has lower fitness values.

random individuals	binary value	solution	fitness
0 0 1 0 0 1 0 1 1 0	278	1.2717	0.1913
0 1 1 0 1 1 0 0 1 1	435	1.4252	0.0156
1 1 0 1 0 0 1 1 1 0	846	1.8270	0.6689
⋮	⋮	⋮	⋮
1 0 1 0 0 1 0 0 0 1	657	1.6422	0.3485

Figure 1.4. Selected individuals with corresponding real values and fitnesses

The table in *Figure 1.4* represents some sample solutions in the initial random population, together with their associated actual values and their fitnesses. The best solution displayed is in the second line, as it has the lowest fitness - it is also the value closest to the perfect solution, as expected.

Reproduction

The vehicle of all evolutionary change in the genetic algorithm is reproduction. The reproduction operation allows the population to progress from one generation into the next. This progression occurs in the most natural way possible, favouring the fitter individuals. Individuals are selected from one generation of the population to be injected into the next generation. This new generation is a permutation (with duplicates) of the original population and when completely formed, it replaces the original population.

The selection process is based on the fitnesses of the individuals. Generally, individuals with a higher fitness are selected more often than individuals with a lower fitness. There have been many strategies to implement this tendency to select fitter individuals.

The most common method is called fitness-proportionate reproduction. In this approach, the probability of selecting each individual is proportionate to its fitness. Thus the fitter individuals get selected more often than the less fit individuals. This

leads to some individuals being selected more than once and others not being selected at all, which is only natural as the better individuals flourish while those that are not good enough perish.

The roulette wheel implementation implicitly forces fitness-proportionate reproduction. In this approach, the fitnesses of all individuals in the population are arranged into a list and then summed. A random number in the range of the sum is generated. Then the fitnesses in the list are summated again until the random number is reached or exceeded. The last individual in the list is the one chosen. The method works because the individuals with higher fitnesses occupy a larger portion of the range from which a random number is being selected - therefore they can be selected more often. This process is repeated until enough individuals are selected to replace the whole of the last generation.

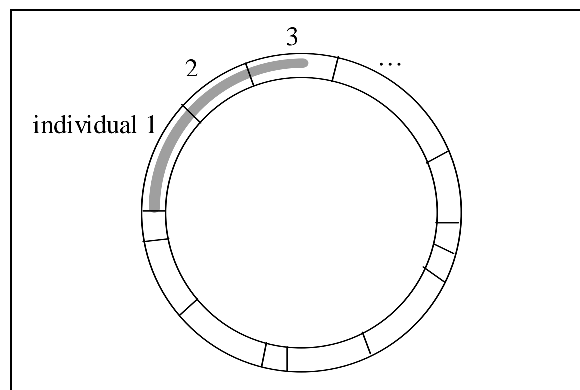

Figure 1.5. Roulette wheel individual selection

Another common approach to selecting individuals is tournament selection. Two individuals are selected from the population and their fitnesses are compared. The one with the higher fitness is progressed into the next generation. The tournament can also be carried out among more than 2 individuals (K-tournament selection).

Elitism is a strategy where the highly fit individuals are explicitly favoured. This can be useful when the fitnesses are linear and the problem has a single solution. However, most fitness functions do not produce a linear relationship between individuals and their fitnesses i.e. there are local minima in the range of fitness values.

The restrictive nature of elitism could cause convergence to one of those local minima, which is most likely a far from optimal solution.

Crossover

Reproduction on its own cannot cause a population of solutions to evolve since the individuals from one generation are simply being copied into the next generation of the population. In order for the fitnesses of individuals to improve, there must be a sharing of genetic material. Crossover swaps some of the genetic material of two individuals, creating two new individuals (children), who are possibly better than their parents. This is analogous to genetic crossover as observed in living organisms.

In genetic algorithms, crossover is implemented by selecting a point in the character string and swapping all characters after that point. This selection point is generated randomly and the operation is applied to two individuals of the newly reproduced population.

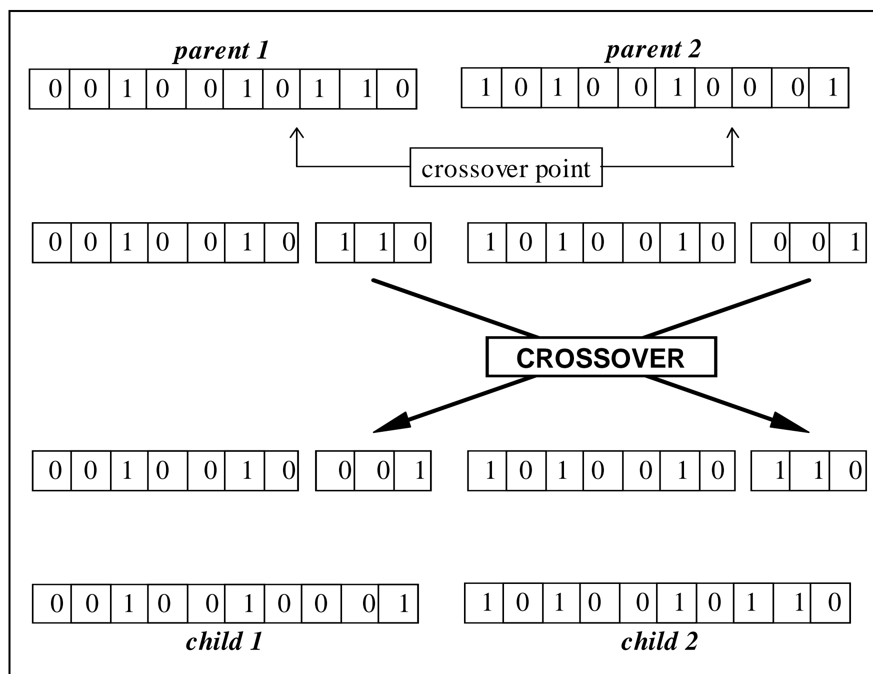

Figure 1.6. Crossover of two individuals in GA

The result of the crossover genetic operation is two individuals who are possibly fitter than their parents. In any event, these individuals are added to the new generation

being created. The simplest strategy is to replace the parents with the children. That way each parent only participates in crossover once. An alternative is to inject the children into the population and replace a pair of individuals with relatively low fitness. Using fitness-proportionate reproduction, this strategy is unnecessary since the population potentially contains more than one copy of the fitter individuals.

This genetic operator does not have to use only one crossover point. Instead, many crossover points can be chosen, and the genetic material exchanged at each point. If two crossover points are chosen, then, effectively, the genes between the points are exchanged.

Mutation

During reproduction, fitter individuals in a population are selected more often than others. This leads to some individuals not being selected for promotion into the next generation. These are generally the least fit individuals. However, they may contain within their structure genes which are part of a better solution. This genetic material is lost to the population since the individuals are no longer propagated.

In order to recover from this loss of genetic material, the individuals are allowed to change their genes randomly. This is a slight perturbation in the genetic material which occurs with a much lower frequency than crossover. A random point or points are chosen in the character string. A random allele is then generated and inserted at each of the mutation points.

Like crossover, mutation can create individuals who replace their parents in the new generation, or they can be added to the population. Individuals must be removed so that the population does not grow unmanageably large. The primary reason for this is to make genetic algorithms feasible for practical implementation.

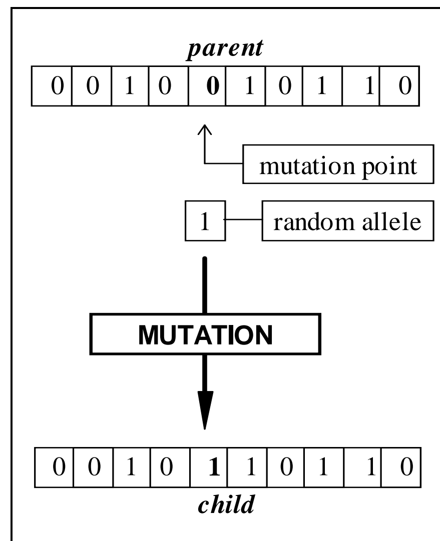

Figure 1.7. Mutation of an individual in GA

General Algorithm

```
//start with an initial generation
G = 0

//initialise a random generation of fixed-format strings
Pop = initpopulation (G)

//evaluate the fitnesses of individuals in the population
evaluate (G)

while not done do
// increase generation counter
G++

// generate new population using fitness-proportionate
reproduction
Pop1 = select (Pop);

// crossover genes
Pop1 = crossover (Pop1);

// mutate genes
Pop1 = mutate (Pop1);

// evaluate fitnesses of new population
evaluate (Pop1);

// replace population with new generation
Pop = Pop1;
```

There are various alternatives and modifications of this algorithm but the essential structure is always the same. One common change is to incorporate the reproduction

operation into the crossover and mutation operations - individuals are selected fitness-proportionately, crossed over (or mutated) and inserted into the new generation in a single operation.

John Holland's Schema Theorem [Holland, 1992] is widely accepted as mathematical proof that the genetic algorithm, due to its fitness-proportionate reproduction, converges to better solutions. According to the schema theorem, individuals are grouped into schemata according to particular subsets of their genes. The number of individuals in each group converges if the fitness of that group relative to the entire population is high, and vice versa. This result is slightly modified by the crossover and mutation operations which create new individuals from the existing population, implicitly changing the schemata into which individuals fall.

Evolutionary Programming and Evolution Strategies

Genetic algorithms are just one example of a paradigm of evolutionary programming. Other techniques were created, with many similarities to genetic algorithms as discussed by Heitkotter and Kinnear [Heitkotter, 1995; Kinnear, 1994].

Evolutionary Programming, conceived by Fogel in 1960, uses only mutation as a means to improve the fitness of individuals. Individuals can be represented by any convenient syntax, since there is no crossover operation. The population is propagated from one generation to another by applying the mutation operation in varying degrees according to the proximity of the individual to the expected solution.

Simultaneously with the development of evolutionary programming, a group of students in Germany, Rechenberg and Schwefel, developed a strategy to optimise shapes of bodies in a wind tunnel. Their technique uses a population of solutions, changed by normally distributed random mutations. Each individual contains both objective and strategy variables - objective variables are representations of the problem domain while strategy variables indicate the decreasing mutation rates to be deployed.

Genetic Programming

Genetic algorithms, although very useful for simple problems, can restrict complex problems due to its inability to represent individuals other than fixed-format character strings. Genetic Programming is a generalisation of genetic algorithms devised by Koza [Koza, 1992]. It is readily accepted that the most general form of a solution to a computer-modelled problem is a computer program. Genetic Programming (hereafter known as GP) takes cognizance of this and attempts to use computer programs as its data representation.

Similarly to genetic algorithms, genetic programming needs only that the problem be specified. Then the program searches for a solution in a problem-independent manner. Most genetic operators can be implemented, albeit somewhat differently from its predecessors. Although Koza has suggested definitional guidelines for GP, these have been relaxed in attempts to achieve greater efficiency with reduced computer resources.

Representation

Each individual in a genetic program is a computer program. However, this definition is a little vague since there is no general structure for all computer programs. On different platforms with differing compilers and interpreters, the structure of the programs can be different. GP is not specific in this regard - it can be applied in all cases.

Most classical programming languages can have their programs represented as sequences of functions. These functions can operate on constants or variables or the results of other functions. This lends itself to a tree structure for a typical program. Computer programs in GP are viewed as free-format trees, consisting of leaves (variables and constants) and non-terminal nodes (functions).

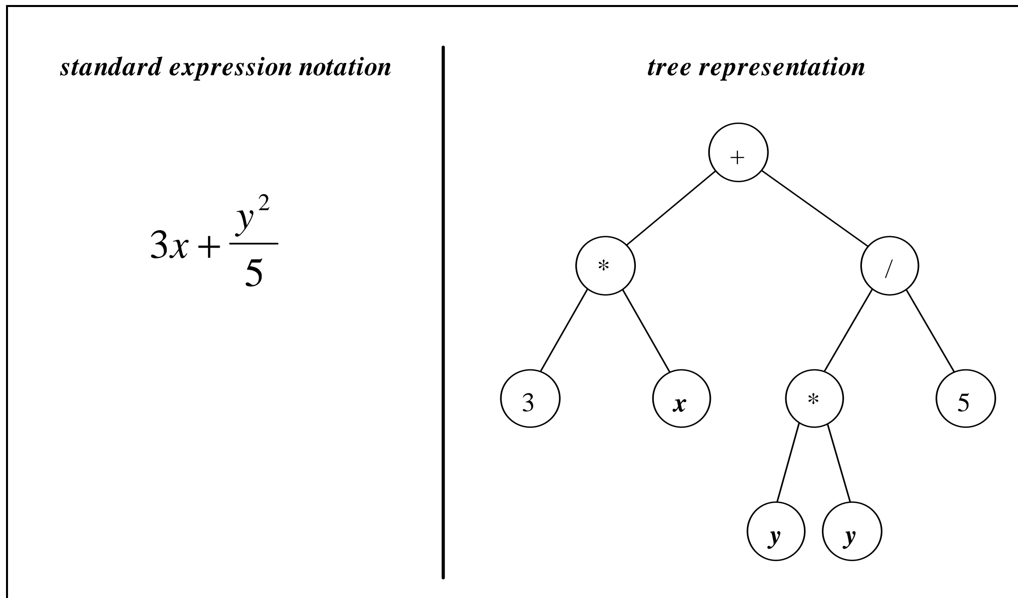

Figure 1.8. Representation of individuals as trees in GP

Any mathematical expression can be considered as a computer program since it takes input, processes the input and produces output. The expressions in *Figure 1.8* are therefore proper programs and can be used to generalise the capabilities of the GP algorithm. The tree representation indicates how the GP ought to store the program internally. The method of storage is not critical as long as the algorithm can manipulate the individual solutions as trees.

In the illustrated example, there are only two variables, two constants and three functions, which totally define the expression. However, real-life computer programs can use many hundreds of variables and functions to solve a modestly complex problem. Although such problems are still not feasible for solution by GP, it has been recognised that the number of variables and functions has a significant impact on the efficiency and scale of GP. Hence, the number of variables, constants and functions needs to be reduced by eliminating those not necessary in a particular problem domain. The functions, appearing only in intermediate nodes, are called the non-terminals. Variables and constants, appearing only on the leaves of the tree, are appropriately called terminals. The non-terminal set for the example is {+, /, *} and the terminal set is {x, y, 3, 5}.

The terminal set is the set of all alleles that can appear at the leaves of a GP tree while the non-terminals are the acceptable functions. These two sets define the search space for the problem - every tree constructed has to get its nodes from the terminal and non-terminal sets. The size of the search space is determined by the sizes of these two sets. An increase in the size of the non-terminal set results in a linear increase in the size of the search space. However, an increase in the size of the terminal set results in an exponential increase in the search space size, since the combinations of parameters available to every function is also increased.

On the other hand, if a terminal or non-terminal set does not contain sufficient variety, it may not be possible to represent some solutions. For example, the expression "-3" cannot in any way be represented by selecting terminals and non-terminals from the given sets. Thus there are two important considerations when selecting terminal and non-terminal sets. Firstly, the set must span the solution space completely. Secondly, these sets must be as compact as possible, to prevent extraneous searches.

For example, if Boolean functions are being considered, then the non-terminal set needs only contain {AND, OR, NOT} [Koza, 1992]. These functions are not the absolute minimum to span the solution space, but the inclusion of a small degree of redundancy allows for the formation of smaller computer programs (expressions).

Koza has also suggested that every function in the non-terminal set must operate only within the scope of the terminal set. The functions must be capable of taking on every combination of terminals possible, and the return values must be in the range of the terminal set. By requiring this of all functions, there is no possibility of parameter incompatibilities. It also allows functions to be nested without restriction. This is an obvious feature of some functions but exceptions must be catered for. If the terminal set contains integers and the non-terminal set the standard operators {+, -, /, *}, then division by zero is a distinct possibility. To cater for this, the division operation can be modified or overloaded so that division by zero returns a large number instead of an error. This protection of functions enables closure of the non-terminal set.

Alternatives to closure include the use of strongly-typed GP, where each non-terminal has a pre-specified return value type, which may be different for various functions.

Haynes [Haynes, 1995] has used this strategy successfully to optimise an artificial predator/prey scenario in a manner better than the standard GP.

Population of Solutions

Similarly to a GA, genetic programming first constructs a population of random individuals and then processes these by simulated evolution. The random individuals in this case are random trees. Due to the closure property of the non-terminal set, it is possible to recursively create any combination of terminals and non-terminals.

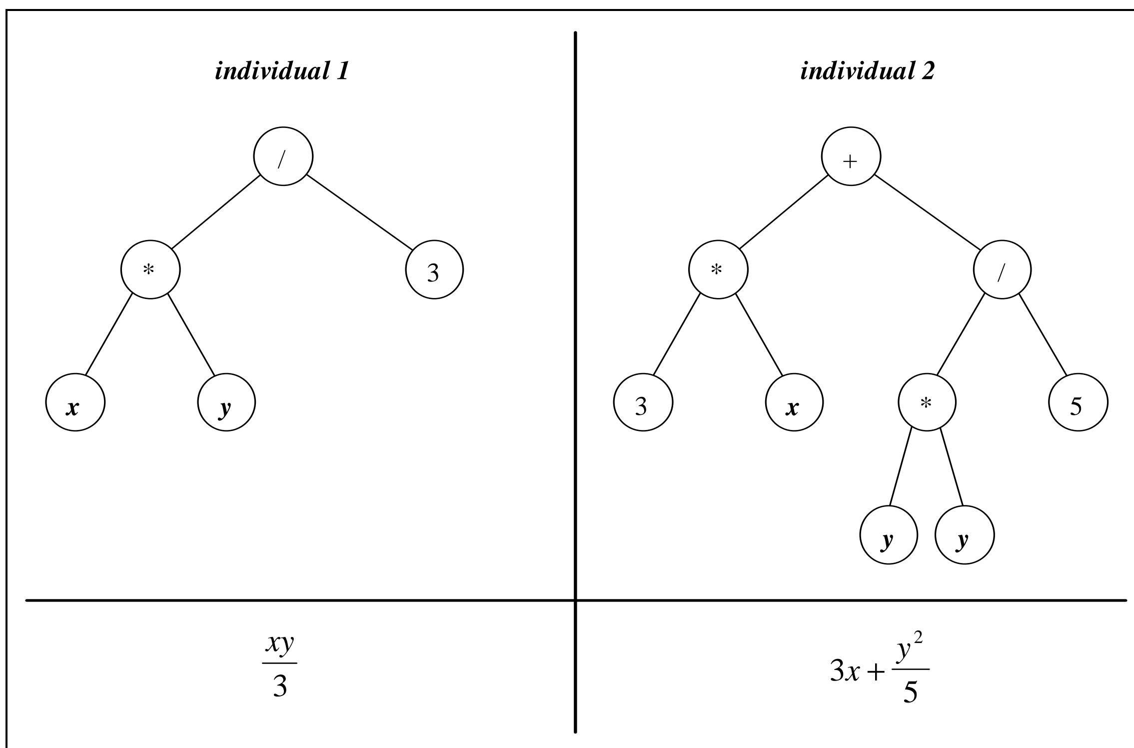

Figure 1.9. Extract from population of GP trees and corresponding expression representation

Populations in GP are normally much larger than those in genetic algorithms. This is chiefly because of the unrestrained nature of the representation. While a GA allows only fixed-format strings, trees have much greater diversity of size and structure. To accommodate this greater diversity, larger populations are necessary.

Fitness

Since individuals are represented as computer programs, the obvious method of testing effectiveness of the solutions would be to execute the programs. Then some means of measuring the performance (error, time taken, etc.) can be used as the fitness measure. This adds extra overhead to the GP algorithm since each individual has to be executed to determine its fitness. Also, most programming languages do not support the execution of data items or dynamic conversion between data and code. In such cases, an interpreter has to be incorporated into the algorithm.

The raw fitness of an individual is the fitness value calculated directly from the execution of the program. This value is not bound to any range so its needs to be modified before it can be used constructively. The standardised fitness converts the raw fitness to a zero-centric function - the standardised fitness of an individual is zero for the best individual and higher for individuals of lower fitness. The standardised fitness attempts to restrict the fitnesses to the range of positive real numbers only. The adjusted fitness changes the fitness value so that it lies strictly within the 0-1 range. This is useful to standardise the result designation and make statistics more meaningful. The adjusted fitness can be generating trivially from the standardised fitness by the following function.

$$AdjustedFitness(x) = \frac{1}{1 + StandardizedFitness(x)} \quad .... \quad (1.2)$$

Kinnear [Kinnear, 1994] stresses the importance of using a fitness function that not only generates the right boundary conditions but also allocates appropriate fitness values for all other expressions. If partial credit is not given for containing features that lead to a better solution, then the fitness function would not be effective.

Reproduction

Fitness-proportionate reproduction in GP is identical to GAs, since the change in representation has no effect on the copying of individuals. In order to produce a new generation, only the fitnesses need be known, and these are gleaned from the adjusted fitness function applied to all the individuals in the original population.

Crossover

Crossover is applied to a pair of individuals from the newly reproduced population in order to exchange genetic material. In the case of the classic GA, genetic material took the form of sub-strings of the character string representation. GP, on the other hand, exchanges sub-trees of the individuals in order to create new individuals. Since the non-terminals have achieved closure, it is possible to exchange a sub-tree rooted with a non-terminal with one rooted by a terminal since the non-terminal function produces a return value in the range of the terminal set.

Another difference between GAs and GP is in the selection of crossover points. In GAs, a single crossover point was chosen and applied to both individuals. In GP this is not possible since the individuals may have different structures, so instead different crossover points are generated for each individual.

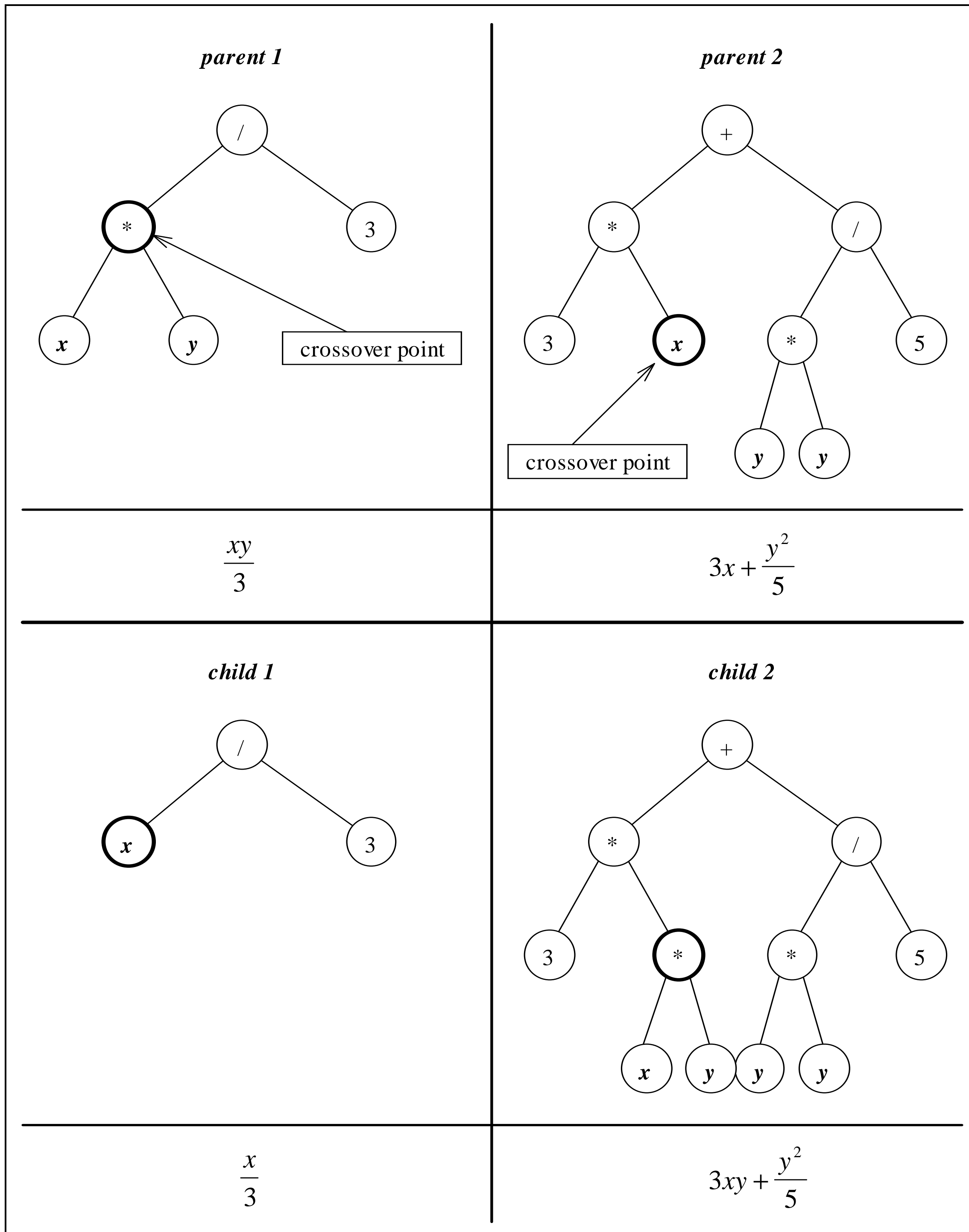

Figure 1.10. Crossover of two individuals in GP

Mutation

Mutation is not necessary in GP because the large population sizes almost always ensure that the genetic material cannot be easily lost. However, large population sizes

require lots of resources and, in the absence of these, steps have to be taken to recover the genetic material. Also, taking into account the successes of mutation-based evolutionary computing, this genetic operator cannot be simply ignored.

Just as in crossover, mutation is applied to a randomly chosen sub-tree in the individual. This sub-tree is removed from the individual and replaced with a new randomly created sub-tree.

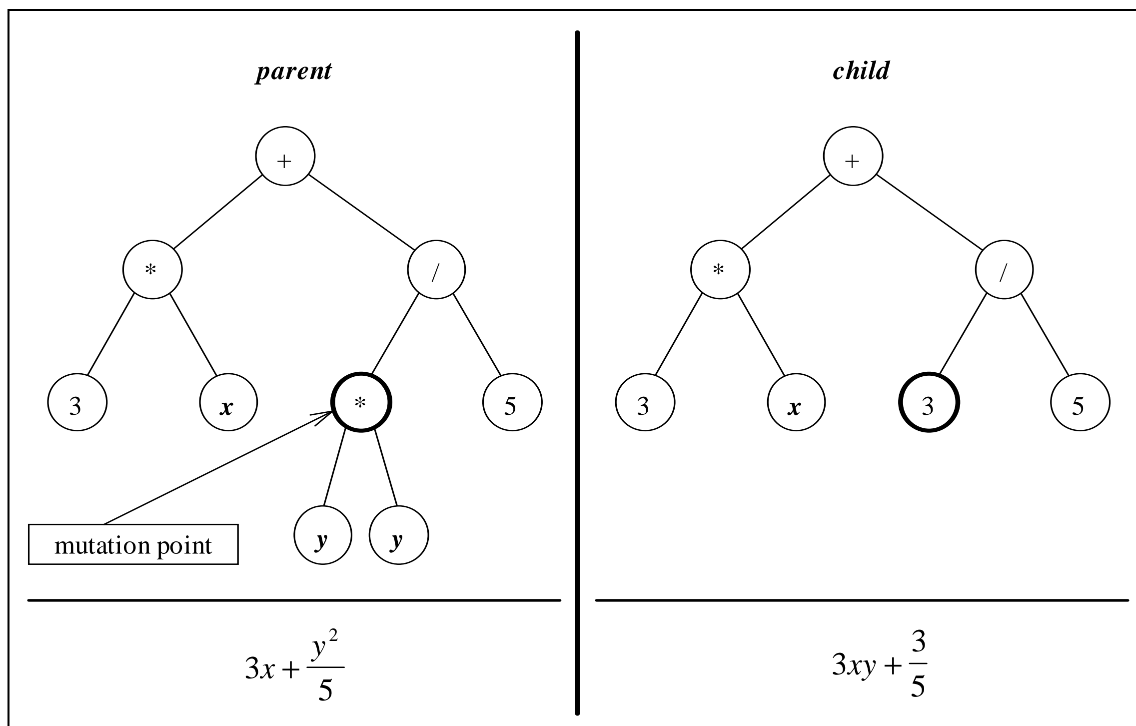

Figure 1.11. Mutation of an individual in GP

General Algorithm

```
// start with an initial generation
G = 0

// initialise a random generation of trees from the terminals
and non-terminals
Pop = initpopulation (G)

// evaluate the fitnesses of individuals in the population
evaluate (G)

while not done do
// increase generation counter
G++

// generate new population using fitness-proportionate
```

```
reproduction
Pop1 = select (Pop);

// crossover sub-trees
Pop1 = crossover (Pop1);

// mutate sub-trees
Pop1 = mutate (Pop1);

// evaluate fitnesses of new population
evaluate (Pop1);

// replace population with new generation
Pop = Pop1;
```

It is apparent that the general algorithm for GP is nearly identical to the GA. As far as implementation is concerned, the major difference is in the representation. But this difference is sufficient to necessitate changes in the genetic operators and all other manipulation routines in the algorithm. There are also implicit differences that affect the efficiency or conceptualisation of GP as compared to standard GAs.

Applications of GP

In traditional evolutionary algorithms, the optimization of existing solutions is a large research area because the algorithms are more suited to slight perturbations rather than outright changes (evolutionary programming and evolution strategies). GAs have the limitation that the structure of the solution needs to be known in advance in order that it may be modelled by the fixed character string. Although some work has been done on variable-length GA strings, this is sufficiently different from the original algorithm to fall within the ambit of GP itself. GP has no such restrictions on representation therefore the scope of applications is much broader. In an ideal situation, any application which requires a solution in the form of a computer program can be solved using a GP.

Koza [Koza, 1992] applied the GP to many benchmark problems that are still used to test the capabilities of GP systems. The most famous of those problems is that of symbolic regression. A set of points is generated from some test data and an equation passing through the points is sought. There exists no definite analytic method to find such an equation if the form of the equation is not known in advance. Statistical methods assume a form for the equation and then try to optimise the coefficients for

the equation. GP can find both the structure and the coefficients for the equation. Oakley successfully extended symbolic regression to chaotic data [Oakley, 1994].

Another popular area of application is the control of artificial animals and robots. Reynolds generated programs to control a robot in order to avoid obstacles [Reynolds, 1994]. Spencer used GP to teach a 6-legged robot how to walk, in terms of the sequence of mechanical actions that had to be performed [Spencer, 1994].

Economic optimization, a complex field for analytical study, has also lent itself to evolutionary computation techniques. Andrews modelled a double auctioning system which used GP to generate a better automatic auctioning program than those previously known [Andrews, 1994].

Koza et al have applied GP to the problem of designing electrical circuits. They trained an artificial animal in maximal food foraging - the algorithm being produced in the form of an electronic circuit discovered by GP [Koza, 1996-1]. In a similar manner, an electronic circuit was successfully built to implement an operational amplifier with desirable amplifier characteristics [Koza, 1996-2].

Andre used GP to learn rules for optical character recognition [Andre 1996]. It is a laborious task to write rules manually to distinguish among different characters in a character set, especially when different fonts and sizes are used. GP successfully found rules to classify characters with few errors.

GP can also be applied to classification problems. A finite automaton, when duplicated and arranged in a regular formation, can exhibit aggregate behaviour about the total structure. A classic problem is to find a boolean-valued automaton that relaxes the total automaton into a steady state corresponding to the value that occurred most often in the start state. This is known as the Majority Classification Problem and can be solved in numerous ways. Andre used GP to find a rule for the cellular automata that was better than any previously known rule (for a particular configuration) [Andre, 1996].

Hand-in-hand with new applications of GP goes the development of new implementations. The early Koza-based implementation of GP was done in LISP, but

attempts are being made to port the GP paradigm to other programming environments. C++ and other 3GLs are useful for implementation but require complex modelling for non-trivial problems. Other platforms (eg. Mathematica) are considered to circumvent this complexity.

CHAPTER 2 :
A MATHEMATICA IMPLEMENTATION

Implementation Languages

Lisp

The first implementations of GP done by Koza used the LISP programming language [Koza, 1992]. LISP (LISt Processor) has some unique characteristics compared to other commonly used languages, which makes it an ideal platform for the implementation of GP.

In LISP, there are only two basic syntactic constructs. The atom is a terminal part of an expression, being either a variable or constant. The other construct is a list. Any program can be represented solely using lists of atoms. Lists can also be nested and embedded recursively. Lists use a prefix notation, as opposed to popular programming languages which prefer infix notation for its more obvious interpretation. These lists in LISP are known as S-expressions.

LISP	**normal interpretation**
(+ 1 2)	1+2
(* a b)	ab
(+ (* a b) (/ c d) 8)	$ab + \frac{c}{d} + 8$

Table 2.1. Sample LISP expressions

It can be shown that all computer programs are essentially sequences of functions. LISP generalises this by requiring all programs to be in the form of a list. The first element of the list is the name of the function while the rest constitute its arguments. Thus, in *Table 2.1*, “+” is the name of the function and its arguments are the numbers “1” and “2”. These lists can also be represented as trees since they allow nesting. This

tree visualisation is ideal since GP requires a tree representation for its various manipulations.

LISP makes no distinction between code and data. Both the program and the data it works on are represented as lists. Thus it is possible to execute an item of data as if it was code. Alternatively, it is also possible to manipulate a program as if it was pure data. The primary reason why most people implement GP in LISP is because they can exploit this feature to make the evaluation of fitnesses easier. Instead of writing an interpreter to execute the individuals, they can be run directly on the computer by virtue of this almost unique LISP feature.

Although these features of LISP are conducive to a GP implementation, LISP is not widely used because programs do not execute fast enough (compared to 3GL languages) and compilers/interpreters are uncommon. It is used by AI researchers but not by many other people.

C++

In order to create a GP implementation that is both fast and portable, C++ is an ideal choice. Of the wide range of 3GL languages available, C++ compilers are available on most platforms. Thus the code can be written in a platform-independent manner. C++ also has an adequate library of functions to enable greater flexibility when designing internal representations and manipulation functions.

Keith discusses some of the problems that accompany a C++ implementation, especially the issue of representation [Keith, 1994]. Since tree structures are not native to C++, these have to be simulated using data structures. In a direct conversion from LISP, these trees can be created using pointers and objects. However, it is also possible to convert the tree into postfix or prefix notation and use a one-dimensional array to store the tree. These different methods have a direct effect on the functions that manipulate the expressions in terms of complexity and speed.

The greatest advantage of LISP over C++ is its ability to execute the individuals directly to gauge their fitnesses. C++ has to use an interpreter to perform this task. This interpreter will have to take the data structure that corresponds to an individual

and simulate execution. For simple problems, such an interpreter may be trivial to build, but a larger non-terminal set may require a complex interpreter on the scale of the compiler itself.

This can be a prohibitive factor since the interpreter will have to be written as part of the GP implementation. In addition, the problem domain will have to be modelled in C++. The complexity of such modelling cannot be predetermined so the effect of such is not obvious. However, without the aid of function libraries, mathematical modelling in C++ is a non-trivial task which may require more development time than the actual GP algorithm.

Mathematica

Mathematica is an environment in which mathematical computations are easily performed. It is essentially an interpreter which takes expressions as input and attempts to make conclusions from these expressions. Most Mathematica users only utilise this subset of its capabilities.

Mathematica can be compared to the BASIC (Beginners All Purpose Symbolic Instruction Code) interpreter which was bundled with the older versions of MSDOS (MicroSoft Disc Operating System). It can execute one command at a time or it can take input from a file, thus processing a batch of input at once. This batch processing allows the user to write programs in Mathematica.

Mathematica stores all expressions internally as trees. This makes it easier to implement GP in Mathematica since GP requires a tree representation. Mathematica also has available a library of functions for manipulation of these trees, and these are useful for genetic operators.

Similarly to LISP, Mathematica makes no distinction between program code and data. Thus a program can be manipulated and modified as if it was plain data, and data could be executed as if it was code. Unlike C++, it is unnecessary to use an interpreter to evaluate the fitnesses of individuals, since the individuals can be executed within the framework of the Mathematica environment.

The most important factor supporting the implementation of GP in Mathematica is the large body of existing and ongoing mathematical modelling in this environment, as demonstrated by the number of conferences and publications devoted to it. Mathematica is becoming a platform of choice because of its ingrained orientation towards the analysis and presentation of mathematical solutions. The ease with which complex problems can be implemented in Mathematica makes it feasible to implement GP on this platform. Since GP is problem-independent, the majority of work done to solve a problem is in the modelling stage. By choosing a platform like Mathematica which supports easier modelling, productivity can be increased.

Nachbar was the first person to document a GP implementation in Mathematica but, subsequently, there has been little work done in this field [Nachbar, 1994]. This study explores the implementation of GP on a Mathematica platform, making full use of the multiple paradigms, optimizations and other advanced features available in the language.

Introduction to Mathematica

The following overview of Mathematica is focused on the aspects that are relevant to the GP implementation. A more in-depth discussion can be found in [Wolfram, 1991], [Wolfram, 1992], [Wickham-Jones, 1994], [Maeder, 1991] and [Abell, 1992].

Platforms and Organisation

Mathematica is available on many different hardware platforms and operating system combinations e.g. DOS, Windows 3.x, Sun, Silicon Graphics. However, the underlying kernel of the environment is the same in all instances. This kernel is a single-line text input processing system. A line of Mathematica code is typed in at the keyboard, this expression is immediately evaluated and the results are output to the screen.

In modern GUI (graphical user interface) operating systems, this method of inputting data into the environment would not be acceptable since it does not conform to the user interface and the advantages of the operating system would be lost. To make Mathematica easier to use, a front-end processor was included. This is a graphical

program that takes input from the user in the most natural way possible and passes this input to the Mathematica kernel. The output from the kernel is then re-directed back to the front-end, which formats it in a more natural way. The input and output are both displayed as a single document, much in the same way as a word processor displays a text document. This allows the user to edit and re-evaluate expressions, which could not be done in the line-by-line version. Also, having both the input and output on a single page allows for easier publishing of results from the session. This document, containing Mathematica input, output and other formatting is known as a Notebook.

Variables

Mathematica can do both numerical and symbolic calculations, attempting at all times to produce a result which is as accurate as possible. If the answer to a calculation is a fraction, then that fraction would be output instead of its numerical equivalent, to preserve computational precision.

The basic data types are String, Integer and Real. These can then be compounded into lists. Values are assigned to variables by means of the standard assignment operator "=".

```
X=12
```

In an actual Mathematica environment, these input and output operations may be preceded by an internal numbering system, which allows the user to refer to results from previous calculations.

After such a definition, all occurrences of **X** (taking case into account) are replaced by its associated value. If the input is simply **X** then the output would be "12". Obviously, the value of one variable can be assigned to another using the same syntax. Variables can be created on-the-fly, without the need to declare the list of variables in advance. A list of values is denoted by curly braces.

```
TestList = {1, 2, 3}
```

There are no pointers in Mathematica since it does its own memory management. Lists can grow as large as memory and hard disk space (used for virtual memory)

allow. They can be embedded and nested to form trees, which are the most general form of data structure directly supported in Mathematica.

Functions

Mathematica is first and foremost a functional programming language. It contains a large collection of pre-defined functions and allows the user to define further functions or even enhance the built-in definitions. A program in Mathematica is simply a sequence of calls to these functions. These calls can themselves be embedded within another function, allowing modular programming.

Functions are called by the exact name of the function, followed by the parameters within square brackets. For example,

```
Plus[2, 2]
```

would produce the following output:

```
4
```

All operations without exception can be written in this form. Even simple functions like addition and subtraction can use this notation. However, in order to make inputting of expressions easier, the kernel allows an alternative notation for some common expressions, like addition and multiplication. Thus the expression

```
2+2
```

is equivalent to the one above and would produce the same output.

Function calls can be nested and the expression is then evaluated depth-first (in most cases). Thus it is possible to write

```
Times[12, Plus[2, 1]]
```

which would evaluate to “36”.

Functions are defined using the following general syntax:

```
NewFunction [x_, y_] := 2 * x + y
```

The name of the function will be **NewFunction**. This will be added to the list of built-in functions. There is no distinction between built-in functions and user-defined functions, allowing the Mathematica environment to be easily extended.

The parameters within brackets are the formal parameters. The underscores after the names of the formal parameters indicate that they are simply placeholders for actual parameters. Mathematica uses a system of pattern-matching to implement its function mechanism. When the function is called, the actual parameters are replaced for the formal parameters wherever they occur in the expression, then the expression is evaluated. If the underscores are omitted, Mathematica would try to match the exact parameters in the list, without any form of pattern-matching. Thus, only **NewFunction[x, y]** would be successfully parsed.

The ":=" indicates that the RHS expression is not to be evaluated until the function is used within another expression. This ensures that parameter substitution by means of pattern-matching gets highest precedence. If the colon was not prefixed to the assignment operator then the RHS would be evaluated when the function is defined; if **x** and **y** are global variables then their values would be substituted, instead of the parameters, and the result of the function would be that constant value generated.

The expression on the RHS of the function definition is the body of the function. The variables used are subject to parameter pattern-matching. The result of the function call is the evaluation of this expression. Thus

```
NewFunction [7, 3]
```

would result in

```
17
```

It is also possible to do symbolic calculations. Variables can be used as input to the function, whether they have a value or not. Consider the following code fragment:

```
a=12; NewFunction [a, b]
```

The output would be

```
24 + b
```

If two statements are separated by a semi-colon, then they are executed in sequence and the result of the expression is the result of the second expression. In the above example, `a` has an associated value while `b` does not. The kernel therefore replaces the a with its value when calling the function. The second actual parameter is `b` since it doesn't have a value. Thus the answer is as accurate as possible with the limited information provided. Using this technique of defining values for variables it is also possible to perform symbolic calculations in Mathematica.

Overloading of functions is an integral part of the environment, allowing for multi-part functions and different parameter types and ranges. Functions are very flexible when pattern-matching. It is possible to write functions that only accept parameters of particular types or ranges or even parameters that obey specific rules. Varying numbers of parameters are also catered for.

Paradigms

Although Mathematica focuses mainly on the functional aspects of programming, there are also mechanisms that enable the user to write procedural and declarative code.

By simple virtue of the fact that function overloading and pattern-matching is available, declarative programming becomes feasible.

Procedural programming relies on constructs that explicitly implement sequence, selection and iteration. Sequence is easily accomplished by consecutive lines of input, possibly separated by semi-colons. A selection mechanism is normally in the form of an "if" statement e.g. in C++ and Pascal. In order not to deviate too much from classical languages, such a construct is provided in Mathematica.

```
If [x==0, 1, 2]
```

Unlike simple functions, the parameters are not evaluated beforehand. The `If` function will evaluate the first parameter. If its result is true then the second parameter is evaluated, otherwise the third parameter is evaluated. The result of the entire expression is therefore the result of either the second or third parameter.

Iteration is implemented in a similar way. The functions are equivalent to their C++ counterparts. The **Do** function is equivalent to the fixed iteration ”for” statement in classical languages like C. **Do** has two parameters, the first being a block of statements and the second being an iteration specification. This specification takes the form of a list, where the first element is the name of the variable, the second the initial value, the third the final value and the fourth the step. There are many different ways of specifying a range of values for fixed loops, where some of these elements may be omitted in favour of default values. The following example prints the string “Hello World” ten times on the screen.

```
Do[Print["Hello World", {i, 1, 10}]
```

Conditional loops are implemented with the **While** function, which takes only two parameters. The first is an expression that is evaluated each time the loop starts, and terminates the loop once it is false. The second parameter is a block of statements that must be executed.

This multiple-paradigm approach to programming is beneficial since the problems can be modelled using any of these three methods. The best techniques of each paradigm can be incorporated into the code. For example, the definition of multiple clauses can be used with functions whose bodies are written in a procedural fashion. Being a functional programming language, however, Mathematica discourages the use of procedural constructs by providing the user with a rich set of functions that implicitly iterate over lists of data.

Modularization - Functions

Since all variables are created dynamically, it eventually happens that variables begin to overlap - i.e. a variable is used for different tasks in different parts of the program. This is not critical until the value of a variable needs to be maintained for further calculations. The classical solution to this is the introduction of local variables in the functions. Since functions do not allow for this in their syntax, Mathematica provides additional functions to define local variables explicitly and then execute a block of code. **Module** is one such function, where the first argument is a list of local variables and the second is a block of statements.

```
Swap [x_, y_]:=Module[{t}, t=x; x=y; y=t, {x, y}]
```

In this example, the variable `t` is a temporary local variable. After swapping the values of the parameters they are expressed as a list, since the last expression represents the return value of the function.

Modularization - Files

Instead of typing in an entire program from the command line, the program can be stored in a Notebook and recalled when needed. Notebooks are especially geared towards storing input, output and additional formatting. In order to store the definition of a function, or a sequence of Mathematica commands, it is not necessary to use a Notebook.

Any text file containing Mathematica code can be used as input to the interpreter. The `Get` function opens the file, reads in the data and executes each line of the file in sequence. The result of the `Get` function is the result of the last expression evaluated. This is the easiest method of storing and retrieving Mathematica programs.

A package is a collection of function definitions stored in a text file. It differs from a normal text input file in that there is the addition of scope mechanisms. Instead of making all variables globally accessible as before, a package can hide its variables and definitions from the rest of the environment. This is accomplished by Mathematica dividing the variable space into contexts. Any variable declared is inserted into the current context. When a package is loaded, it creates a new context and inserts its definitions into that context, finally switching back to the old context. That way its definitions are protected from being accidentally overwritten by new definitions. It is possible to access members of another context explicitly but this is sufficiently complex that it does not happen accidentally. Also, contexts can export their definitions so that certain functions may be used in all contexts - after loading a package the user can directly call the functions exported by that package but not its internal functions.

Simple Genetic Programming Implementation

The complete set of Mathematica files for this implementation is contained in Appendix A.

Representation of Data

Since Mathematica already stores all data internally in the form of expression trees, this can be exploited readily to represent the individuals in a GP implementation.

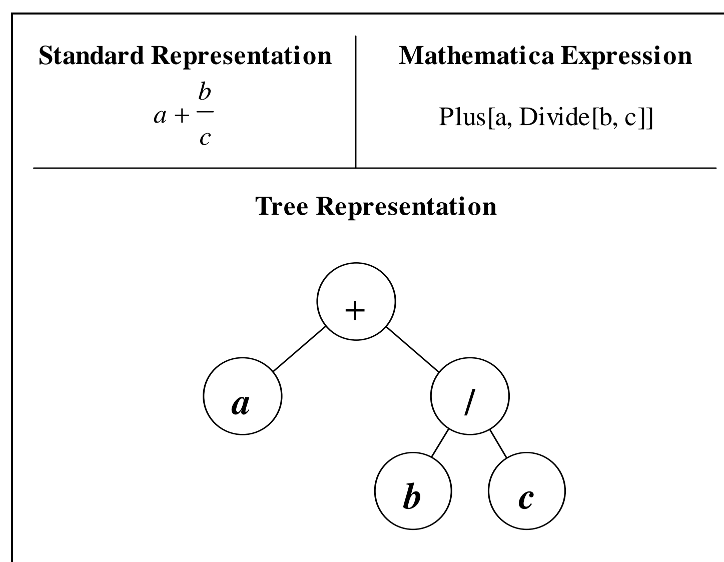

Figure 2.1. Representation of an expression

The individuals in a population could be represented simply as Mathematica expressions due to their correspondence to trees. However, Mathematica would attempt to simplify all expressions immediately. Thus any expression with constant parameters would be folded immediately to the numerical value of the constant expression. For example,

```
Plus[2, 3, 7]
```

becomes

```
12
```

This is not always desirable since genetic material would be lost each time an expression is simplified. In order to prevent Mathematica from simplifying individuals, the standard functions are replaced with dummy functions. **`Plus`** is replaced with **`PPlus`**, **`Minus`** is replaced with **`PMinus`**, etc. Since Mathematica knows nothing about the functions called **`PPlus`** and **`PMinus`**, it will not attempt to reduce the expressions. The above expression would now be

```
PPlus [2, 3, 7]
```

and Mathematica would not reduce the expression since it would not know how to do that. However, in order to use the expressions in fitness evaluations, they must be meaningful to the interpreter. At the last point before evaluation, the expressions can be converted to the proper form with a simple transformation.

```
XTrans={PPlus->Plus, PMinus->Minus, PTimes->Times,
PDivide->Divide}
```

This defines a set of rules for converting sub-expressions from one value to another. In this example, all occurrences of **`PPlus`** would be changed to **`Plus`**, and so forth. Mathematica provides a mechanism to apply this set of transformations to any expression as illustrated below.

```
PPlus [2, 3, 7] /. Xtrans

12
```

After the expression has been transformed, it is immediately evaluated by the kernel and the result is returned.

According to Koza, the first two elements to consider when modelling a GP are the function and terminal sets [Koza, 1994]. The functions can be simply the collection of dummy Mathematica functions, corresponding to real functions that may be contained in individuals. For simple polynomials, this would include the four basic operations.

```
Functions={PPlus, PTimes, PMinus, PDivide}
```

Since Mathematica has no knowledge of the functions in the function set, there is no way of telling how many parameters each can take. This is required to construct syntactically correct individuals, so it has to be specified explicitly as a list of arities.

```
Parameters={2, 2, 1, 2}
```

The terminal set would contain all the variables available to each individual. Just as with the function set, this is specific to each problem.

```
Terminals={x, y, z}
```

Closure of Function Set

Since GP can construct any expressions with any possible numerical values, it is quite conceivable that an individual may attempt to divide by zero. This can be prevented by explicitly assigning a non-error value to that operation. Mathematica allows the programmer to override any function, which includes the standard operations.

```
ClearAttributes[Divide, Protected]

Divide[_, 0]:=1

SetAttributes[Divide, Protected]
```

Every function has attributes to indicate what is possible with the function. The **`Protected`** attribute indicates that the definition of a function cannot be changed. In order to change the definition, this attribute must therefore be temporarily removed.

It is not necessary to provide a name for the first formal parameter of the definition since this parameter is never used. All that Mathematica needs check for is the zero as a second parameter - then the value "1" is returned. Since the parameter list is more specific, this clause has higher priority than the general case - the kernel will attempt to match these parameters before trying the built-in definition.

Similarly, all other functions used in the implementation must be scrutinised for undefined values. Any such values must be overridden with appropriately defined values. Besides **`Divide`**, it may be useful to overload the definitions for **`Log`** and **`Power`** as well.

```
ClearAttributes[Log, Protected]

Log[0]:=0

Log[x_ /; x<0]:=Log[-x]

Log[E^x_]:=x

SetAttributes[Log, Protected]
```

```
ClearAttributes[Power, Protected]
Power[0, -1]:=1
SetAttributes[Power, Protected]
```

Early Mathematica kernels could not automatically simplify some **`Log`** expressions so those were defined here as well. Once they are defined, these functions will be used automatically by the kernel.

Power has to be overloaded simply because 0^{-1} is equivalent to division by zero.

Fitness

The fitness of an individual can be defined as a function that takes the individual as its single parameter and returns the associated fitness value. This function is specific to the problem domain so it cannot be included in the general algorithm. However, it is possible to pre-define the transformations that the fitness value undergoes.

The raw fitness is a raw indication of the fitness of the individual. The standardised fitness is the zero-based fitness, such that a fitness of zero represents the perfect solution. The adjusted fitness maps the standardised fitness onto the range 0-1 such that 1 is the best fitness and 0 the worst.

```
(* RawFitness *)
StandardizedFitness[x_]:=RawFitness[x]
AdjustedFitness[x_]:=N[1/(1+StandardizedFitness[x])]
```

`RawFitness` is enclosed within comment delimiters since it is defined differently for each problem domain. **`AdjustedFitness`** returns its result in numerical format by applying the numerical approximation function **`N`**. This forces the kernel to convert all fractions to real numbers, which is necessary for the fitness-proportionate reproduction stage.

Parameters

These parameters control the GP execution. They are used in conjunction with the fitness and function/terminal sets to uniquely define the GP approach to finding a solution in a particular problem domain.

```
MaxGenerations = 51
```

`MaxGenerations` is the maximum number of generations that must be created by the algorithm. If no acceptable solution is found after **`MaxGenerations`** generations, then the algorithm terminates.

```
PopulationSize = 250
```

`PopulationSize` is the number of individuals in a single generation of the population. This is a static number to prevent the population from outgrowing the computer's resources or dwindling to obscurity.

```
MaxInitialSize = 6
```

`MaxInitialSize` is the maximum initial depth of the trees in generation 0.

```
MaxSize = 17
```

`MaxSize` is the maximum depth of the trees. This is different from **`MaxInitialSize`** since it is expected that better trees in later generations will be larger than the initial ones.

```
MaxComplexity = 50
```

`MaxComplexity` is the maximum number of nodes that a tree can have. This is necessary to prevent bushy trees, which correspond to complex expressions. In effect, this parameter controls the parsimony of the generated solutions. A smaller value generates more parsimonious individuals but may miss the solution altogether. A larger value generates complex expressions but has a better chance of finding solutions.

```
CrossoverProbability = 0.9
```

`CrossoverProbability` is the probability that crossover will occur between a pair of individuals during the creation of a new generation. It is expressed as a fraction relative to 1, thus 0.9 represents a 90% probability of crossover.

```
MutationProbability = 0.1
```

`MutationProbability` is the probability that an individual will be mutated during the creation of a new generation. 0.1 represents a 10% probability of mutation.

```
MinFitness = 0.99
```

`MinFitness` is the minimum fitness value that indicates termination of the algorithm. If any individual achieves a fitness equal to or better than this, then that is denoted the solution and the algorithm stops iterating.

Generation of Random Population

```
GenerateNormal[d_]:=
   Module[
     {r, Poss, PossPar},
     If[
        d>1,
        Poss=Join[Functions, Terminals];
        PossPar=Parameters,
        Poss=Terminals;
        PossPar={}
       ];
     While[
           Length[PossPar]<Length[Poss],
           PossPar=Append[PossPar,0]
          ];
     r=Random[Integer, {1, Length[Poss]}];
     Switch[
            PossPar[[r]],
            0,
            Poss[[r]],
            1,
            Poss[[r]][Generate[d-1]],
            2,
            Poss[[r]][Generate[d-1], Generate[d-1]]
           ]
   ]
```

`GenerateNormal` recursively generates a random expression tree. It takes a single parameter being the depth of the tree and then produces a tree of at most this depth, composed entirely of functions and terminals from the pre-specified sets.

The first statement checks if the depth is greater than one. If so, it allows the generation of functions as well as terminals. If the depth is exactly one, then only

terminals are allowed. If terminals and non-terminals are acceptable, then they are joined together into one list. In either case, the number of parameters associated with terminals needs to be set to zero for each terminal.

After this is done, a random number (between 1 and the number of possible functions/terminals) is generated to decide on the sub-expression to be generated at that point. The number of parameters for this function is extracted from the **PossPar** list, built in the previous lines, and used to recursively generate expressions for each parameter. The output of the **Switch** function is what is returned by the function so each possible output is formed by a function/terminal followed by a set of parameters. These parameters are generated using the same **GenerateNormal** function, except that the maximum depth is reduced by one for each parameter.

Reproduction

A set of functions works together to create a new population from the previous generation, using fitness-proportionate selection.

```
(* List of fitnesses of expressions in current generation *)
Fitnesses={}
```

Fitnesses is a list of the fitnesses of all individuals in the population. These are calculated whenever a new generation has been created, after all the genetic operators have been applied. The list of fitness values are necessary to implement roulette-wheel selection.

```
(* Make cumulative fitnesses vector *)
CalcFitnessSum:=
   Module[{},
          FitSum=Table[Apply[Plus, Take[Fitnesses, i]],
                       {i, 1, Length[Fitnesses]}
                      ];
          FitSum=Insert[FitSum, 0, 1];
         ]
```

CalcFitnessSum creates a list of partial sums of the fitnesses of individuals. For example, if the fitnesses of a 5-individual population corresponds to {1, 2, 3, 4, 5}, then the value of **FitSum** would be {1, 3, 6, 10, 15}. Each element of **FitSum** is the sum of all fitnesses up to that point. Finally an element with value “1” is inserted at the beginning of the **FitSum** list to assist with the search technique employed below.

```
(* Bisection algorithm search for roulette wheel fitnesses *)
Search[x_] :=
  Module[{Mid, Start=1, Stop=Length[FitSum]},
         While[Start+1 != Stop,
               Mid = Floor[(Start+Stop)/2];
               If[FitSum[[Mid]] > x,
                  Stop=Mid,
                  Start=Mid
                 ]
              ];
         Start
        ]
```

In order to implement roulette-wheel selection of individuals, the normal procedure is to add together all fitnesses, generate a random number in the range of this sum and then add fitnesses until the random number is exceeded. The bottleneck in such a mechanism lies in the linear search through the list of fitnesses that must be done to find the selected individual. Freeman modified this technique when applying it to GAs, by producing partial sums and executing a binary search for the selected individual [Freeman, 1994].

The partial sums, as created by the **CalcFitnessSum** function, are obviously sorted in ascending order. A binary search applied to this **FitSum** list produces exactly the same results as the linear search technique applied on **Fitnesses**.

```
(* Create new generation from previous one *)
NewGen[x_] := Module[
  {maxwheel, newgen, lenx},
  newgen={};
  maxwheel=Apply[Plus, Fitnesses];
  lenx=Length[x];
  CalcFitnessSum;
  Do[
    Module[
      {spot, index, isum},
      spot=Random[]*maxwheel;
      index=Search[spot];
      newgen=Append[newgen, x[[index]]]
    ],
    {i, 1, lenx}
  ];
  newgen
]
```

NewGen creates a new generation of individuals. The **newgen** is first initialised to an empty list. The sum of fitnesses (**maxwheel**) and the size of the population (**lenx**) are calculated. It can be argued that the **PopulationSize** can be used. However, by generating the population size dynamically, it is possible to apply this function to subsets of the population as well.

CalcFitnessSum creates the list of partial sums needed for the binary search. A new generation is then created iteratively. A random number is generated and the associated individual is selected by the **Search** function. The individual is then appended to the new generation in **newgen**.

Finally, the value of **newgen** is returned as the result of the function, being the new population.

Crossover

Two child expressions are produced from a pair of parents by means of the crossover genetic operator. **Cross1** takes two individuals and performs crossover. **Crossover** applies this function to an entire population.

```
(* Get list of all indices of internal points in expression *)
RemoveZero[x_]:=If[Position[x, 0]=={}, x, {}]
Points[x_]:=Union[Map[RemoveZero, Position[x, _]], {}]
GetInternal[{x___}]:=x
```

The unique position of any node or subtree in a tree can be specified by a list of indices, which represent the path from the root to the node. **Points** is a function which generates a list of the positions of every subtree of a given tree.

```
(* Perform crossover operation on two expressions *)
Cross1[x_, y_]:=
   Module[
          {spot1, spot2, point1, point2, temp1, temp2},
          If[
             Random[]<CrossoverProbability,
             point1=Points[x];
             spot1=Random[Integer, {1, Length[point1]}];
             point2=Points[y];
             spot2=Random[Integer, {1, Length[point2]}];
             temp1=x[[GetInternal[point1[[spot1]]]]];
             temp2=y[[GetInternal[point2[[spot2]]]]];
             { If[
                  point1[[spot1]]=={},
                  temp2,
                  ReplacePart[x, temp2, point1[[spot1]]]
                 ],
               If[
                  point2[[spot2]]=={},
                  temp1,
                  ReplacePart[y, temp1, point2[[spot2]]]
                 ]
             },
             {x, y}
            ]
         ]
```

Cross1 crosses over two individuals to produce a pair of new individuals. First, a random number is generated and this is used to decide whether to apply crossover or simply copy the individuals.

If crossover is to be applied, the node list is generated for each individual by calling **Points**. Random sub-trees are extracted from the individuals and then stored in the **temp1** and **temp2** variables. Finally, the sub-trees are swapped and replaced in the individuals and the list of two new individuals is returned from the function. The additional check before replacing the sub-tree handles the special case where the sub-tree is the entire individual.

```
(* Perform crossover on all expressions in new generation *)
Crossover[x_] := Module[
  {newx, oldx, n2, leno, origlen},
  oldx=x;
  newx={};
  leno=Length[oldx];
  origlen=leno;
  While[
    leno>0,
    If[
      leno==1,
      newx=Append[newx, First[oldx]];
      oldx=Rest[oldx],
      n2=Cross1[oldx[[1]], oldx[[2]]];
      If[((Depth[n2[[1]]]<=MaxSize) &&
          (LeafCount[n2[[1]]]<=MaxComplexity)),
          newx=Append[newx, n2[[1]]],
          newx=Append[newx, oldx[[1]]]
         ];
      If[((Depth[n2[[2]]]<=MaxSize) &&
          (LeafCount[n2[[2]]]<=MaxComplexity)),
          newx=Append[newx, n2[[2]]],
          newx=Append[newx, oldx[[2]]]
         ];
      oldx=Drop[oldx, 2];
    ];
    leno=Length[oldx]
  ];
  newx
]
```

Crossover applies the **Cross1** function to an entire population. Once again, the new generation (**newx**) is initialised to an empty set and the length of the population is calculated (**leno**). The first two elements of the old population are crossed. The new individuals are separately tested to make sure that they do not exceed the maximum size or complexity parameters. Each individual that passes the test is added to the new population, while those that fail the test are discarded and the original

individuals are then added to the new population. Finally, the first two individuals are removed from the list, and the process continues as before. If there is only one individual left in the population, that is simply copied to the new population. The iteration terminates when the entire old population has been processed.

Mutation

`Mutation` is a function that applies the mutation genetic operator to an individual.

```
Mutate[x_]:=Module[
                      {spot1, point1, y, xold},
                      xold=x;
                      If[
                         Random[]<MutationProbability,
                         y=Generate[MaxInitialSize];
                         point1=Points[x];
                         spot1=Random[Integer, {1,Length[point1]}];
                         If[
                            point1[[spot1]]=={},
                            y,
                            ReplacePart[x, y, point1[[spot1]]]
                           ],
                         If[
                            ((Depth[x]<MaxSize) &&
                            (LeafCount[x]<MaxComplexity)),
                            x,
                            xold
                           ]
                        ]
                     ]
```

Before modifying the individual in any way, a copy is kept in **`xold`**. Then a random number is generated to decide whether to apply the mutation operator or not. If the operator is not applied, the individual is simply returned as the result.

Otherwise, a random expression is generated. Just as with crossover, a random point is chosen in the tree. The new expression is inserted at this point, replacing whatever was there before. During this replacement, it is still important to check if the whole expression needs replacing. Finally, before returning the new individual, it is necessary to check that it does not exceed the complexity or size requirements.

Result Designation

The best individual from all the generations is designated as the solution, if it satisfies the **`MinFitness`** criterion. In order to keep track of this solution, it is necessary to

store the individual as well as its fitness. **CheckSolution** checks the population at each iteration to determine if an acceptable solution has been found.

```
(* Update best-of-run individual *)
CheckSolution[gen_, x_]:=
   Module[
          {minf, maxf},
           Fitnesses=AdjustedFitness /@ x;
           minf=Position[Fitnesses, Min[Fitnesses]][[1,1]];
           maxf=Position[Fitnesses, Max[Fitnesses]][[1,1]];
           If[
              SolutionFitness<Fitnesses[[maxf]],
              Solution=x[[maxf]];
              SolutionFitness=Fitnesses[[maxf]]
             ];
           SolutionSet=Append[
                             SolutionSet,
                             {gen, Fitnesses[[maxf]],
                              x[[maxf]],
                              Fitnesses[[minf]], x[[minf]]}
                            ];
           Print["G", gen, ": max ", Fitnesses[[maxf]],
                 "    min ", Fitnesses[[minf]]];
         ]
```

First the fitness is calculated for all individuals in the population. Then the position of the minimum and maximum fitnesses are calculated. The best solution of the current generation is checked against the global solution (**Solution**, **SolutionFitness**) and the global values are replaced if appropriate. Finally, the best and worst fitness values and their associated individuals are stored for statistical purposes (in **SolutionSet**).

Initialisation

Running the GP is a two-step process. First the population and variables must be initialised with default or initial values. Then the GP can be run until one of the termination criterion is satisfied.

```
(* Initialise Genetic algorithm *)

Initialize:=Block[{poplog},
                  Population=Table[Generate[MaxInitialSize],
                                   {PopulationSize}];
                  SolutionFitness=0;
                  SolutionSet={};
                  Generation=0;
                  TotTime=0;
                  Print["G", Generation, ": calculating",
                        "fitnesses ..."];
                  Print["G", Generation, ": done ... ",
                        Timing[CheckSolution[Generation,
```

```
                Population]][[1]]];
            Print["G", Generation, ": best-of-run "
                "fitness so far = ",
                SolutionFitness];

            Off[DeleteFile::nffil];
            DeleteFile["pop.log"];
            On[DeleteFile::nffil];
            poplog=OpenAppend["pop.log"];
            WriteString[poplog, "pop={"];
            Write[poplog, {Generation, Fitnesses}];
            Close[poplog];

            Information[Population];
            GInformation;
           ]
```

First an initial generation 0 population is created. All global variables are given their initial values. **`SolutionFitness`** is set to the absolute minimum fitness (0) so that the very first time **`CheckSolution`** is run, it would attach a value to this variable. **`Generation`** is set to 0, being the initial generation, and **`SolutionSet`** is empty since no generations have been processed yet. Then the initial generation is checked by **`CheckSolution`** and the results displayed on the screen.

POP.LOG stores statistical information used to monitor the distribution of individuals in the population. It is deleted and then initialised with the data for the initial generation.

Finally, the individuals in the initial population are displayed on the screen, together with information about the parameters of the impending execution.

ApplyGen

The GP algorithm itself is controlled solely by the **`ApplyGen`** function.

```
(* Apply Genetic algorithm *)

ApplyGen := Module[
  {onetime, poplog},
  newpop=Population;

  While
   [
    (SolutionFitness<MinFitness) && (Generation<MaxGenerations),
   onetime=Timing[
    Print["G", Generation, ": creating mating pool ..."];
    Print["G", Generation, ": done ... ",
          Timing[newpop=NewGen[newpop]][[1]]];
    Print["G", Generation, ": performing crossover ..."];
```

```
    Print["G", Generation, ": done ... ",
          Timing[newpop=Crossover[newpop]][[1]]];
    Print["G", Generation, ": performing mutation ..."];
    Print["G", Generation, ": done ... ",
          Timing[newpop=Map[Mutate, newpop]][[1]]];
    Generation++;
    Population=newpop;
    Print["G", Generation, ": calculating fitnesses ..."];
    Print["G", Generation, ": done ... ",
          Timing[CheckSolution[Generation, newpop]][[1]]];
    Print["G", Generation, ": best-of-run fitness so far = ",
          SolutionFitness];
   ][[1]];
    Time[onetime, "G", Generation,
         ": total time for Generation change = "];
    TotTime+=onetime;
    Time[TotTime, "G", Generation, ": total time so far = "];

    poplog=OpenAppend["pop.log"];
    WriteString[poplog, ","];
    Write[poplog, {Generation, Fitnesses}];
    Close[poplog];
   ];
  {Solution /. XTrans, SolutionFitness}
]
```

The iteration proceeds as long as the current best solution does not exceed **MinFitness** and the maximum number of generation has not been reached. A new generation is created by fitness-proportionate reproduction using the **NewGen** function. **Crossover** and **Mutation** are applied to this new generation and it then replaces the original population. Finally, **CheckSolution** checks the fitnesses of individuals. Throughout the iteration, the time taken is measured and extensive reporting on current activity is carried out. At the end of the iteration, this time is reported as well as the time taken for all generations thus far. The population fitness data is saved in POP.LOG for statistical purposes and the next iteration begins.

```
(* Start run of algorithm *)
StartGen:=Timing[
                 CheckAbort[
                            ApplyGen,
                            {Solution /. XTrans,
                             SolutionFitness}
                           ]
                ]

ContinueGen[gen_]:=Module[{},
                          MaxGenerations=gen;
                          MinFitness=2;
                          StartGen
                         ]
```

StartGen and **ContinueGen** simply enhance the capabilities of **ApplyGen**. **StartGen** incorporates the ability to break out of the calculation as well as displaying timing information and the solution at the end of the run. **ContinueGen** continues the algorithm after the termination criterion has been met, in an attempt to find even better solutions or alternatives.

Automatic Recovery

Although the GP algorithm works fairly well if left to run unattended, it takes extremely long to find non-trivial results. If a computer is working on a problem for a long period of time, it is quite possible that there could be a power failure. In such cases, all intermediate calculations would be wasted and the algorithm would have to be started from scratch.

To save these results, the state of the system at each stage of the calculation can be stored in a text file by the following code fragment, for easy continuation at a later stage:

```
Save["restart.log", PopulationSize];
Save["restart.log", ContinueGen];
```

Mathematica saves the definition of **PopulationSize** and **ContinueGen** in the file called RESTART.LOG. However, since **ContinueGen** calls **ApplyGen**, that is also saved. All the functions called by **ApplyGen** are saved as well and this process continues recursively. Eventually, every function needed to execute the GP is stored in the file.

There is always the danger, albeit quite small, that the power failure may occur while the backup is taking place. The solution to this is to make the backup in a temporary file and only swap the files once the backup is complete. Using this technique, in the worst case scenario where the power failure occurs during backup, the previous backup is still secure and can be used.

```
Print["Saving state of system..."];
Save["restart.log", PopulationSize];
Save["restart.new", ContinueGen];
RenameFile["restart.log", "restart.old"];
RenameFile["restart.new", "restart.log"];
DeleteFile["restart.old"];
```

This saving of data must be incorporated into both **`ApplyGen`** and **`Initialize`**. In order to use this data, **`ApplyGen`** must load the data from disk before going into the processing loop. This can be accomplished simply by

```
Get["restart.log"];
```

Mechanics of a Sample Implementation

In order to use this GP implementation, the programmer must first model the problem domain in Mathematica. Then appropriate terminal and function sets must be chosen along with a reasonably well-scaled fitness function. Parameters can be tweaked to accommodate peculiarities of the problem domain; for example, a larger population size may be needed if the function is larger.

The initial population is generated and processed by calling

```
Initialise
```

The GP algorithm is begun by calling

```
StartGen
```

Thereafter the progress of the algorithm can be monitored on the screen.

After a successful GP run, it is possible to utilise the built-in features of Mathematica to analyse the results, produce statistics and generate graphs and histograms.

CHAPTER 3 : SYMBOLIC REGRESSION

Statistical Analysis Techniques

A series of experiments was conducted to evaluate the effectiveness of the implementation. These experiments were compared on the basis of time taken, resources used, and the changes in the population as the generations progressed, the most important changes being those in the fitness values. These fitness values were streamed, in Mathematica expression format, to a text file during each run of the GP algorithm.

After the algorithm terminated, it was possible to read in the complete list of fitnesses over all generations and extract information regarding the convergence, divergence or other shifts in the population. This data could then be displayed graphically using the built-in graph-plotting routines in Mathematica.

```
ShowCurve:=Module[
                    {t},
                    t=MapThread[List, SolutionSet];
                    ListPlot[MapThread[List, {Join[t[[1]],
                                         t[[1]]],
                                         Join[t[[2]], t[[4]]]}],
                                         PlotRange->{{0, 51},
                                                     {0,1}}]
                   ]
```

`ShowCurve` displays a graph of the minimum and maximum fitnesses of each generation. This function is general and can be applied to all problem domains. A typical output from **`ShowCurve`** is shown in *Figure 3.1*. This graph indicates whether the algorithm is convergent or not. If there is visible convergence and no solution has yet been found, then the algorithm can be extended over more generations. If convergence is not reached, then the parameters of the run can be tweaked to better suit the problem domain.

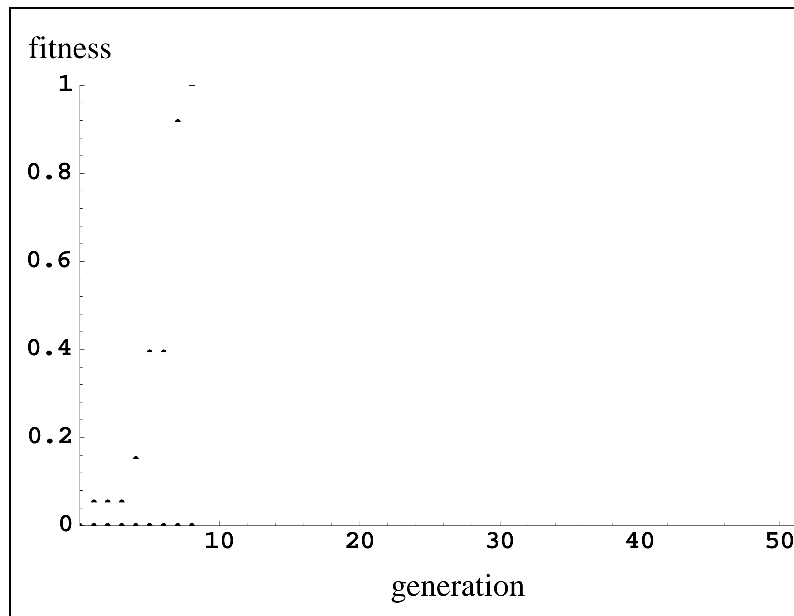

Figure 3.1. Best and worst fitnesses per generation

In *Figure 3.1*, the x-axis represents the generations and the y-axis represents the fitnesses of the best and worst individuals.

Just watching the fitnesses of best and worst individuals may not be enough. If the best individual of the run is found in generation 0, then the graph from **ShowCurve** may indicate only a horizontal line. However, the fitnesses of other individuals may have changed drastically, making it necessary to visualise the entire population instead of just the extremities. For any given generation, every individual's fitness can be plotted on a graph to display the distribution of fitness values. This introduces new difficulties since the size of the population dictates the amount of information that needs to be contained in the graph. One approach employed throughout this study is to divide the fitness value range into discrete intervals. Then the individuals can be split into sub-ranges according to their fitnesses. A histogram of fitness values can be generated from these discrete ranges. Separate histograms can be created for each generation and animated (using built-in Mathematica functions) to display the implicit movement of the population towards a greater average fitness.

```
Run["copy pop.log+pop.m pop.ful /Y > nul"]
<<pop.ful
popfit=MapThread[List, pop][[2]]
```

The first few lines of the histogram generation routines convert the raw data from the previous run into a list, containing lists of fitness values for each generation.

```
Histogram[x_, opts___]:=
   Module[{data, fl, figs},
          data=Table[0, {10}];
          figs=Map[Floor, popfit[[x+1]]*10];
          figs=Map[If[#==0, 1, #]&, figs];
          Map[(data[[#]]++)&, figs];
          BarChart[data, BarLabels->Table[i, {i, 0, 0.9, 0.1}],
                   PlotRange->{{0, 11}, {0, PopulationSize}},
                   PlotLabel->StringJoin["Generation ",
                                         ToString[x]],
                   opts]
         ]
```

`Histogram` generates a histogram from the fitness data for a single population. The fitness values are divided into 10 discrete ranges, each with length 0.1. The x-axis represents the fitness ranges and the y-axis represents the number of individuals in each category. A typical output from **`Histogram`** is shown in *Figure 3.2*.

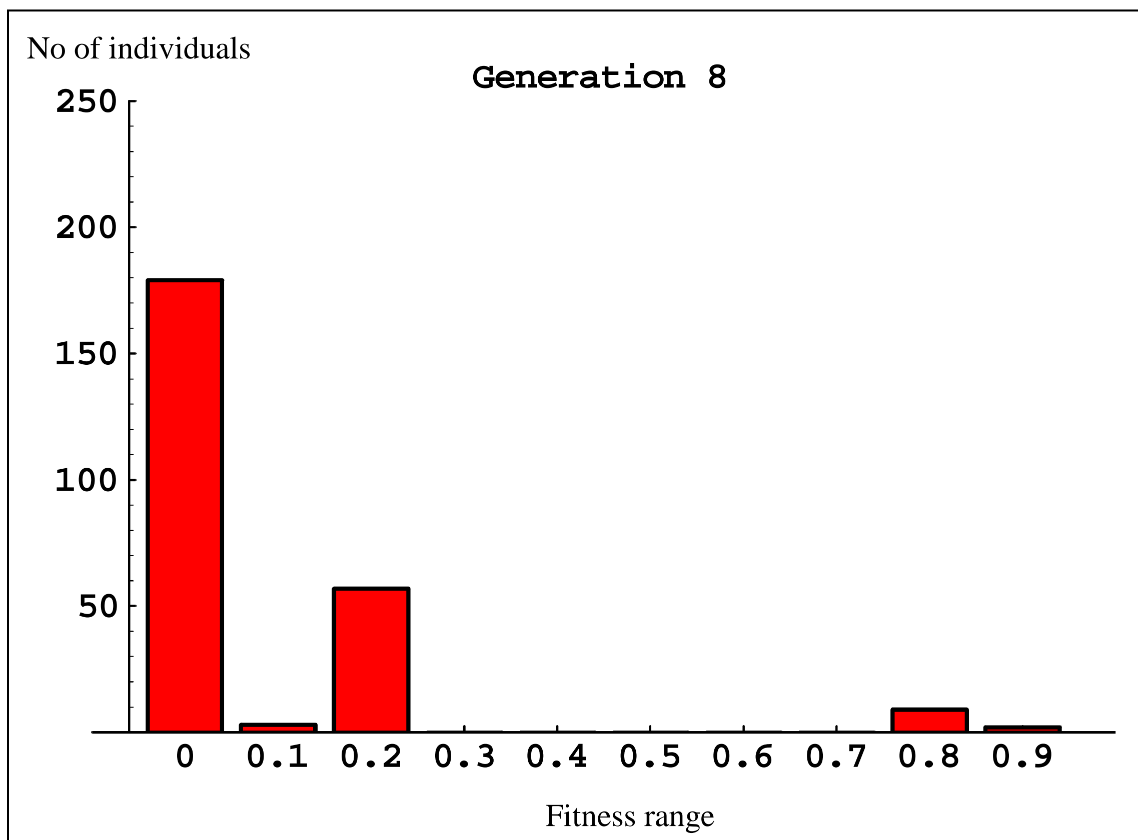

Figure 3.2. Fitness histogram for generation 8

```
HistTable:=Table[Histogram[i, DisplayFunction->Identity],
                 {i, 0, Length[pop]-1}]
```

`HistTable` creates fitness histograms for all populations, without displaying them on the screen - they are simply created and stored in memory.

```
AnimateHist:=ShowAnimation[HistTable];
```

`AnimateHist` displays an animation of the fitness histograms as created by `HistTable`. This can be used to study changes in the overall fitness of the population as generations progress.

Experiment 1: Symbolic Regression in Mathematica

Problem Selection

Regression is essentially the problem of fitting an equation through a set of sample points. Statisticians use various techniques to perform different types of regression on test data. However, in almost all cases the form of the equation needs to be pre-specified. For example, in the case of linear regression, it is attempted to find the equation of a straight line that passes through the points. Knowing that a straight line equation has the format

$$y = ax + b \quad \text{...} \quad (3.1)$$

it is only necessary to find the values of the coefficients a and b. In quadratic regression, coefficients a, b and c need to be found in the following equation:

$$y = ax^2 + bx + c \quad \text{..} \quad (3.2)$$

This is not always possible since the test data may be noisy, in which case the search is for an equation that produces the least overall error.

All regression techniques are calculation-intensive and try to find a solution by minimising the error between the prospective solution and the test data. If the form of the equation is unknown, then various forms are tried and the one with the least error

is assumed to be the solution. This selection process is largely intuitive and becomes more difficult as the complexity of the required equation increases.

Symbolic regression is an attempt to solve this problem, by searching for both the form and the coefficients of the equation. This is not easily accomplished by normal analytical and statistical techniques. A complete expression is sought and that is precisely what GP produces. This makes GP an ideal vehicle to implement symbolic regression. If the evolution of the expressions is directed by the error between the actual data and that generated from the expressions, then the expressions will gradually tend towards better-fitting equations.

Test Data

In selecting a test problem to apply GP to, it has to be decided whether to use real data or simulated data. Since the aim of this experiment was to test the operation of the algorithm, data was simulated. The data was a set of 2-dimensional coordinates in the x-y plane.

x	*y*	*x*	*y*	*x*	*y*
-2.0	10	-0.6	-0.3264	0.8	2.3616
-1.8	6.1056	-0.4	-.2784	1.0	4.0000
-1.6	3.4176	-0.2	0.1664	1.2	6.4416
-1.4	1.6576	0	-2.7756x10^{-16}	1.4	9.9456
-1.2	0.5856	0.2	0.2496	1.6	14.8096
-1.0	4.4409x10^{-16}	0.4	0.6496	1.8	21.3696
-0.8	-0.2624	0.6	1.3056	2.0	30.0000

Table 3.1. 21 pairs of x-y coordinates used as test data in Experiments 1.4-1.7

The generation of test points as shown in *Table 3.1* can be either random or derived from some known equation. With random data a solution is not guaranteed so it was decided to use latter approach. The data in *Table 3.1* was generated by selecting equidistant points along the x-axis and determining corresponding y-values from the given equation (*Equation 3.3*). This set of test data was used in Experiments 1.4-1.7.

Experiments 1.1-1.3 generated y-coordinates from uniformly random non-equidistant points along the x-axis in the range [-1, 1]. New sample points were generated for each run of the experiment. These sample points are shown graphically in *Figures 3.3, 3.5 and 3.7*.

Experiment 1.8 added 20% random noise to the sample data indicated in *Table 3.1*. This is further elaborated upon in the discussion of that experiment.

The advantage of equidistant x-coordinates is that the equations generalise better to points in between those given. In the case of non-equidistant x-coordinates, the points may be clustered, and there would exist gaps between the clusters that are larger than the average gap size. These large gaps can result in unnecessary fluctuations in the equations, since there are no points to constrain the path of the curve.

Rather than generate random data, all the subsequent experiments used an equation, that was known to converge in a reasonable amount of time, to generate test cases [Koza, 1992].

$$y = x^4 + x^3 + x^2 + x \quad \text{...} \quad (3.3)$$

During the course of the experiment, it became clear that *Equation 3.3* has some useful properties that are not found in other equations (e.g. $y=x^4+1$, $y=x^3+x+1$). Firstly, the points were rarely fitted by any other equation, thus preventing convergence to a local minimum. Secondly, the equation can be factored in a multitude of different ways. Thus there are many different parse trees or representations of the equation, which means that the solution occupies a larger portion of the search space; hence it can be found more easily. When other equations were substituted, GP did not converge to a solution since the population size was no longer large enough. It was decided to run all tests using *Equation 3.3* so that large populations would not be necessary.

Platform

All experiments were run on a 486 DX2-66 machine with 16 megabytes of RAM, under Mathematica for MS-DOS version 2.2.

Statistics

Additional statistics, specific to this problem, were produced for each run of the experiment.

```
ShowSample:=ListPlot[MapThread[List, {XPoints, YPoints}]]
```

`ShowSample` displays the test data in graphical format.

```
ShowSolution:=Plot[Solution /. XTrans, {x, -2, 2}]
```

`ShowSolution` plots the equation generated by the GP.

```
ShowFit:=Show[ShowSample, ShowSolution,
              PlotRange->{{-2, 2}, {-2, 10}},
              PlotLabel->Solution /. XTrans, AxesLabel->{x, ""},
              Frame->True
             ]
```

`ShowFit` superimposes the graphs from **`ShowSample`** and **`ShowSolution`** to graphically display the equation passing through the sample points. A typical graph generated by this function in shown in *Figure 3.3*.

```
Stats[s_String]:=Module[{},
                        Display[StringJoin[s, ".sam"],
                                ShowSample];
                        Display[StringJoin[s, ".sol"],
                                ShowSolution];
                        Display[StringJoin[s, ".fit"], ShowFit];
                        Display[StringJoin[s, ".scu"],
                                ShowCurve];
                       ]
```

`Stats` produces all the graphs relevant to the problem and stores them on disk for future reference.

Problem Representation and Parameters

The parameters used during the initial experiments (1.1-1.3) are indicated in *Table 3.2*.

Parameter	Value
Population Size	**250**
Max no of Generations	**51**
Max initial size	**5**
Max size	**17**
Maximum complexity	**50**
Min solution fitness	**0.95**
Mutation probability	**0.05**
Crossover probability	**0.9**
Terminal set	**{x}**
Function set	**{PPlus, PMinus, PTimes, PDivide, PExp, PLog}**

Table 3.2. GP Parameters for symbolic regression - Exp 1.1-1.3

Most of the parameters assume default values. The rest of this section discusses those parameters that have been over-ridden as well as those parameters that are specific to symbolic regression.

```
MutationProbability=0.05
```

`MutationProbability` is set to a low value because the terminal and function sets are not large so loss of genetic material should not be a problem.

```
XTrans={PPlus->Plus, PMinus->Minus, PTimes->Times,
        PDivide->Divide, PLog->Log, PExp->Exp}
```

`XTrans` defines the transformations for all functions, whether they are used in the actual function set or not.

```
Functions={PPlus, PMinus, PTimes, PDivide, PExp, PLog}
```

The function set is defined to contain the basic operators as well as logarithms and exponents since the form of the equation is supposedly unknown. It is also of interest to determine if another totally different equation can fit the exact same points.

```
Parameters={2, 1, 2, 2, 1, 1}
```

`Parameters` define the arity of each corresponding function in the function set.

```
Terminals={x}
```

The terminal set contains only a single variable since the expression sought is a function of one variable. Constants are excluded to further shrink the solution space.

```
f[x_]:=x^4 + x^3 + x^2 + x
```

`f[x]` represents the perfect solution, used to generate the test data. Beyond this, it is not again used during the course of the experiment.

```
NoOfSamples=20
```

`NoOfSamples` is the number of points that are used as test data.

```
InitSample:=Block[{},
                    XPoints=Table[(Random[]*2)-1, {NoOfSamples}];
                    YPoints=Map[f, XPoints];
                    ShowSample
                   ]
```

`InitSample` creates the test data from the given equation. The x-values are either random distributed (Experiments 1.1-1.3) or equidistant (Experiments 1.4-1.7) and the y-values are generated from the given function **`f`.**

```
Calc[a_, xvalue_]:=a /. XTrans /. x->xvalue
```

`Calc` returns the y-value calculated from an individual expression and a single x-value, after transforming the function names.

```
RFitness[x_]:=N[Apply[Plus, Abs[( (Calc[x, #1])& /@ XPoints)-
YPoints]]]
RawFitness[expr_]:=Check[RFitness[expr], 20000]
```

`RFitness` calculates the raw fitness of an individual. The expression is used to generate a set of new y-values from the given x-values. These are then compared to the original y-values and the absolute sum of the errors represents the fitness. **`RawFitness`** traps computational errors like overflow and returns sufficiently a high fitness value so that that expression is penalised.

Experiment 1.1

The range for the x-values in this experiment was [-1, 1]. The experiment was stopped after 51 generations. The best individual found was the expression

$$e^{x} + \frac{x^{2}}{e^{-x} + e^{x^{2}} - x - x^{2}} \quad \text{..} \quad (3.4)$$

This expression had a fitness of approximately 0.476, which was far from the expected fitness. However, the expression fitted the sample data quite reasonably. Examination of the sample data and the solution curve indicated that the sample data was not evenly spaced, which may have led to the complexity of the solution.

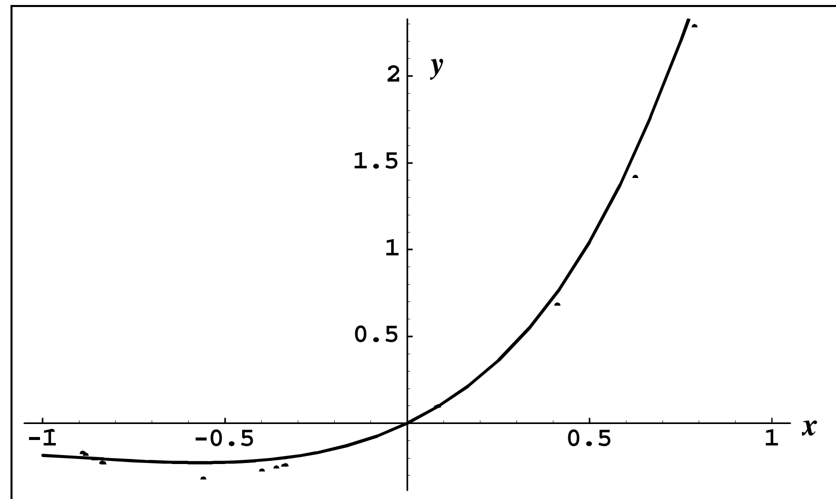

Figure 3.3. Fitting of solution to sample points - Exp 1.1

Figure 3.3 shows how closely *Equation 3.4* fits the sample points (the dots represent the sample data while the curve represents *Equation 3.4)*. However, it is noted that the fit is not perfect; the parameters can potentially be further tweaked to generate a better solution.

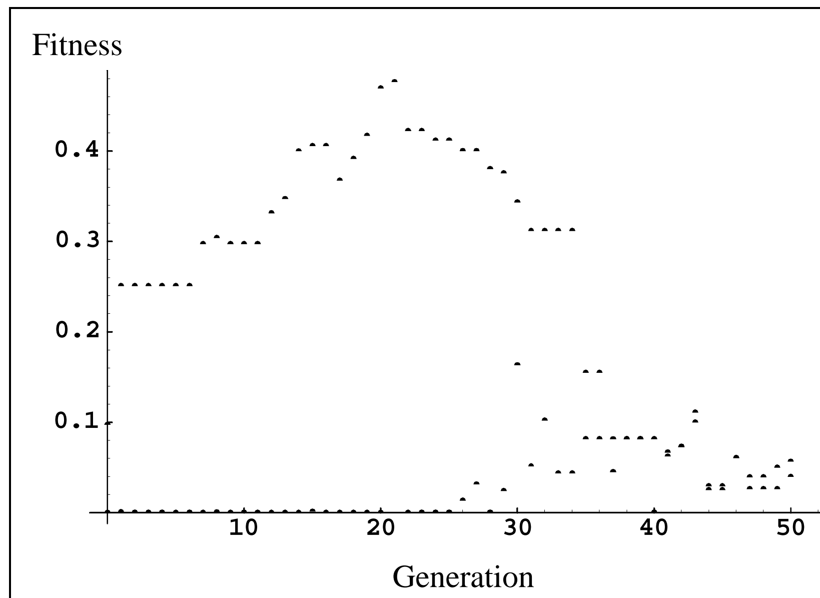

Figure 3.4. Maximum/minimum fitness curve - Exp 1.1

The graph of minimum and maximum fitnesses in *Figure 3.4* indicates that the highest fitness values are reached around generation 20. Thereafter the fitness values decrease rapidly. There is no promise of finding further solutions as a direct consequent of the current genetic material in the population. Since not every run of a GP is guaranteed to find a solution, it was decided to rerun the experiment, with a different initial population.

Experiment 1.2

The parameters were carried over from Experiment 1.1 (*Table 3.2*). However, this time the distribution of points was slightly more uniform, which contributed to a better fit as illustrated in *Figure 3.5*.

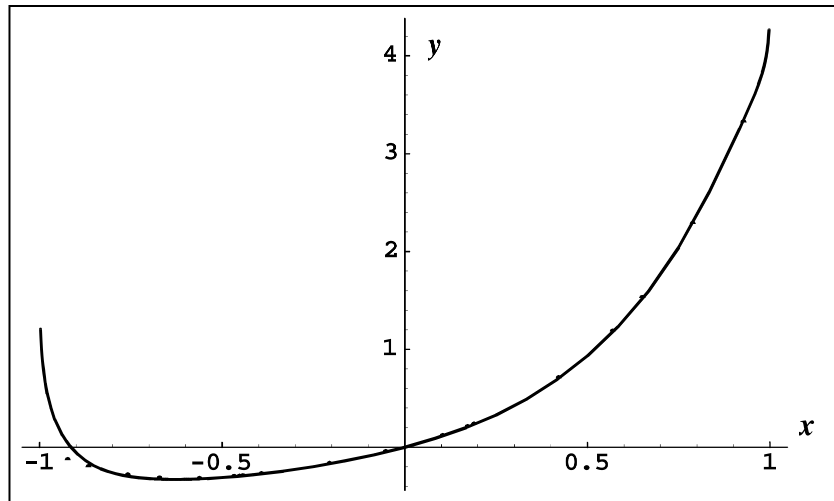

Figure 3.5. Fitting of solution to sample points - Exp 1.2

The maximum fitness reached was approximately 0.743, which was higher than the previous result. The solution expression was also more complex, as it attempted to fit almost every point precisely :

$$e^{x}x+e^{2x-e^{x}x^{4}}x^{4}(x+\log(x)\log(2e^{x^{3}}x^{3}))(x-\frac{x^{5}(\log(-x)+\log(\log(x)))}{e^{x}}) \qquad \text{.................................} \quad (3.5)$$

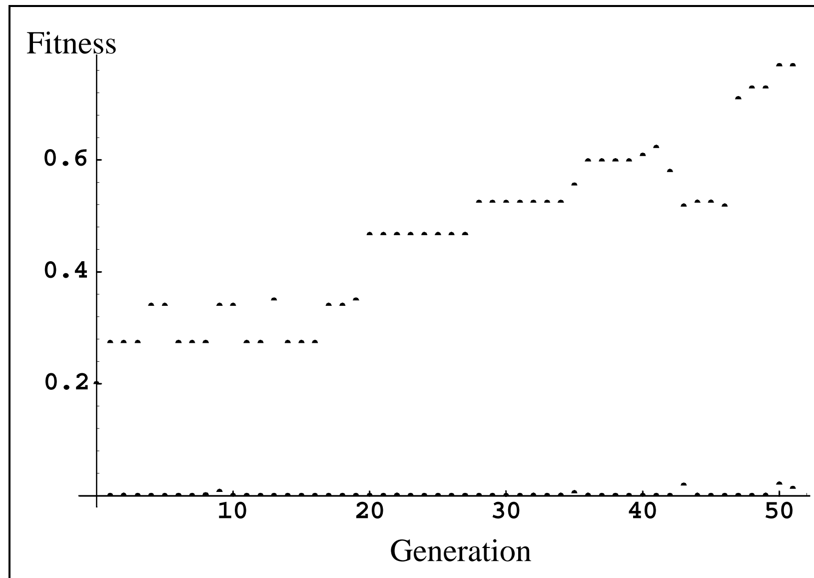

Figure 3.6. Maximum/minimum fitness curve - Exp 1.2

Figure 3.6 illustrates that the fitness of individuals was gradually increasing. This implies that further generations could find better solutions, albeit more complex ones. Although a perfect solution was not found, it appeared viable to continue along similar lines for further experiments.

Experiment 1.3

Using the same parameters (*Table 3.2*) as the previous two experiments, an even better solution was found with a fitness of approximately 0.884 :

$$x + \frac{x^5}{x + e^x x} + e^x x.\log(e^x) \quad \text{..} \quad (3.6)$$

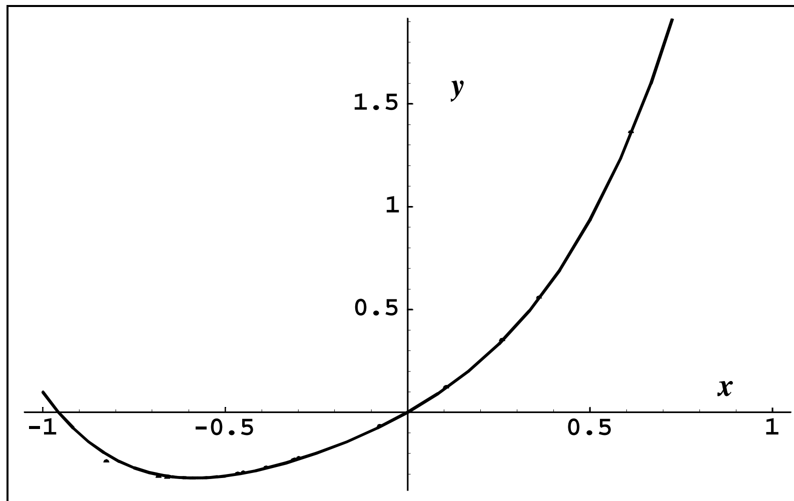

Figure 3.7. Fitting of solution to sample points - Exp 1.3

The fit of the equation to the sample data was nearly visibly perfect (*Figure 3.7*). This equation is still vastly different from the one used to generate the sample. For greater accuracy, it was decided to use a larger range of x-values in the sample data for subsequent runs.

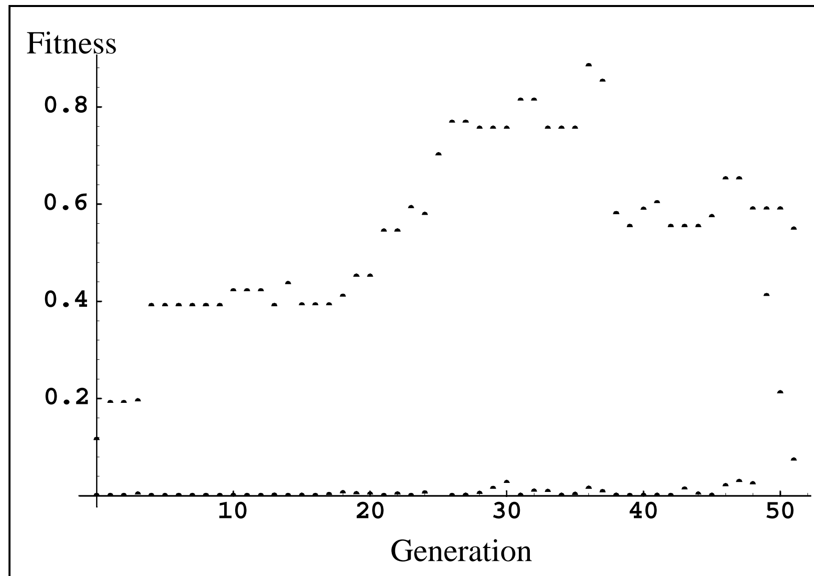

Figure 3.8. Maximum/minimum fitness values - Exp 1.3

Although the solution obtained has quite high fitness, the fitness curve (*Figure 3.8*) indicates that the fitness values are not increasing steadily. Thus further improvements would require much more computation. It was apparent that other means of finding solutions faster should be explored beyond larger populations and more generations.

Experiment 1.4

Two improvements were added into the code to speed up convergence.

Although Mathematica is an interpreted language, it allows some functions to be compiled to an intermediate format for faster execution. These functions may contain only a small subset of the standard Mathematica functions within their bodies. This subset includes the four standard arithmetic operations, making this technique applicable to the problem of symbolic regression. The definition of RawFitness was changed to incorporate compiled functions, as illustrated below.

```
RFitness[expr_]:=Apply[Plus,
                          ((Compile[{{x, _Real}},
                                     Evaluate[expr /. XTrans]
                                   ]
                          /@ XPoints)-YPoints)^2]
RawFitness[expr_]:=Check[RFitness[expr], 200000]/20
```

The operations of addition and multiplication were originally defined to take only two parameters. However, most expressions generated thus far contained sums or products of more than two sub-expressions. This is normally accomplished by a combination of two functions. It is easier to form such expressions with addition and multiplication functions of greater arity, so these were added to the function set. Addition and multiplication functions with arity 4 resulted in much too complex expressions being formed, but arity 3 functions sped up the evolution.

The range of x-values was broadened to [-2, 2] so that evolved expressions would be a better fit to the original function. Also, the sample data was generated from equidistant x-values as indicated in *Table 3.1*. The parameters for this experiment are indicated in *Table 3.3*.

Parameter	Value
Population Size	**250**
Max no of Generations	**51**
Max initial size	**5**
Max size	**17**
Maximum complexity	**50**
Min solution fitness	**0.95**
Mutation probability	**0.05**
Crossover probability	**0.9**
Terminal set	**{x}**
Function set	**{PPlus, PPlus, PMinus, PTimes, PTimes, PDivide, PExp, PLog}**

Table 3.3. GP Parameters for symbolic regression - Exp 1.4

The experiment was run three times and each run found the perfect solution, as illustrated in *Figure 3.9*.

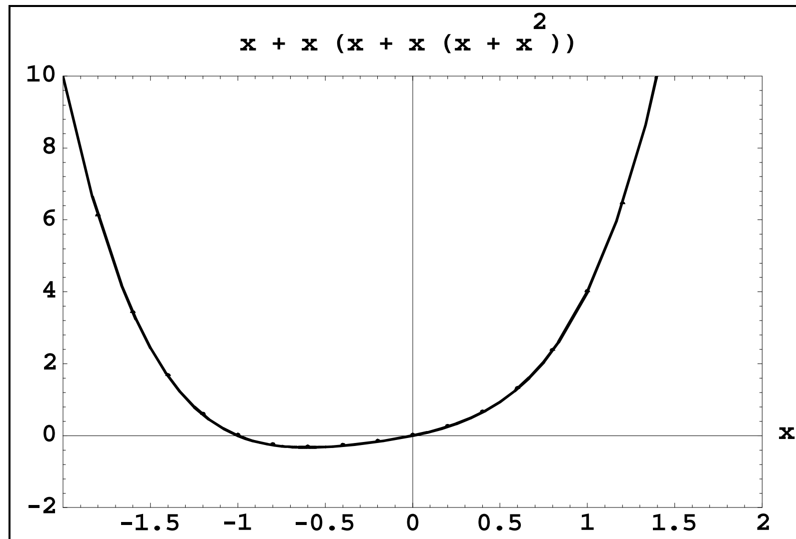

Figure 3.9. Fitting of solution to sample points - Exp 1.4 Run 1

Table 3.4 displays the times taken for each run of the experiment.

Run 1	**1 hour 26 minutes**
Run 2	**44 minutes**
Run 3	**2 hours 10 minutes**
Average	**1 hours 26 minutes**

Table 3.4. Time taken for GP runs - Exp 1.4

The fluctuations in execution times occurred because of the random nature of the GP algorithm. The initial random population might contain individuals that have high fitnesses, resulting in faster convergence, or individuals with very low fitnesses, resulting in slower convergence.

Experiment 1.5

Three runs were carried out to further test the stability of the algorithm and to generate histograms of population fitnesses as the generations progressed. In order to speed up convergence, the **`Exp`** and **`Log`** functions were removed from the function set, forcing the expressions to be strictly polynomials. The list of parameters is shown in *Table 3.5*.

Parameter	Value
Population Size	**250**
Max no of Generations	**51**
Max initial size	**5**
Max size	**17**
Maximum complexity	**50**
Min solution fitness	**0.95**
Mutation probability	**0.05**
Crossover probability	**0.9**
Terminal set	**{x}**
Function set	**{PPlus, PPlus, PMinus, PTimes, PTimes, PDivide}**

Table 3.5. GP parameters for symbolic regression - Exp 1.5

All three runs successfully found the best possible solution. In the second run the algorithm terminated because the minimum fitness criterion was reached. This minimum fitness was set at 0.95 in these experiments and changed to 0.99 for future runs.

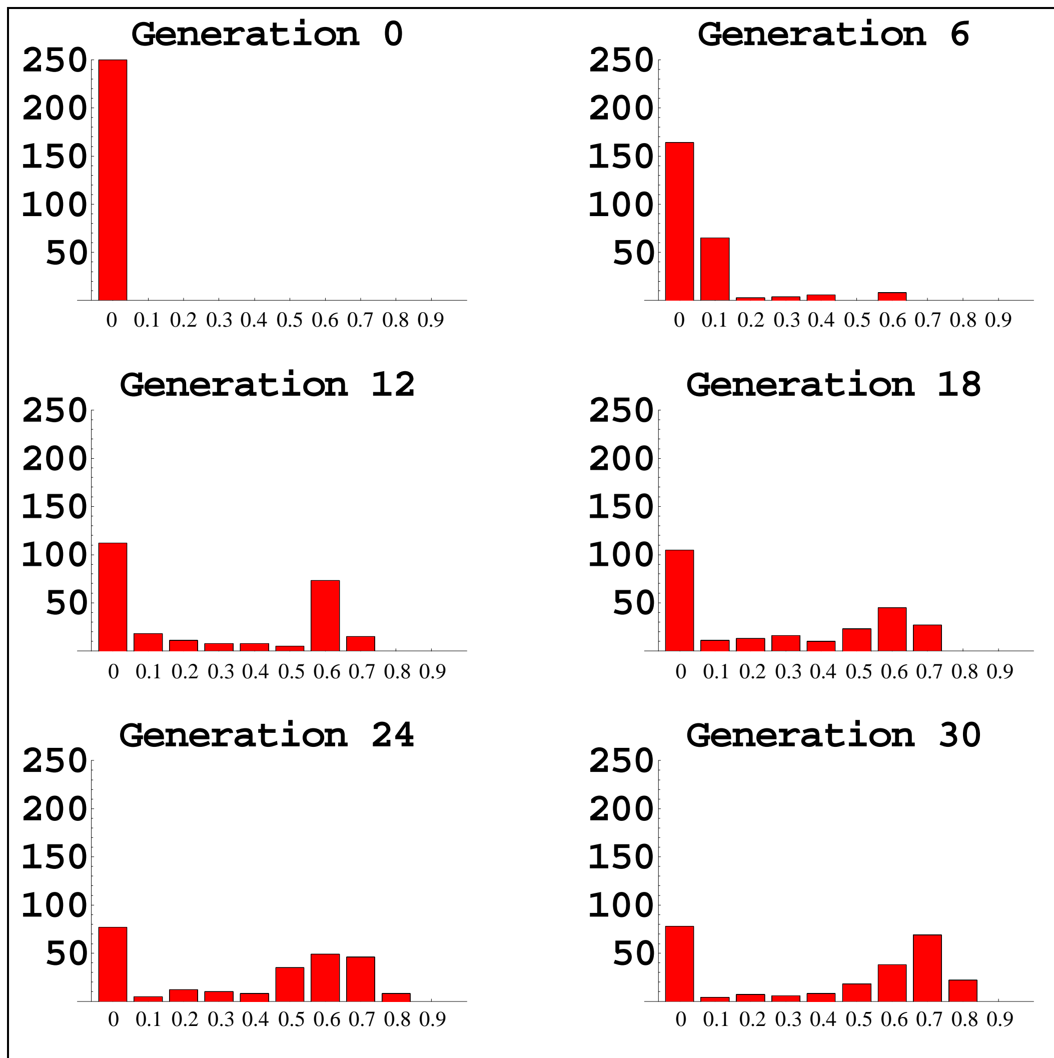

Figure 3.10. Fitness histograms

The histograms in *Figure 3.10* represent the division of individuals into the range of fitnesses displayed. There is an obvious move towards individuals with a higher fitness. In the initial generations, there are more individuals with lower fitnesses, but as the generations progress, the number of individuals with higher fitnesses increases. This is further indication that the average fitness of the population increases through evolution.

The histogram generation functions create a list of graphs. These graphs can either be animated or displayed individually. After extracting a subset of the graphs, Mathematica can display them in grid format as in *Figure 3.10*.

Experiment 1.6

This experiment tested the reaction of the algorithm to a reduced population size. The population was fixed at 150 individuals instead of the normal 250, and the complexity of expressions was reduced to 40 to promote parsimony. The parameters for this experiment are indicated in *Table 3.6*.

Parameter	Value
Population Size	**150**
Max no of Generations	**51**
Max initial size	**5**
Max size	**17**
Maximum complexity	**40**
Min solution fitness	**0.99**
Mutation probability	**0.05**
Crossover probability	**0.9**
Terminal set	**{x}**
Function set	**{PPlus, PPlus, PMinus, PTimes, PTimes, PDivide}**

Table 3.6. GP Parameters for symbolic regression - Exp 1.6

Six parallel runs were executed and the results are indicated in *Table 3.7*.

Run	Max Fitness	Time Taken (hours:minutes)
1	**1**	**3:35**
2	**1**	**3:22**
3	**1**	**3:27**
4	**0.469308**	**3:09**
5	**1**	**3:22**
6	**0.182729**	**2:55**

Table 3.7. Maximum fitnesses and times taken - Exp 1.6

Two runs did not find the perfect solution because of the reduced genetic material in the population. This smaller population size resulted in the GP algorithm searching more complex expressions rather than expressions with greater variety. It was concluded that the changed parameters did not allow sufficient variety to produce solutions with high probability.

Experiment 1.7

After tweaking the fitness function (Experiment 1.4), function set (Experiment 1.4/1.5), convergence criterion (Experiment 1.5), complexity restriction (Experiment 1.6) and population size (Experiment 1.6), the stability of the algorithm was tested in an additional 8 parallel runs. The population size was returned to 250 and the complexity to 50. The parameters for this experiment are indicated in *Table 3.8*.

Parameter	Value
Population Size	**250**
Max no of Generations	**51**
Max initial size	**5**
Max size	**17**
Maximum complexity	**50**
Min solution fitness	**0.99**
Mutation probability	**0.05**
Crossover probability	**0.9**
Terminal set	**{x}**
Function set	**{PPlus, PPlus, PMinus, PTimes, PTimes, PDivide}**

Table 3.8. GP Parameters for symbolic regression - Exp 1.7

All runs were continued beyond the maximum generations limit, and every one found the perfect solution.

Run	Time Taken (hours:minutes)
1	**3:20**
2	**0:22**
3	**0:38**
4	**0:23**
5	**1:54**
6	**0:13**
7	**2:25**
8	**4:54**

Table 3.9. Time taken for runs - Exp 1.7

Table 3.9 indicates the times taken for each run of the experiment. The average time taken was 1 hour and 46 minutes. Once again it can be seen that the randomness of the initial population has an influence on the path of evolution. Run 6 started off with

individuals that contained desirable genetic material, so found the perfect solution quickly. On the other hand, Run 8 took longer to find the solution because its initial population did not contain many highly fit individuals.

Experiment 1.8

After proving the stability of the algorithm, its reaction to noisy data was tested. The sample data was generated from the given equation in the usual manner and the y-values were perturbed by a maximum of 20%. For each perturbation, a uniformly-distributed random number was generated between -10 and +10 and this was then used as a percentage by which to either increase or decrease the y-value.

It was not expected that the algorithm would end with the perfect solution as before because of this noise. The experiment was repeated 8 times and two of these resulted in the original equation in spite of the imperfect data. The other six runs all ended in graphs which did not deviate much from the original path, as shown in *Figure 3.11*. This led to the conclusion that GP performs well with noisy data.

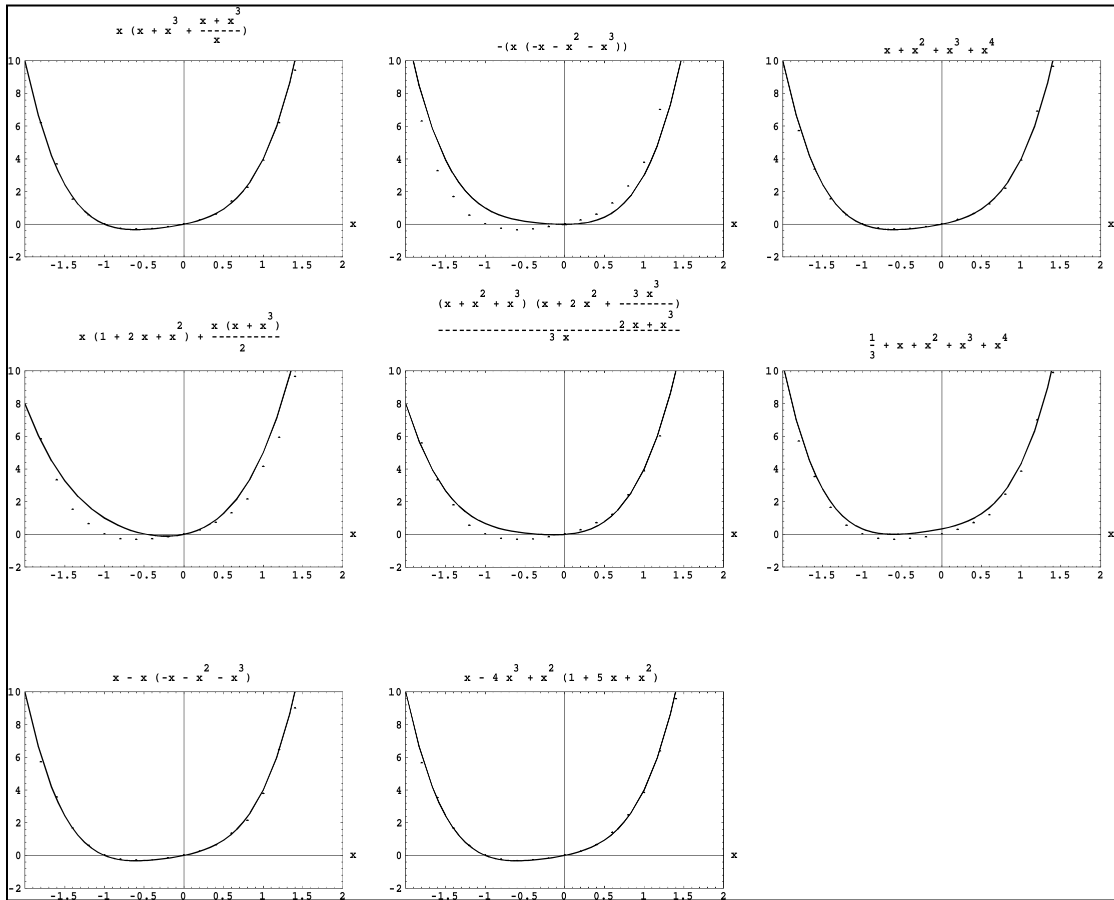

Figure 3.11. Sample data and their fitted equations for noisy data

Figure 3.11 shows the equations generated for each run, plotted against the sample data in each case.

Conclusion

The set of experiments 1.1-1.8 illustrates the effectiveness of the Mathematica implementation of GP in solving simple symbolic regression problems. The execution time is the most important concern since it affects the feasibility of such implementations.

Memory is another factor that affects performance of the algorithm. During runs which involved complex expressions or many generations, the Mathematica environment frequently ran out of memory and began using disk space for temporary storage. This had a distinctly negative impact on the speed of the operations.

During the course of the experiments, many optimisations were applied to the original code to speed it up and most of these had a considerable effect. The net effect is that the general algorithm cannot be improved on much more. So, if more complex problems are attempted, then the computing power would need to be increased. This increase can be either a change to a faster machine or a move towards parallel computing.

In order to further test the ability of Mathematica to solve problems using GP, the binary multiplexer problem, as described by Koza, was modelled in Mathematica [Koza, 1992]. GP must find an expression for a combinatorial logic circuit that multiplexes 2^n binary lines on the basis of an n-digit binary selector (where n is any small integer). The experiment was abandoned because the computer could not handle the complexity of expressions nor the size of populations necessary to find solutions. This further supported the need for greater computing power.

CHAPTER 4 : PARALLEL GENETIC PROGRAMMING

Introduction

Suitability of Parallel Processing for GP

Relative to classic analytical algorithms, evolutionary computation techniques like GAs and GP usually require vast computer resources in order to achieve a moderate success rate. Ideally, evolutionary algorithms can be executed on supercomputers or machines with comparable computational power. However, most researchers do not have access to such equipment, especially for research in previously unexplored areas. Attempts are made to improvise by using faster desktop machines and optimised algorithms. Sometimes it is possible to split up portions of the algorithm so that it can be run on multiple desktop machines simultaneously. This ability to process in parallel is inherent in many artificial intelligence paradigms, including evolutionary techniques.

Genetic programming is especially well suited for parallel processing because of the nature of the general algorithm. Most of the processing time in a GP can be attributed to the evaluation of fitnesses of individuals. This evaluation can be done in parallel for the simple reason that the fitness of each individual is independent of the rest of the population. The genetic operators do not depend on each other or any other routines, so they can safely be applied to individuals in a parallel fashion. Fitness-proportionate reproduction needs information about the entire population to implement the roulette-wheel mechanism. This process cannot be sub-divided, but this doesn’t have a major effect on the algorithm since the percentage of time taken for reproduction is comparatively much lower than that for fitness evaluations.

Parallel processing immediately brings to mind the notion of an algorithm executing cooperatively on multiple computers or a system supporting symmetric multiprocessing. This distributed model has the advantage of producing results faster, but is not the only reason for parallelisation. Since a parallel algorithm has to be split up into smaller execution modules, it requires less computational power at each

workstation. This makes it feasible to work on problem domains which necessitate large populations or large numbers of generations.

Mathematica stores all intermediate calculations in memory, filling up memory space with unnecessary details. If it is no longer necessary to execute the complete algorithm in one session, then Mathematica can be restarted at regular intervals. This prevents extraneous swapping to disk, as memory runs out. In order for Mathematica to be restarted, all necessary data has to be saved to disk. This makes it easier to recover from a computer crash during a GP run.

On a philosophical plane, it can be argued that parallel processing is better suited to GP because of the implicitly parallel nature of evolution. Since evolutionary computation techniques are based on nature and nature works in parallel, it seems reasonable that some benefit could be derived from parallelising evolutionary computation. This theory has been tested and found to be true in some cases, as described later in this section.

Parallel Processing Methodologies

There exist many approaches to applying parallel processing to an algorithm. One of the most important considerations is the programming layer at which the algorithm is divided. If the operating system and compiler support parallel processing, then this is normally done at a very low level, where single machine language instructions or high-level commands can constitute modules for parallel processing. If the computer does not have built-in support for parallel processing, then this has to be written in by the programmer. Built-in support for parallel processing can take advantage of finely-tuned operating systems and compilers. Programmatic implementations, on the other hand, allow greater freedom of choice in design of the algorithm, especially when deciding on the size and functionality of program sub-sections.

Fine-grain parallel processing refers to those instances where the algorithm has been sub-divided at the level of individual instructions or other similarly small program sections. In the context of GP, the fitness of each individual can be evaluated in parallel. This approach to parallel processing has the advantage that the general algorithm need not be changed, beyond the delegation of fitness evaluations. The

disadvantage of this strategy is that some parts of the algorithm will still have to be executed in a serial fashion, most notably fitness-proportionate reproduction. Since crossover involves more than one individual, it cannot be accomplished in parallel for each individual. Instead, the individuals will have to be submitted for processing in pairs.

Coarse-grain parallelism divides the problem into significantly large sub-sections. In the context of GP, the population of individuals is divided into sub-populations (e.g. a population of 800 is divided into 16 sub-populations, each containing 50 individuals). GP is then applied to each of these sub-populations in parallel. The advantage of this approach is that the entire algorithm can be executed on each sub-population simultaneously. Thus, fitness-proportionate reproduction will not create a bottle-neck as with fine-grain parallelism. The main disadvantage of this approach is that the general algorithm has to be changed substantially to sub-divide the population and coalesce the results. Since a single sub-population is too small to generate solutions with high fitnesses on its own, it has to work together with the other sub-populations. This interaction can be implemented in the form of either inter-population genetic operators or movement of individuals from one sub-population to another (aptly called migration). The latter approach is preferable since this movement can be separated from the process of creating new generations.

In any distributed computing environment the storage of data is a critical concern. GP requires the storage of expressions that correspond to the individuals of a population or populations. These individuals can be stored at either the workstations or on a central server. If the individuals are stored at the workstations, then there need be no interaction among the workstations during the creation of new generations. If the individuals are stored on a central server then the server has to send the individuals for processing to appropriate workstations. The latter approach results in more interaction among the computers (or processors in a multi-processor system), thus slowing down execution of the algorithm. This client-server model is better suited to coarse-grain parallelism, where interactions occur in batches rather than in a continuous sequence.

Some Existing Implementations

Since the parallelisation of GP does not depend on the form of the individuals, the issues surrounding its implementation are identical to the GA equivalent. As such, it is useful to consider parallel implementations of GAs, since the amount of research done in this field is fairly substantial [Cantu-Paz, 1995].

GALOPPS (Genetic ALgorithm Optimized for Portability and Parallelism) is a freely available library to implement parallel GAs in a coarse-grain manner [Goodman, 1996].

Koza also implemented parallel GP , using a network of transputers [Koza, 1995]. He used a coarse-grain algorithm to show that an optimal migration rate can be achieved, which would make the parallel algorithm perform relatively faster than a serial algorithm with the same population size. It was shown that it is possible to achieve a speedup in processing that is more than just linearly proportional to the number of processors or computers. This potential for super-linear performance can be exploited to speed up parallel algorithms, even if executed on a single processor.

Parallel Processing Model

Sub-populations and Migration

Coarse-grain parallelism (also known as island parallelism) was used as the underlying philosophy when changing the serial Mathematica algorithm into a parallel one.

The population of individuals is first split up into a pre-specified number of sub-populations. These populations then undergo evolution as in the serial model, possibly on different computers. After each new generation is created, the best individuals from each sub-population are compared to find the global solution. Statistical and recovery information is stored and the cycle continues until an acceptable solution is found.

However, such simple operation reduces the algorithm to a number of runs using smaller population sizes. A mechanism must be introduced to bind the populations together so that genetic material from one population can interact with individuals

from other populations. This is done by means of migration. After each generation has been processed, some individuals from one population may swap places with individuals from other populations. This migration is done on a fitness-proportionate basis to ensure that only the better genetic material can influence other sub-populations.

In the most general case, migration can occur between any two sub-populations. The net effect of such migration is that an individual from one sub-population may mate with an individual from any other sub-population during a single iteration of the algorithm. This is not desirable since it reduces the sub-populations to the original single population model. The advantages of the parallel model include its ability to preserve variety by allowing different populations to co-evolve without much interaction. This advantage is lost if there is too much migration or migration is allowed between any two sub-populations. To preserve variety, migration must be restricted to occur only between specified pairs of sub-populations. This is readily accomplished if the sub-populations are distributed spatially on the surface of a 2-dimensional grid, as shown in *Figure 4.1*.

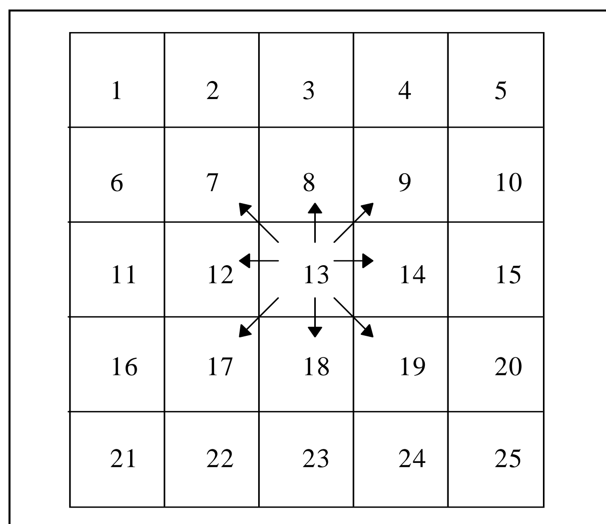

Figure 4.1. Rectangular spatial distribution of sub-populations showing migration possibilities for sub-population 13

Migration can be restricted to occur only between neighbouring sub-populations. This ensures that genetic material being evolved in one region of the population grid cannot directly affect the genetic material in other parts of the population grid.

In *Figure 4.1*, the possible migration partners for sub-population 13 in a 25 sub-population grid is shown. Since sub-population 13 is in the centre of the grid, there are 8 neighbouring sub-populations. However, the sub-populations along the edge of the grid have either 3 or 5 neighbours. In order not to bias the algorithm in favour of the central sub-populations, the sub-populations along the edge wrap around to the opposite ends of the grid. Thus, sub-population 6 has 1,2,7,11 and 12 as immediate neighbours, but may also perform migration with sub-populations 5, 10 and 15. Sub-populations at the corners wrap around to the diagonally opposite corners. This wrapping around of edges results in the 2-dimensional grid being transformed to a toroidal representation, where every sub-population has exactly 8 neighbours.

Although this migration strategy is used successfully to solve problems using the Mathematica implementation, there are other strategies that are either equivalent or better. Ryan discussed the differences between panmictic schemes (where migration can occur between any sub-populations), the Island Model (where migration with neighbours has a higher probability without excluding sub-populations that are further away) and Spatial Mating, as discussed above [Ryan, 1994]. Levine used a parallel GA with exactly one individual migrating during each iteration for implementation-specific reasons [Levine, 1994]. Toth incorporated migration into the reproduction operation [Toth, 1993].

An altogether different approach to migration was proposed by Punch [Punch, 1996]. He suggested that the best individuals from each sub-population be injected into a master population. This alternative may produce better results in some problem domains since it is geared towards the preservation of variety.

General Parallel Algorithm

initialise *global variables*

initialise *sub-populations*

check *all sub-populations for global solution*

__while__ solution not found

evolve, in parallel, new generations in each sub-population, using __reproduction__, __crossover__ and __mutation__

perform __migration__ between selected sub-populations

__check__ all sub-populations for global solution

Data Storage

During a run of the GP algorithm, population data, log files and statistical data need to be stored and retrieved. Population data, in particular, is accessed by the processors (or computers) that perform evolution on the population. In a multi-processing environment, the data has to be stored on the storage devices of the computer. In a distributed environment, however, the data can either be stored on a central server or on the workstations. Workstation-based storage of population data necessitates regular communication of data between workstations. This communication has to conform to a pre-specified network protocol. Since network protocols are specific to the platforms in use, it was decided not to use this form of direct communication. Instead, the data is stored on a central server and the directory in which the data resides is shared with all the workstations. Thus the server and workstations have access to all the data and communications can be handled transparently by the operating system. The implementation is portable across computers and operating systems, as long as file sharing is supported. The experiments in the next chapter were successfully conducted on the following platforms: Windows 3.1 (server), MsDos (clients), Windows 95 (clients/server), Linux (server).

Approaches to Job Control

In any environment where tasks are carried out in parallel, these tasks have to be scheduled to execute in the correct order. For example, migration cannot be started until all the sub-populations have been processed.

In the parallel GP, the sub-populations need to be evolved in parallel. Thereafter all populations must be checked for a fitter global solution. Finally, migration takes place in parallel. This sequence of steps repeats until an acceptable solution is found.

There are three distinctly different scheduling scenarios:

- the number of processors is greater than the number of sub-populations
- the number of processors is equal to the number of sub-populations
- the number of processors is less than the number of sub-populations

If the number of processors is greater than the number of sub-populations then every sub-population can be assigned to a single processor. Each processor performs evolution on only one sub-population, with some processors lying idle - the available resources outnumber the requirements, resulting in wastage. Migration has seemingly more stringent requirements since, in the worst case, the number of pairs of sub-populations is equal to $4n$, where n represents the number of sub-populations. Thus $4n$ processors would be required for the migration. However, unless sophisticated record locking is used, it is not possible for two processors to simultaneously access individuals from a single population. Each sub-population would not be able to participate in simultaneous migration with more than one of its eight neighbours. Scheduling would be needed for this stage, to coordinate the selection of pairs of sub-populations to which the migration operation is applied. In fact, the migration stage requires scheduling irrespective of the ratio of processors to sub-populations.

If the number of processors is equal to the number of sub-populations then there is no wastage of computer resources. Once again, every processor can operate on different sub-populations, as described above.

If the number of processors is less than the number of sub-populations then each processor cannot evolve just one population. Scheduling is necessary to assign tasks to the processors, be the tasks evolution or migration. This is the most general case since it will not be dependent on the number of processors or sub-populations. During the course of the experiments conducted (as outlined in the next chapter), a scheduling system was built to cater for these requirements.

Initially a peer-to-peer system was created, where scheduling was a cooperative function of the processors. During initialisation, the population is partitioned and stored in separate files. A series of lock files is created, one for each sub-population, with appropriate names eg. POP1.LCK, POP2.LCK, etc. Each processor then starts executing a loop, where it first searches for a lock file and then processes the corresponding sub-population, erasing the lock file when complete. In order to preclude the possibility of two processors evolving the same sub-population, the lock files must somehow be flagged. Two methods of flagging were attempted. Since Mathematica does not provide file locking mechanisms, the ability of the operating system (in particular Windows 3.1, but applicable to most operating systems) to disallow two processes simultaneously having write access to a file was exploited. The code to implement this is shown below.

```
Lock2[x_]:=Module[
                   {aFile},
                   Off[OpenAppend::noopen];
                   Off[General::aofil];
                   aFile=OpenAppend[x];
                   On[OpenAppend::noopen];
                   On[General::aofil];
                   If[
                      SameQ[aFile, $Failed],
                      -1,
                      aFile
                     ]
                  ]

Lock[x_]:=Module[
                  {},
                  If[
                     FileNames[x]=={},
                     -2,
                     Lock2[x]
                    ]
                 ]
```

`Lock` attempts to lock a sub-population, as denoted by the file given as its parameter. If the file does not exist, the function returns **-2**. If the file is already locked by another process, the function return **-1**. Otherwise, it opens the file for writing (using `Lock2`) and returns the file handle. After the population has been processed, the file can be deleted.

This scheme did not work since file locking has to be an indivisible operation to support parallel processing, and that could not be guaranteed in a high-level language

like Mathematica. A few random scheduling errors occurred because of the instability of the platform; these were unacceptable. The alternative was not to rely on the implicit locking of files by the operating system and Mathematica. Instead of locking files before processing a population, the files were simply deleted. This also failed as a scheduling mechanism. Primarily for these reasons, it was decided to introduce a secondary program into the algorithm for the express purpose of performing scheduling among the processors.

This program could be run on any machine with the same shared directory as the processing workstations. Since the only link between processors is the shared directory, this scheduler also has to use files to signal the start and end of each job. The scheduler only manipulates files, so it was not necessary to implement it in Mathematica. By writing the scheduler in C++ for MS-Windows, it had the added advantage that the scheduler could be run on a workstation simultaneously with a Mathematica session. This obviated the need for a separate scheduling computer. It was also possible to incorporate dynamic starting and stopping, timing of the algorithm and continuous displaying of the state of the GP network into the scheduler. *Figure 4.2* shows a screen snapshot of the scheduler program.

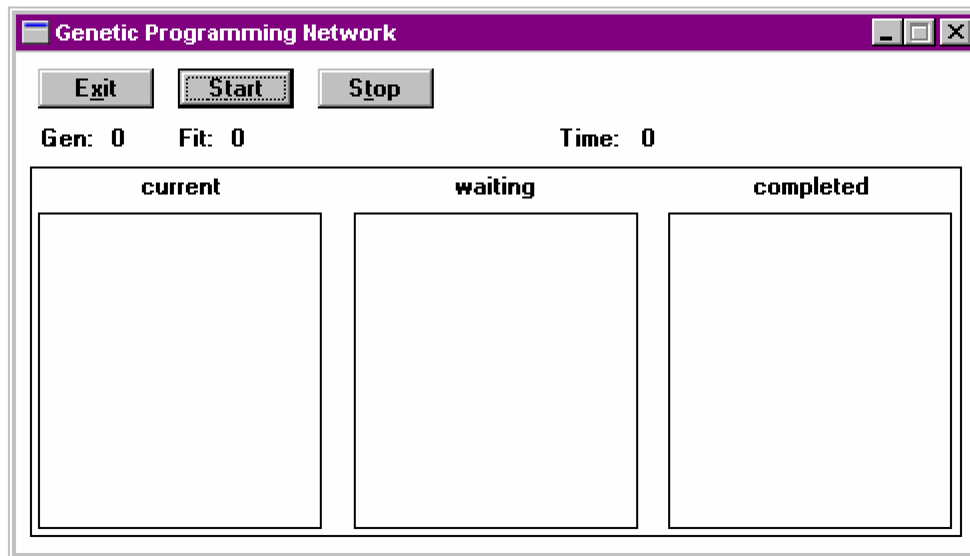

Figure 4.2. Screen snapshot of scheduler

When the algorithm is run, the first column indicates the tasks in progress, the second indicates waiting tasks and the third completed tasks. The current generation, fitness of best individual and total time taken are also shown.

Scheduling

The code for the scheduler is contained in Appendix B.

The scheduler uses the same shared directory as all the processors. Communication is performed by the creation and deletion of files in particular directories. It is assumed that directories whose names begin with "PROC" refer to processors. The number appended to "PROC" is the unique identification number of the processor eg. PROC4 refers to processor 4. These directories are created by Mathematica during initialisation of the run.

Files are created in these directories to signal that the corresponding processors must execute particular tasks. Each processor, whenever idle, constantly monitors its directory for such signals. When a file is found, the processor interprets the task to be performed and then deletes the file. The absence of the signal file is noticed by the scheduler, which then attempts to allocate a new task.

The first task allocated to processors is that of evolving new generations for each sub-population. The signal files are named "POP", suffixed by the number of the sub-population. Each processor continuously processes sub-populations until all the sub-populations have been progressed one generation. The first processor (PROC1) is then given the task "MSTART" which signals it to check for global solutions and prepare for migration. In preparation for migration, random pairs of neighbouring sub-populations are selected and stored in the "POP.INF" file. This file is read in by the scheduler and the contents are stored in a matrix, associating each sub-population with a subset of its neighbours. A matrix is used to store these associations as efficiently as possible; also the access time to check on a particular pair in the matrix is constant, irrespective of the number of sub-populations.

second sub-population	first sub-population *1*	*2*	*3*	*4*
1	0	0	0	1
2	0	0	1	0
3	0	1	0	1
4	1	0	1	0

Figure 4.3. Matrix of migration possibilities

Figure 4.3 shows these associations for a 4 sub-population model. According to the table, migration of members may occur between sub-population 4 and sub-population 1. Obviously this corresponds to the pair containing 1 and 4, resulting in a symmetric matrix. The storage space is reduced by using only a triangular matrix. Each position in the matrix indicates whether or not the two associated sub-populations are eligible for migration.

In addition, a list of sub-populations is maintained. This list indicates whether each sub-population is currently involved in a migration operation or not.

When a processor is free, the scheduler searches through the list of sub-populations until it finds a pair where migration is impending, as per the matrix. Migration is signalled by a file beginning with the letter "M" and ending with a unique number assigned to each pair of sub-populations. The migration of that pair is cancelled in the matrix and the state of the pair is updated in the list of sub-populations.

The interaction of the boolean-valued list and boolean-valued matrix provides a compromise in terms of speed and efficiency in anticipation of larger search spaces and greater numbers of sub-populations.

When all sub-population pairs are removed from the matrix, the scheduler resets itself and begins to repeat the process of evolving sub-populations.

Mathematica Implementation

The Mathematica implementation was altered to support parallel execution of the GP algorithm. The complete code for this implementation is found in Appendix C. Although genetic operators are not affected, initialisation, sequencing of operations, population manipulation and statistical routines have to be changed.

The population of individuals is first split up into sub-populations during the initialisation stage. The number of sub-populations must be pre-specified.

```
NoOfSubpopulations = 4

Initialize:=Module[
                    {Proc, DelList},

                    (* paragraph 1 *)
                    Off[DeleteFile::nffil];
                    DeleteFile["calced.m"];
                    DeleteFile["pop.inf"];
                    DelList=FileNames["logfile.*"];
                    If[DelList!={}, DeleteFile[DelList]];
                    DelList=FileNames["*.plg"];
                    If[DelList!={}, DeleteFile[DelList]];
                    DelList=FileNames["*.log"];
                    If[DelList!={}, DeleteFile[DelList]];
                    DelList=FileNames["backup.*"];
                    If[DelList!={}, DeleteFile[DelList]];
```

```
On[DeleteFile::nffil];
Map[
    (DeleteDirectory[#,
     DeleteContents->True])&,
    FileNames["PROC*"]
   ];

(* paragraph 2 *)
Genetic`Parameters`GlobalSolution=1;
Genetic`Parameters`GlobalSolutionFitness=0;
Genetic`Parameters`GlobalSolutionSet={};
Genetic`Parameters`TotTime=0;

(* paragraph 3 *)
Save["pop.log",
  Genetic`Parameters`GlobalSolution];
Save["pop.log",
  Genetic`Parameters`GlobalSolutionFitness];
Save["pop.log",
  Genetic`Parameters`GlobalSolutionSet];
Save["pop.log", Genetic`Parameters`TotTime];

(* paragraph 4 *)
MakePossibilities;

(* paragraph 5 *)
Save["calced.m",
  Genetic`Parameters`GPossibilities];
Save["calced.m",
  Genetic`Parameters`GPossParameter];
Save["calced.m",
  Genetic`Parameters`GTermLength];
Save["calced.m",
  Genetic`Parameters`GPossLength];

(* paragraph 6 *)
InitNames;

(* paragraph 7 *)
Save["calced.m",
  Genetic`Parameters`PopulationNames];
Save["calced.m",
  Genetic`Parameters`MigrationPairs];

(* paragraph 8 *)
Genetic`Parameters`PopulationSize=
 Genetic`Parameters`PopulationSize/
 Genetic`Parameters`NoOfSubpopulations;

(* paragraph 9 *)
GInformation;

(* paragraph 10 *)
Map[InitializePop,
    Genetic`Parameters`PopulationNames];

(* paragraph 11 *)
CheckGlobalSolutions;
]
```

`Initialize` initialises all variables and sub-populations in preparation for the execution of the GP algorithm.

All traces of previous runs are erased. This includes log files created and directories used to store processor information (paragraph 1). Global variables are initialised (2) and stored in the global information file (3). In order to save time during the generation of individuals, the terminal and function sets are joined during initialisation and stored in a disk file - CALCED.M (4/5). The names of populations are generated together with migration pairs and these are stored in the same disk file (6/7). The population size is divided by the number of sub-populations (8) and information on the run is displayed (10). Each sub-population is initialised with random individuals (11), their fitnesses are evaluated and global statistics are calculated (12).

After initialising the variables, each processor must be registered for scheduling purposes. This registration simply creates a unique directory for each processor.

```
RegisterProc[x_]:=Module[
                         {proc},
                         proc=StringJoin["PROC", ToString[x]];
                         CreateDirectory[proc];
                        ]
```

The algorithm is started from the command-line of the operating system using a batch file. This batch file creates a unique copy of itself for each processor and then continuously runs the GP algorithm in Mathematica.

```
Contents of START.BAT

copy start2.bat temp%1.bat
temp%1 %1

Contents of START2.BAT

:st
call math -run "<<p.m;Genetic`Main`StartRun[%1];Quit[]"
goto st
```

START.BAT is called with the single parameter being the number of the processor. This parameter is passed onto the Mathematica function **`StartRun`**, which executes the GP.

```
(* Start run of algorithm *)
StartRun[x_]:=Module[
              {result, log, i},
```

```
Do[
   log=StringJoin["LOGFILE.", ToString[x]];
   $Output=Append[$Output, OpenAppend[log]];
   SetOptions[$Output[[2]], FormatType->TextForm];

   Genetic`Parameters`Processor=
    StringJoin["PROC", ToString[x]];

   CheckAbort[
             ApplyGen,
             0
            ];

   Close[$Output[[2]]];
   $Output=Take[$Output, 1],
   {i, 1, Genetic`Parameters`Epoch}
  ];
]
```

A log file is opened at the beginning of the routine to mirror all screen output during the session. This log file is subsequently closed at the end of the routine. The name of the processor is gleaned from the parameter and **ApplyGen** is called. This is repeated **Epoch** (default value = 20) times before restarting the Mathematica interpreter, to minimise the effect of time taken to run the interpreter from disk.

```
(* Apply Genetic algorithm *)

ApplyGen := Module[
   {popfile, onetime, poplog, mig, OrigDirectory},

   (* paragraph 1 *)
   BeginPackage["Genetic`Parameters`", "Global`"];
   Get["calced.m"];
   EndPackage[];

        (* paragraph 2 *)
        Print["Waiting for processor start flag ..."];
        popfile=GetPopFile;
        While[
              SameQ[popfile, "NOFILES"],
              Pause[1];
              popfile=GetPopFile
             ];
        If[
           SameQ[StringTake[popfile, 1], "M"],
           Migrate[popfile];
           Return[]
          ];

       (* paragraph 3 - process population *)
        BeginPackage["Genetic`Parameters`", "Global`"];
        Get[StringJoin[popfile, ".log"]];
        EndPackage[];

        (* paragraph 4 *)
        onetime=Timing[
         Print[popfile, "-G", Generation, ": mating pool ... ",
```

```
      Timing[newpop=CreateNewGeneration[Population]][[1]]];
     Print[popfile, "-G", Generation, ": crossover   ... ",
      Timing[newpop=Crossover[newpop]][[1]]];
     Print[popfile, "-G", Generation, ": mutation    ... ",
      Timing[newpop=Map[Mutate, newpop]][[1]]];
     Generation++;
     Population=newpop;
     Print[popfile, "-G", Generation, ": fitnesses   ... "];
     Print[popfile, "-G", Generation, ": done        ... ",
      Timing[CheckSolution[Generation, newpop,
             popfile]][[1]]];
     Print[popfile, "-G", Generation, ": best-of-run   = ",
      SolutionFitness];
    ][[1]];
    Time[onetime, popfile, "-G",
     Generation, ": time for gen  = "];

    (* paragraph 5 *)
    TimeTaken+=onetime;
    Save[StringJoin[popfile, ".new"], Population];
    Save[StringJoin[popfile, ".new"], Fitnesses];
    Save[StringJoin[popfile, ".new"], Generation];
    Save[StringJoin[popfile, ".new"], TimeTaken];
    Save[StringJoin[popfile, ".new"], Solution];
    Save[StringJoin[popfile, ".new"], SolutionFitness];
    Save[StringJoin[popfile, ".new"], SolutionSet];
    RenameFile[StringJoin[popfile, ".log"],
               StringJoin[popfile, ".old"]];
    RenameFile[StringJoin[popfile, ".new"],
               StringJoin[popfile, ".log"]];
    DeleteFile[StringJoin[popfile, ".old"]];

    (* paragraph 6 *)
    poplog=OpenAppend[StringJoin[popfile, ".plg"]];
    WriteString[poplog, ","];
    Write[poplog, {Generation, Fitnesses}];
    Close[poplog];
    Print[popfile, "-G", Generation, ": system saved ..."];

    (* paragraph 7 *)
    OrigDirectory=Directory[];
    SetDirectory[Genetic`Parameters`Processor];
    DeleteFile[popfile];
    SetDirectory[OrigDirectory];
]
```

First the complete function and terminal sets are loaded from the disk file, where they were saved during initialisation (paragraph 1). Then the processor goes into a loop, waiting for a signal file to be created by the scheduler (2). If the name of this file begins with “M” then it is assumed that migration is intended and the relevant function is called. If evolution is intended, then the normal genetic operators are applied (4). Since the interpreter exits after every task, the sub-population is loaded from disk before evolution (3) and saved afterwards (5). Statistical information is stored (6) and the signal file is deleted to inform the scheduler that the task is complete (7).

```
(* perform migration based on parameters *)
Migrate[popf_]:=Module[
                        {OrigDirectory, FullNum, firstpop,
                         secondpop},

                        If[
                           SameQ[StringDrop[popf, 1], "START"],
                           CheckGlobalSolutions;
                           If[
                         Genetic`Parameters`GlobalSolutionFitness
                              >=MinFitness,
                              OrigDirectory=Directory[];
                              SetDirectory[
                               Genetic`Parameters`Processor];
                              Save["DONE", MinFitness];
                              SetDirectory[OrigDirectory]
                             ],
                           FullNum=ToExpression[
                                    StringDrop[popf, 1]];


                           firstpop=Floor[
                            FullNum/NoOfSubpopulations]+1;
                           secondpop=Mod[FullNum,
                            NoOfSubpopulations]+1;
                           MigratePop[StringJoin["POP",
                                      ToString[firstpop]],
                                      StringJoin["POP",
                                      ToString[secondpop]]]
                          ];
                        OrigDirectory=Directory[];
                        SetDirectory[
                         Genetic`Parameters`Processor];
                        DeleteFile[popf];
                        SetDirectory[OrigDirectory];
                       ]
```

`Migrate` handles all tasks except evolution of generations.

If the signal file is “MSTART” then **`CheckSolutions`** is called to extract the best solution from all the sub-populations. Otherwise, the names of the populations to participate in migration are generated and the **`MigratePop`** is called with these as parameters. Finally, the signal file is deleted to inform the scheduler that the task is complete.

`MigratePop` performs migration between two sub-populations. They are loaded simultaneously into memory and random individuals are swapped. Thereafter the populations are saved over the original data. Individuals are selected for migration in a fitness-proportionate manner, using the roulette-wheel technique as discussed earlier. The average number of individuals to migrate are specified by **`MigrationPercentage`** (default = 0.1). The actual number of individuals is

generated by a Gaussian-distributed random number with **`MigrationPercentage`** as a mean and a standard deviation of **`MigrationDeviation`** (default = 0.05).

In order to speed up the algorithm, migration is not performed between every possible pair of sub-populations during each iteration of the algorithm. The probability that migration occurs between any two sub-populations is defined by **`MigrationProbability`** (default is 1 in 4).

Sequence of Function Calls

In order to use the parallel implementation, the parameters for the run must be defined in a text file in Mathematica input format (with a default name of "P.M"). Function and terminal sets and the fitness function are mandatory but the other parameters will be assigned default values if not defined.

All computers working on the problem must be networked and a shared directory created, containing the parallel GP program and data files.

Mathematica should be started on a single computer in order to **`Initialize`** the populations. Thereafter each processor must be registered with the **`RegisterProc`** function.

The algorithm can be started on each processor by running the START.BAT batch file, supplying the number of the processor as the single parameter. All the processors will go into a loop, waiting for tasks to be assigned to them.

The scheduler, GPNET.EXE, must then be run and, by clicking on the Start button, the scheduling operations begin. The various processors will then cooperatively evolve new generations and perform migration whenever necessary.

CHAPTER 5 :
APPLICATIONS OF PARALLEL GP

Statistical Analysis Techniques

Statistics in a parallel GP can be produced to analyse the performance of either the entire population or the individual sub-populations. The entire population indicates global trends while a study of the sub-populations can ensure differences in the composition of the population at different points on the population grid.

As before, graphs can be generated to indicate the convergence or divergence of the algorithm by plotting the maximum and minimum fitnesses of each generation.

```
GlobalCurve:=Module[
                    {t, MaxG, MinG, AveG},

                    BeginPackage["Genetic`Parameters`"];
                    Get["pop.log"];
                    EndPackage[];

                    t=MapThread[List,
                     Genetic`Parameters`GlobalSolutionSet];

                    MaxG=ListPlot[MapThread[List,
                                   {t[[1]], t[[2]]}],
                                  PlotRange->{{0, Max[t[[1]]]},
                                   {0, 1}},
                                  PlotStyle->{RGBColor[1,0,0]},
                                  Frame->True,
                                  FrameLabel->{
      "Generation       Fit(ness): red=max green=min blue=ave",
                                              "Fit"},
                                  PlotLabel->
                                   "Global Fitness Curve",
                                  PlotJoined->True,
                                  DisplayFunction->Identity];

                    MinG=ListPlot[MapThread[List,
                                   {t[[1]], t[[4]]}],
                                  PlotRange->{{0, Max[t[[1]]]},
                                   {0, 1}},
                                  PlotStyle->{RGBColor[0,1,0]},
                                  Frame->True,
                                  FrameLabel->{
      "Generation       Fit(ness): red=max green=min blue=ave",
                                              "Fit"},
                                  PlotLabel->
                                   "Global Fitness Curve",
                                  PlotJoined->True,
                                  DisplayFunction->Identity];
```

```
                    AveG=ListPlot[MapThread[List,
                                  {t[[1]], t[[6]]}],
                                 PlotRange->{{0, Max[t[[1]]]},
                                  {0, 1}},
                                 PlotStyle->{RGBColor[0,0,1]},
                                 Frame->True,
                                 FrameLabel->{
"Generation        Fit(ness): red=max green=min blue=ave",
                                             "Fit"},
                                 PlotLabel->
                                  "Global Fitness Curve",
                                 PlotJoined->True,
                                 DisplayFunction->Identity];

                    Show [{MaxG, MinG, AveG},
                          DisplayFunction->$DisplayFunction];
                 ]
```

The global statistical information saved during the run is read in by **`GlobalCurve`** and three different graphs are generated in memory, one each to display the maximum, minimum and average fitnesses. Eventually, the three graphs are superimposed and displayed on the screen.

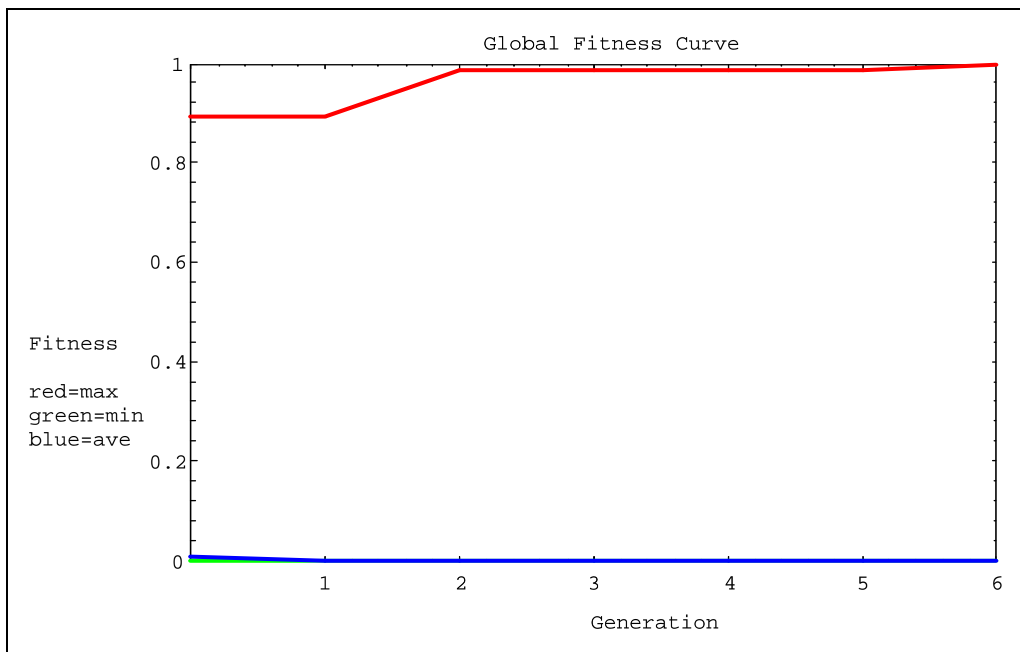

Figure 5.1. Output from GlobalCurve, displaying maximum, minimum and average fitnesses of generations

A typical output from **`GlobalCurve`** is shown in *Figure 5.1*. The maximum, minimum and average fitnesses are drawn in red, green and blue respectively to

enhance clarity. In this graph, as in most fitness curves, the average fitness is almost equal to the minimum. This is not critical because the maximum fitness is of greater importance.

Similar statistics can be generated for individual sub-populations. In order to cater for all sub-populations simultaneously, the graph can be promoted to a 3-D format with the number of the sub-population being the third dimension. This is not desirable since the sub-populations would have to be re-arranged in a linear fashion. Peculiarities in the population grid are more obvious if the statistics are arranged in a grid corresponding to the sub-populations. However, since this is already a two-dimensional structure, only one piece of information can be displayed. For example, a 3-D surface can be used to indicate the maximum fitnesses in generation 0 in all sub-populations. A series of such graphs can then indicate maximum fitnesses of subsequent generations.

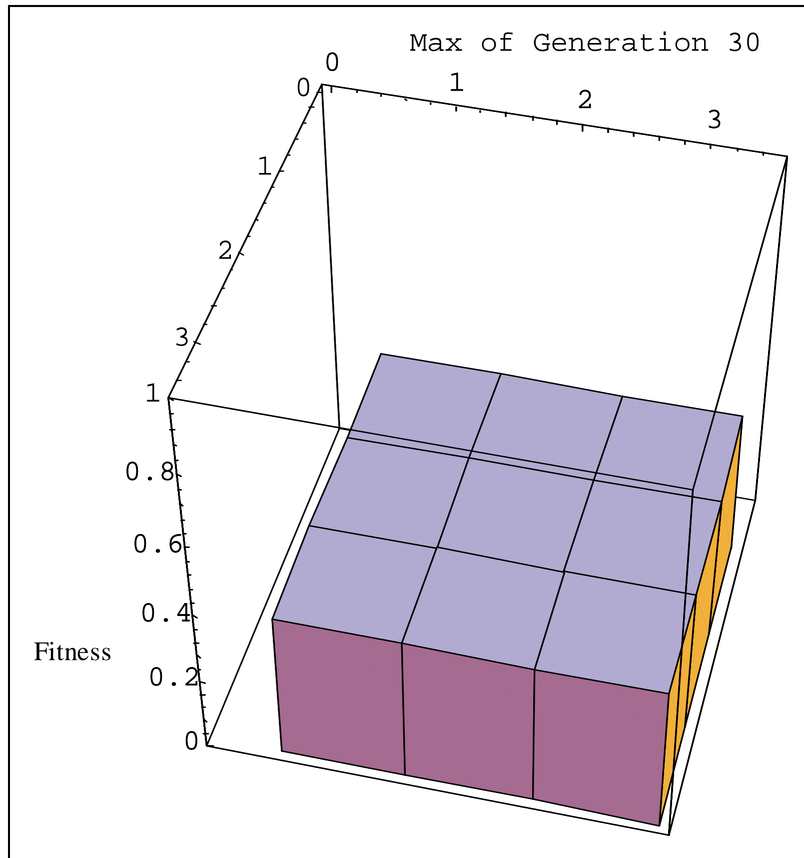

Figure 5.2. Typical maximum fitness histogram

Figure 5.2 shows a typical histogram for individual sub-populations. The horizontal plane indicates the position of each population in the population grid while the heights of the bars represent the maximum fitnesses for that generation. All the experiments in this chapter resulted in similar graphs, where there is not much difference in fitness among the various sub-populations. This is because the small number of sub-populations used did not promote variety of individuals. Few sub-populations were used in order to minimise the ratio of communication time to actual computation time.

Similar graphs can be generated for the average fitnesses. Graphs such as these are produced by the **`CalcHistogram`** function, using Mathematica's existing 3-D graph capabilities.

```
CalcHistogram:=Module[
                        {t, data, popfit, figs, gen,
                         popsize=0, numgen,
```

```
 popfiles, first=1, maxes, popnumber,
 inFile, outFile},

popfiles=FileNames["pop*.plg"];
popfiles=Sort[
              popfiles,
              (Less[GetPopNumber[#1],
               GetPopNumber[#2]])&
             ];
Histogram3DMax=Table[0,
 {Length[popfiles]}];
Histogram3DAve=Table[0,
 {Length[popfiles]}];

Map[
    (Print["copying file ", #];
     inFile=OpenRead["pop1.plg"];
     outFile=OpenWrite["pop.ful"];
     While[
           i=Read[inFile, String];
            Not[SameQ[i, EndOfFile]],
           WriteString[outFile, i, "\n"]
          ];
     Close[inFile];
     WriteString[outFile, "}"];
     Close[outFile];

     Print["reading in data"];
     BeginPackage[
      "Genetic`Parameters`"];
     Get["pop.ful"];
     EndPackage[];

     Print["separating data"];
     popfit=MapThread[List,
      Genetic`Parameters`pop][[2]];
     numgen=Max[MapThread[List,
      Genetic`Parameters`pop][[1]]];

     If[
        first==1,
        data=Table[Table[0, {10}],
                   {numgen}];
        first=0
       ];

     Print["discretizing data"];
     Do[
        figs=Map[Floor,
        popfit[[gen]]*10];
        figs=Map[If[#==0, 1, #]&, figs];
        Map[(data[[gen, #]]++)&, figs],
        {gen, 1, numgen}
       ];

     Print["extracting maximums"];
     maxes={};
     Do[
        maxes=Append[maxes,
         Max[popfit[[gen]]]],
        {gen, 1, numgen}
       ];
```

```
 popnumber=ToExpression[
  StringDrop[StringDrop[#, 3], -4]];
 Histogram3DMax[[popnumber]]=maxes;

 Print["extracting averages"];
 maxes={};
 Do[
    maxes=Append[maxes,
           Apply[Plus,
popfit[[gen]]]/Length[popfit[[gen]]]
                ],
    {gen, 1, numgen}
   ];
 Histogram3DAve[[popnumber]]=maxes;

 popsize+=Length[popfit[[1]]])&,

 popfiles
];

Print["generating global graphs"];
HistogramData=
  Table[
        BarChart[data[[gen]],
         BarLabels->Table[i, {i, 0,
          0.9, 0.1}],
         PlotRange->{{0, 11},
          {0, popsize}},
         PlotLabel->StringJoin[
          "Global Generation ",
          ToString[gen]],
         DisplayFunction->Identity],
        {gen, 1, numgen}
       ];

Print["generating maximum graphs"];
Histogram3DMax=MapThread[List,
 Histogram3DMax];
Histogram3DMax=Map[Partition[#,
 Sqrt[Length[popfiles]]]&,
 Histogram3DMax];
Histogram3DMax=
  Table[
   BarChart3D[Histogram3DMax[[gen]],
    PlotRange->{Automatic,
     Automatic, {0,1}},
    PlotLabel->StringJoin[
     "Max of Generation ",
     ToString[gen]],
    ViewPoint->{4,1,4},
    DisplayFunction->Identity],
   {gen, 1, numgen}
       ];

Print["generating average graphs"];
Histogram3DAve=MapThread[List,
 Histogram3DAve];
Histogram3DAve=Map[Partition[#,
 Sqrt[Length[popfiles]]]&,
 Histogram3DAve];
Histogram3DAve=
  Table[
```

```
                                        BarChart3D[Histogram3DAve[[gen]],
                                         PlotRange->{Automatic,
                                          Automatic, {0,1}},
                                         PlotLabel->StringJoin[
                                          "Ave of Generation ",
                                          ToString[gen]],
                                         ViewPoint->{4,1,4},
                                         DisplayFunction->Identity],
                                        {gen, 1, numgen}
                                            ];
                                ]
```

CalcHistogram generates these 3-D graphs for the maximum and average fitness values. As a result of the function, two lists of graphs are created: **Histogram3DMax** contains the maximum fitness graphs and **Histogram3DAve** contains the average fitness graphs. In addition, the set of global histograms is generated and stored in **HistogramData**. Although the routines to generate the global histograms were already available in the original serial algorithm, it was decided to incorporate all graph generation activity into one loop to prevent repetitive preprocessing of the fitness data.

The population data file corresponding to each sub-population is read in and processed. First the unnecessary information is pruned from the data, then the data is divided into discrete batches for the global histograms. Finally, the maximum and average values are calculated and the graphs are created in memory.

Mathematica's built-in animation capabilities were exploited to animate this information, thus overcoming the requirement for an additional dimension in representing the data. In the absence of animation capabilities, it is still possible to display multiple graphs on a single page, as shown in *Figure 3.10*.

These various graphical statistics display the trends that manifest themselves in the population data as generations progress. The global fitness curve and the 3-D histograms indicate the nature of convergence or divergence of the algorithm. The global histogram shows the implicit shifts in fitness of the entire population. These were used extensively during the modelling of experiments in order to optimise parameters to increase the probability of acceptable solutions.

Experiment 2: Parallel Symbolic Regression

In order to evaluate the effectiveness of the parallel GP algorithm in Mathematica, the symbolic regression problem (Experiment 1) was revisited. This time, the population was divided into 9 sub-populations and the computations were distributed among a set of workstations. The number of workstations was varied in order to assess its impact on the ratio of computation time to communication time.

Test Data

The equation used to generate sample points was once again

$$y = x^4 + x^3 + x^2 + x \quad \text{...} \quad (5.1)$$

The range of x-values from -1 to 1 was divided into 10 adjacent sections, with 11 boundary points. Y-values were generated for each of these eleven boundary points using *Equation 5.1*. The x-values and corresponding y-values are shown in *Table 5.1*.

x	y
-1	**0**
-4/5	**-164/625**
-3/5	**-204/625**
-2/5	**-174/625**
-1/5	**-104/625**
0	**0**
1/5	**156/625**
2/5	**406/625**
3/5	**816/625**
4/5	**1476/625**
1	**4**

Table 5.1. Sample points - Exp 2

All values are stored as fractions to retain a high degree of accuracy when calculating the fitnesses. These sample values were used for all iterations of Experiment 2. As was done previously, the x-values are equidistant to promote the generation of a more parsimonious equation.

Experiment 2.1

The first iterations of the experiment attempted to compare the performances of various configurations of workstations/processors. The parameters for the run were consistent at the values indicated in *Table 5.2*.

For this experiment, migration took place on a single computer after each generation was evolved i.e. one computer performed migration on the entire set of sub-populations.

Parameter	Value
Population Size	**450**
No of Sub-populations	**9**
Max no of Generations	**51**
Max initial size	**5**
Max size	**17**
Maximum complexity	**50**
Min solution fitness	**1**
Mutation probability	**0.1**
Crossover probability	**0.9**
Terminal set	**{x}**
Function set	**{PPlus, PPlus, PTimes, PTimes, PMinus, PDivide}**

Table 5.2. Parameters for parallel symbolic regression - Exp 2.1

As shown in the table, the number of sub-populations is 9, implying that the sub-populations were distributed on a 3x3 grid. Although this does not assist is preserving variety of the population, it does make it possible to execute the algorithm in parallel, which was the primary focus of this experiment.

The experiment was repeated 15 times, 5 times each using 1 processor, 3 processors and 9 processors. In all instances the perfect solution, as indicated by *Equation 5.1*, was evolved. The times taken to achieve these results are shown in *Table 5.3*.

Run No	No of Processors	Time Taken to find Solution (h:m:s)	Generations Processed	Time Taken to Process Single Generation (s)
1A	**1**	**2:20:58**	**18**	**470**
1B	**1**	**2:44:30**	**20**	**494**
1C	**1**	**2:0:10**	**15**	**481**
1D	**1**	**2:47:24**	**21**	**478**
1E	**1**	**0:20:44**	**3**	**415**
Average	**1**	**2:02:45**	**15**	**485**
2A	**3**	**1:20:24**	**19**	**254**
2B	**3**	**0:22:07**	**7**	**190**
2C	**3**	**0:29:31**	**9**	**197**
2D	**3**	**0:45:20**	**12**	**227**
2E	**3**	**0:32:07**	**10**	**193**
Average	**3**	**0:41:55**	**12**	**220**
3A	**9**	**0:26:28**	**10**	**159**
3B	**9**	**0:14:56**	**6**	**149**
3C	**9**	**1:16:31**	**22**	**209**
3D	**9**	**0:47:08**	**14**	**202**
3E	**9**	**0:42:26**	**14**	**182**
Average	**9**	**0:41:30**	**12**	**188**

Table 5.3. Time taken to run parallel symbolic regression on multiple processors

First the algorithm was run on a single machine (Run1A-Run1E) and this found the solution in an average time of 2 hours, 2 minutes and 45 seconds. When the algorithm was run on a network of 3 computers, it took only an average of 41 minutes and 55 seconds to find the solution. When 9 processors were used, the increase in speed was minimal and the average time taken was reduced to only 41 minutes and 30 seconds.

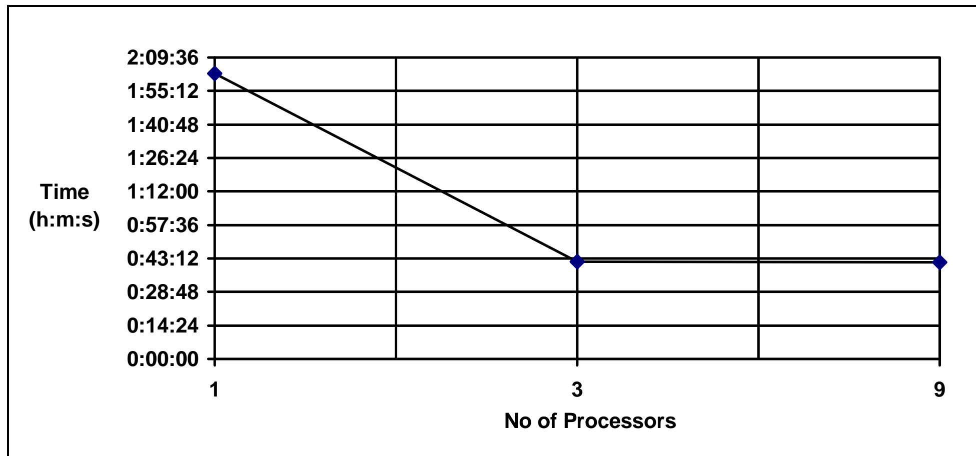

Figure 5.3. Graph showing overall time taken vs. no of processors - Exp 2.1

Figure 5.3 illustrates the differences in time taken during the three runs. There is a substantial decrease in time when the number of processors is increased to 3 but not much improvement gained from increasing the number of processors to 9. This is due to the serial nature of migration. When 9 processors were used, the time taken for evolution was small compared to the time taken for migration. At this point it was decided to parallelise the migration operation as well.

Although the time taken for a complete evolutionary run is significant, it is not the best metric for comparative analysis since the length of each run is most probably different. Thus, when comparing the time taken to reach a solution with different numbers of processors, it is more accurate to use the average times taken to evolve each new generation. Using this data, *Figure 5.4* was generated.

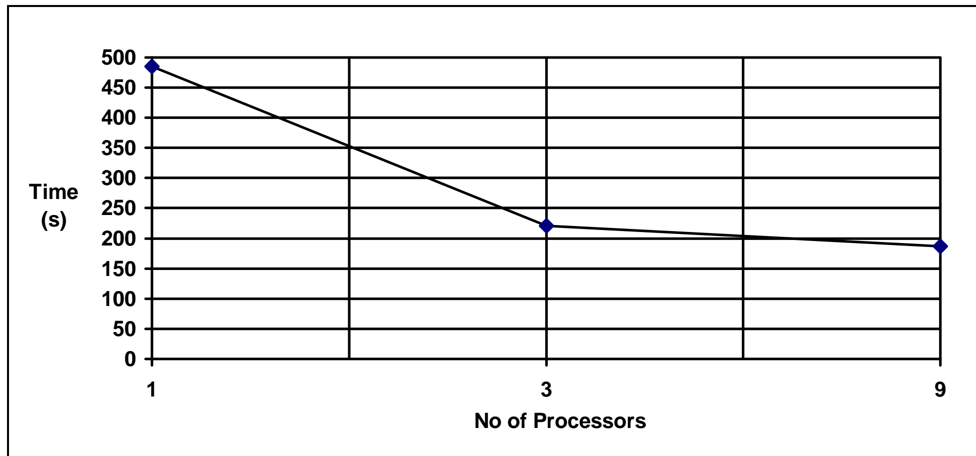

Figure 5.4. Graph showing time taken per generation vs. no of processors - Exp 2.1

It can be seen that the total time taken for each run is related almost proportionately to the time taken for evolution of a single generation.

However, if one compares the average total time taken for 1 processor (2h 02m 45s) to that of 3 processors (41m 45s), it superficially seems that the latter case achieves greater than linear speedup. This is, of course, not the case, since migration and collation of results were still serial operations, resulting in lower than optimal increases in speed. Thus the 3 processors ought to have achieved less than linear speedup of execution. Now, if the average time taken to evolve single generations is used instead, then comparisons can be made between different numbers of processors. The average time taken to process one generation was 485 seconds for 1 processor and 220 seconds for 3 processors. This ratio is below 3:1, as was expected.

Experiment 2.2

In order to prove that the parallel algorithm really does speed up the execution of the algorithm, a single-population model was also tested with all parameters being the same except the number of sub-populations, as indicated in *Table 5.4*.

Parameter	Value
Population Size	**450**
No of Sub-populations	**1**
Max no of Generations	**51**
Max initial size	**5**
Max size	**17**
Maximum complexity	**50**
Min solution fitness	**1**
Mutation probability	**0.1**
Crossover probability	**0.9**
Terminal set	**{x}**
Function set	**{PPlus, PPlus, PTimes, PTimes, PMinus, PDivide}**

Table 5.4. Parameters for parallel symbolic regression - Exp 2.2

The times taken for each run of the experiment is indicated in *Table 5.5.*

Run No	No of Processors	Time Taken to find Solution (h:m:s)	Generations Processed	Time Taken to Process Single Generation (s)
A	**1**	**0:43:33**	**15**	**172**
B	**1**	**0:29:28**	**10**	**177**
C	**1**	**0:27:45**	**10**	**167**
D	**1**	**0:40:07**	**12**	**201**
E	**1**	**0:39:40**	**13**	**183**
Average	**1**	**0:36:00**	**12**	**180**

Table 5.5. Time taken to run single-population symbolic regression on single processor

It was expected that the single-population algorithm would be outperformed by both the 3-processor and 9-processor models. However, the results of the single-population model surpass all models of the parallel algorithm. This occurred primarily because of the serial nature of migration, taking a substantial percentage of the total computation time.

Experiment 2.3

Migration was completely parallelised before these experiments were run. Using 3 processors, Experiment 2.1 was repeated (Run 2A-2E), reverting to the usage of 9 sub-populations. The times taken for these experiments are indicated in *Table 5.6*.

Run No	No of Processors	Time Taken to find Solution (h:m:s)	Generations Processed	Time Taken to Process Single Generation (s)
A	**3**	**0:04:55**	**3**	**98**
B	**3**	**0:14:07**	**7**	**121**
C	**3**	**0:27:21**	**12**	**137**
D	**3**	**0:26:15**	**12**	**131**
E	**3**	**0:11:28**	**6**	**115**
Average	**3**	**0:16:45**	**8**	**120**

Table 5.6. Time taken to run parallel symbolic regression on 3 processors with parallelised migration operation

The average time taken to process a single generation was 120 seconds, which is significantly lower than both the single-population case (experiment 2.2 - 180 seconds) and the serial migration multi-population case (experiment 2.1 - 220s).

Conclusion

Mathematica can successfully be utilised to execute a GP in parallel on a network of workstations. The primary advantage of the parallel implementation is that the restriction on population size and generation numbers is removed. The restrictions of the physical computer can be overcome by appropriately-sized parameters for the parallel algorithm.

In addition, speed-of-execution improvements can be obtained by performing both the evolution and migration operations in parallel. These will be affected by the speed of the server and the ratio of computation time to communication time, as dictated by the number of sub-populations and their sizes.

Experiment 3: CSTR Controller

A Continuous Stirred Tank Reactor (CSTR) is a chemical reactor that was modelled in Mathematica for a simple exothermic reaction [Hajek, 1994]. For some reactions, it is desirable to attain a particular state of the reactor, in terms of the temperature, concentration of reactant and other parameters. With optimal control of the reaction, the chemical reactor may produce maximal yield. It was attempted to control the reactor, by means of changes in coolant and reactant inflow. Hajek applied fuzzy logic, optimised by a genetic algorithm in order to generate equations to control the reactor towards a known unstable steady state. The Mathematica model for this reactor was obtained by personal contact with the author and GP was applied in an attempt to find controlling equations that achieve the objective with as little control deviation as possible.

The fitness function was pre-specified to be the sum of differences between the desired set points and the control variables, temperature and reactant concentration, over a set of discrete time intervals. This summation included four scenarios of the experiment with different starting points (temperature and reactant concentration). This is discussed further in [Hajek, 1994].

The function set contained only the four standard arithmetic operators (Plus, Minus, Times, Divide), to streamline the genetic processes. The terminal set contained the two control variables, temperature ($\boldsymbol{x}$) and concentration of reactant ($\boldsymbol{y}$), as well as some constant values. The parameters used for the GP run are indicated in *Table 5.7.*

Parameter	Value
Population Size	**360**
No of Sub-populations	**9**
Max no of Generations	**51**
Max initial size	**5**
Max size	**17**
Maximum complexity	**50**
Min solution fitness	**1**
Mutation probability	**0.1**
Crossover probability	**0.9**
Terminal set	**{x, x, x, x, x, y, y, y, y, y, 1000, 100, 10, 1, 0.01, 0.001, 0.0001}**
Function set	**{PPlus, PPlus, PTimes, PTimes, PMinus, PDivide}**

Table 5.7. GP Parameters for CSTR

The constants in the terminal set were introduced in order to allow greater scaling of the variables i.e. to increase the range of values spanned by the control variables. Many copies of the control variables (***x*** and ***y***) were included in the terminal set in order to increase the probability of selection of the variables relative to the constants in the same set.

Since two equations were sought, the genetic operators were modified to cater for this. Each individual was generalised to be a list of expressions rather than a single expression. Then, all operations could be applied to the lists. Crossover on a list of expressions was extended to operate on a single expression from the list, chosen with uniform randomness - the corresponding expression is chosen from another individual. Mutation was changed similarly to operate on one of the expressions within the list.

Experiment 3.1

Raw fitness criteria were compared to a supplied heuristic estimate for the control functions, which produced a value of 25337.2. This criterion corresponds to the cumulative error so lower values are indicative of better solutions. The GP algorithm produced comparatively better criteria.

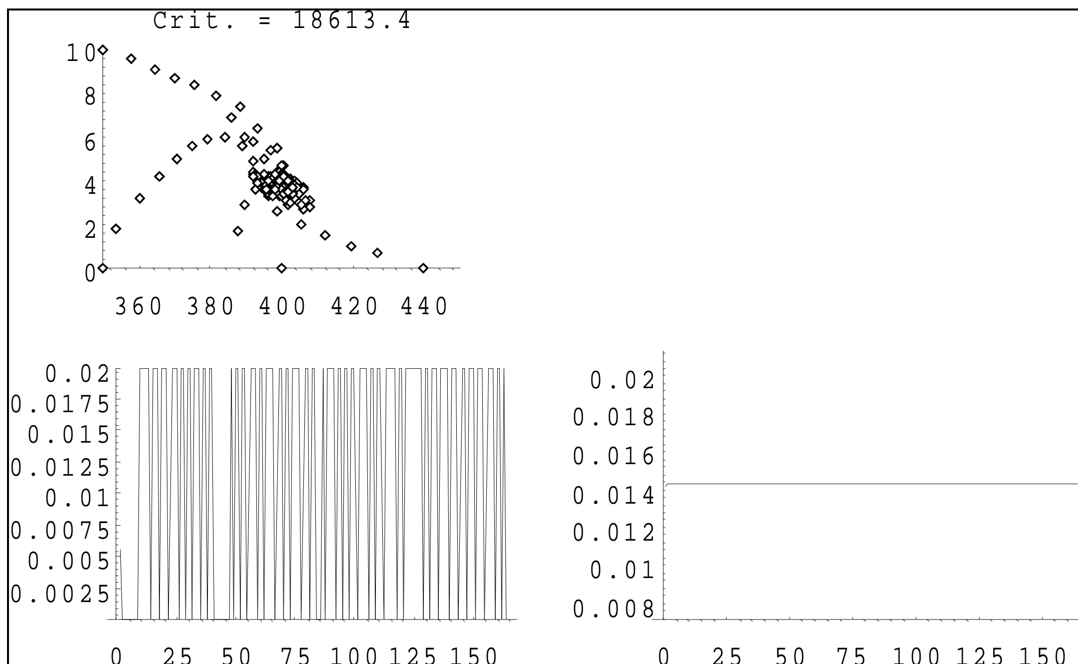

Figure 5.5. Control path for CSTR functions obtained by GP - Exp 3.1

Figure 5.5 shows the control trajectory achieved (top left graph) as well as the values of the control function during the each time interval (coolant inflow on the left and reactant inflow on the right). The criterion was 18613.4, which corresponded to a fitter solution. The control functions generated were:

coolant (x,y,qc) = 10 + 3x + 10y + qc .... *(5.2)*

reactant (x,y,q) = 0.0001 + q .... *(5.3)*

where ***x*** represents the temperature differential, ***y*** represents the concentration differential and ***qc*** and ***q*** are the values of the control functions during previous iterations.

From the graphs of control functions, it is obvious that the control of coolant inflow is not a convergent function but rather an oscillatory one (*bang-bang* control). The controller does not stabilise the reactor during the course of the experiment. If the reaction is continued, there is no guarantee that it will stabilise - the criterion will continue to increase.

Experiment 3.2

The GP algorithm was repeated in order to search for functions which have low fitness in the window of the experiment and generally stabilise the controller as well.

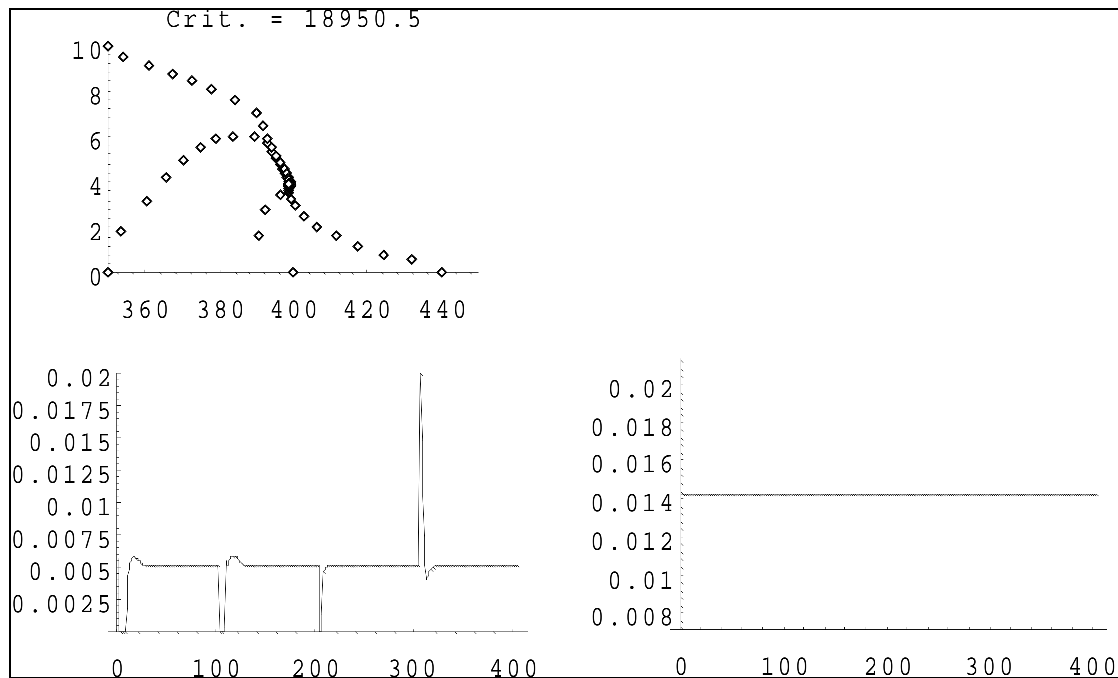

Figure 5.6. Control path for CSTR functions obtained by GP - Exp 3.2

Figure 5.6 indicates such a case, where the criterion is low but the control functions are not oscillatory in nature. The equations generated were:

$$\textit{coolant}\ (x,y,qc) = 0.001\ (\ 1.001 + x + 3y + 0.02\ xy\ ) + qc \quad (5.4)$$

$$\textit{reactant}\ (x,y,q) = 0.0000001 + q \quad \ldots\ldots\ldots \quad (5.5)$$

The variables have the same meanings as discussed above.

Conclusion

In both experiments, the coolant inflow control function is non-linear - it depends on the values of the current concentration as well as the current temperature. Since there are no analytical methods that guarantee the generation of optimal non-linear controllers, GP can be applied to evolve near-optimal controllers. In addition, the

Mathematica implementation is useful in situations such as these where existing problems have already been modelled and alternative solution methodologies are sought.

Experiment 4: PID Controller

A Proportional, Integral, Differential (PID) Controller is another example of a derivative controller for a chemical reactor. The Mathematica model of this controller was obtained by personal contact from M. Hajek. This class of controllers takes as input the current and previous two control deviations (i.e. the differences between the required values and those obtained during the reaction). In order to speed up the generation of equations, only one previous control deviation is used, effectively reducing the controller to a PI controller. The goal of the optimisation was to find a controller that followed a given trajectory, as indicated below.

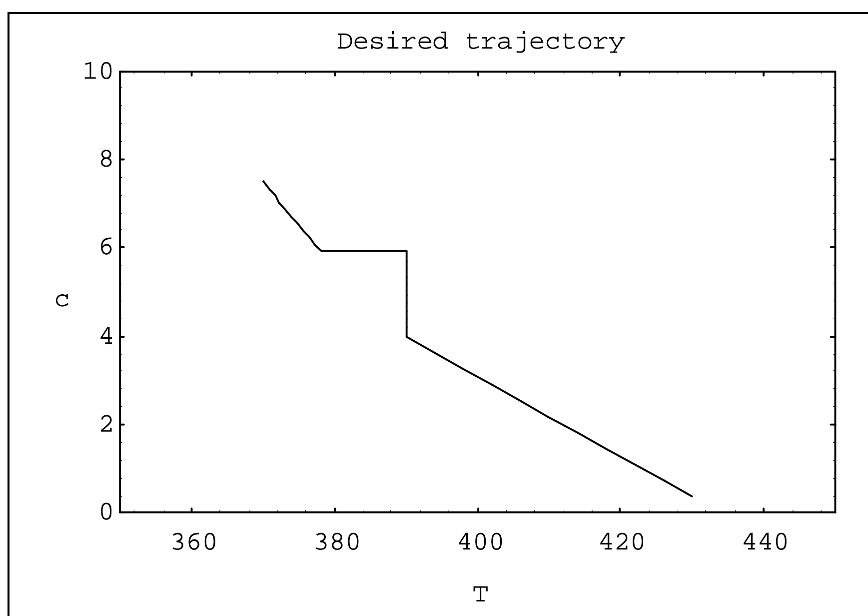

Figure 5.7. Desired control trajectory of PID controller

Figure 5.7 shows the desired trajectory. The horizontal axis represents the temperature while the vertical axis represents the concentration in the reactor. The reaction starts in an initial condition that corresponds to the upper left corner of the given path. The

control equations must thereafter control the reactor so that it follows this path as closely as possible.

GP was applied to this problem using the parameters as indicated in *Table 5.8*.

Parameter	Value
Population Size	**960**
No of Sub-populations	**16**
Max no of Generations	**51**
Max initial size	**5**
Max size	**17**
Maximum complexity	**50**
Min solution fitness	**1**
Mutation probability	**0.1**
Crossover probability	**0.9**
Terminal set	**{dt1, dt1, dt2, dt2, dx1, dx1, dx2, dx2, ec, ec, ec, ec, 100, 10, 1, 0.01, 0.001}**
Function set	**{PPlus, PPlus, PTimes, PTimes, PMinus, PDivide}**

Table 5.8. GP Parameters for PID Controller

In the terminal set, the variables ***dt1*** and ***dt2*** correspond to the current and previous temperature deviations while ***dx1*** and ***dx2*** correspond to the current and previous concentration deviations. ***ec*** is a placeholder to introduce random constants into the algorithm. It is used when generating expressions, and immediately replaced with a random value at each occurrence when the expression is complete. The population size is larger than usual to cater for a sufficient variety of random coefficients.

A pair of given heuristic equations to control the reaction had a criterion of 464.09. These given equations were:

temperature (dt1, dt2, dx1, dx2) = -0.0111792 dx1 + 0.00882075 dx2 *(5.6)*

coolant (dt1, dt2, dx1, dx2) = 0.000637217 dt1 - 0.000502783 dt2 .... *(5.7)*

GP attempted to find sets of equations with a smaller criterion.

Experiment 4.1

The GP algorithm was executed and, after 79 generations, a solution with criterion 1319.15 was found. This solution is not fitter than the given one, but attempts to follow the trajectory by changing the direction of control if the error is large. This results in coarse oscillatory control, and a non-convergent criterion.

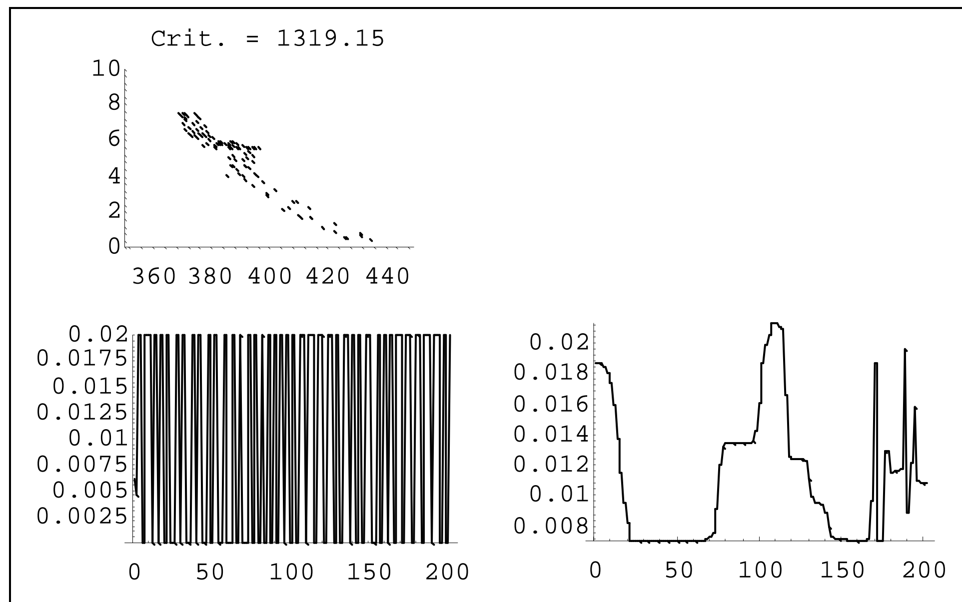

Figure 5.8. Control path for PID controller functions obtained by GP - Exp 4.1

Figure 5.8 indicates the oscillatory nature of the coolant inflow function (bottom left) as well as the discrete trajectory formed by the generated equations (top left).

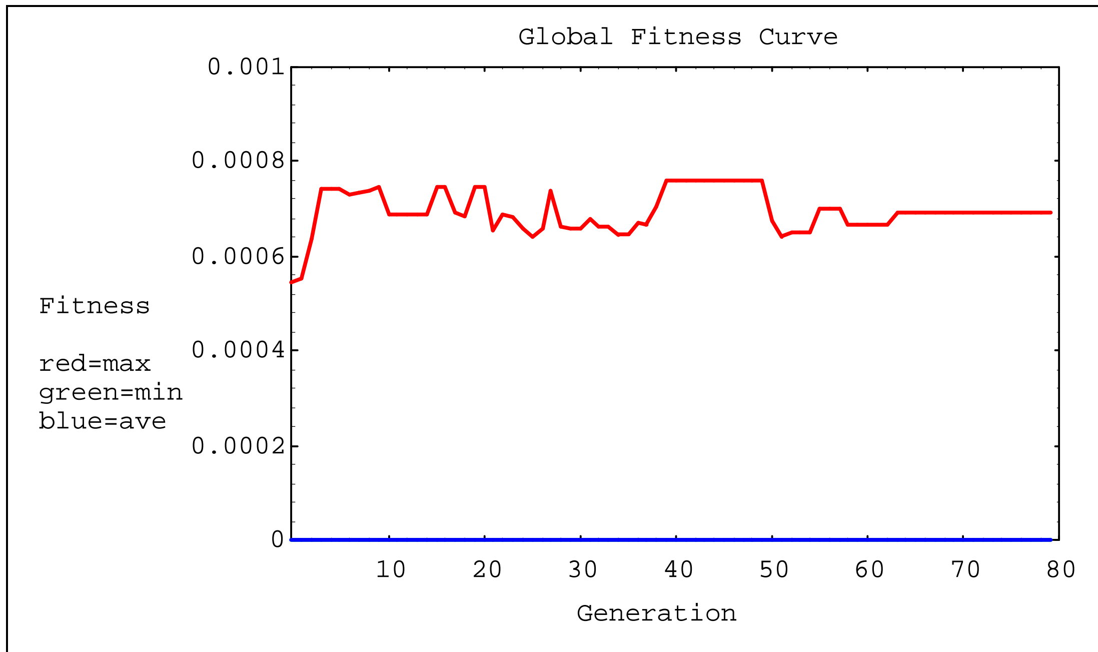

Figure 5.9. Global fitness curve for PID controller - Exp 4.1

Figure 5.9 indicates the global fitness curve for this experiment. It seems from the pattern of evolution that the discovery of a much fitter solution is very unlikely. Repetition of this experiment produced similarly unsatisfactory results, necessitating modification of the parameters.

Experiment 4.2

Since the aim of this experiment was to optimise the control functions, it was decided to incorporate the heuristic functions (*Equations 5.6 and 5.7*) into the initial population, by seeding each sub-population with the pair.

The resulting evolved equations had a noticeably lower criterion than the given equations. *Figure 5.10* indicates the control trajectory and control actions for the set of control functions.

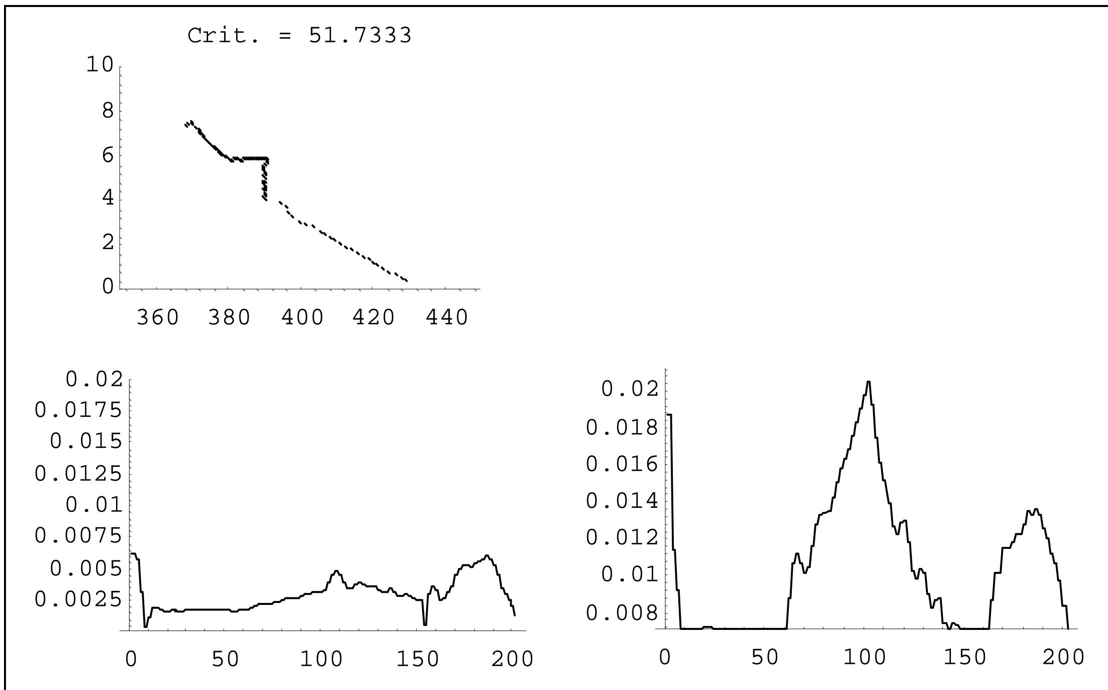

Figure 5.10. Control path for PID controller functions obtained by GP - Exp 4.2 Run 1

The generated controller had a criterion of 51.7333, which is much lower than that of the given equations (*Equation 5.6 and 5.7*). The functions produced by GP were:

***temperature** = 0.0013 dt1 - 0.0005 dt2* .... *(5.8)*

***coolant** =* $-0.0111\times dx1-0.011\left(-1.3971\times10^{-6}-0.011\times dx1+39823\times dx1\left(0.4012+\frac{0.0004\times dt1}{dx2}-8.6981\times10^{-7}(0.0001+dx2)\right)\right)$

.... *(5.9)*

These functions were obviously non-linear and, while still producing a control action similar to the given equations (with no oscillations), incurred less error in the criterion. The experiment was repeated twice and both times the resulting criterion was similar to the first run. *Table 5.9* shows the criteria obtained during the 3 runs of this experiment.

Run	Criterion
1	**51.7333**
2	**54.9929**
3	**55.3069**

Table 5.9. Criteria for PID controllers

The similarity of the criteria for different runs suggests that the solutions obtained are near-optimal for the given problem domain. The global fitness curve for Run 2 is indicated in *Figure 5.11*.

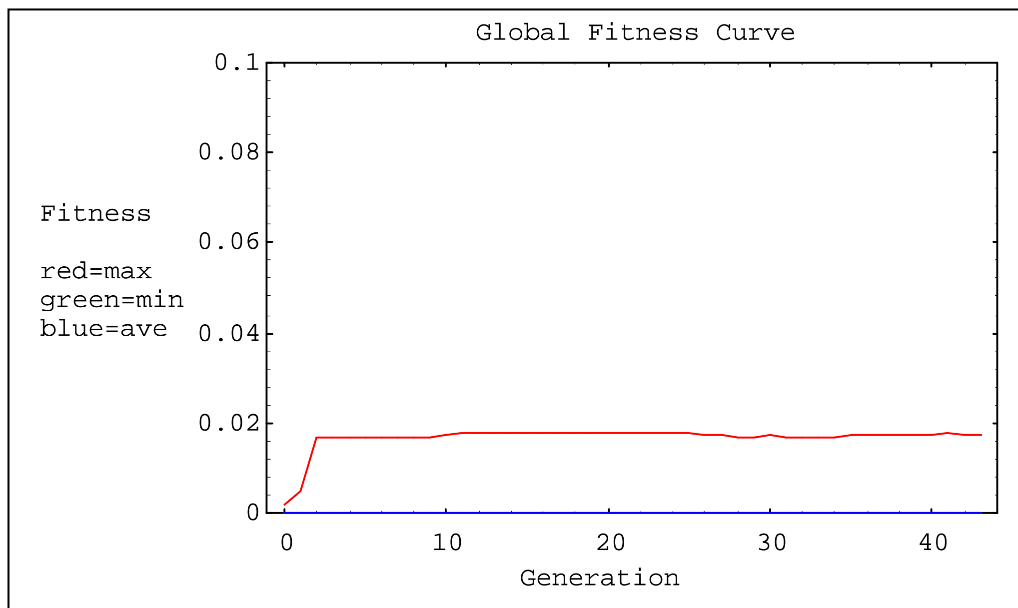

Figure 5.11. Global fitness curve for PID controller - Exp 4.2 Run 2

This global fitness curve is almost identical for all runs of the experiment. The trend suggested by this graph is that the fitness of the best equations will not improve significantly in the following generations.

Experiment 4.3

In a final attempt to further optimise the equations, the initial population was seeded with the given individuals as well as those generated during the three runs of Experiment 4.2. However, this did not result in much improvement in the criterion.

After 73 generations the best individual had a criterion of 49.8443, which is not comparatively much smaller than the criteria from the previous experiment. It is not expected that further runs of the experiment will result in major improvements in the criterion, unless the parameters are changed or the sizing restrictions are relaxed to allow searching of a wider range of solutions.

Conclusion

The non-linear PID controller generated by GP had a lower criterion than the given heuristic equations. GP can be used successfully to take existing equations and evolve better solutions from them. Although GP does not need this problem-specific information, it helps to speed up evolution if as much known information as possible is incorporated into the modelling of the problem domain.

Experiment 5: The Magic Star

Discussion

A magic square is a matrix of numbers with specific properties for its elements e.g. the sum of numbers along each row or column could be equal to the same constant. In terms of a star, similar rules can be applied. Consider the case of a 6-point star, as indicated in *Figure 5.12.*

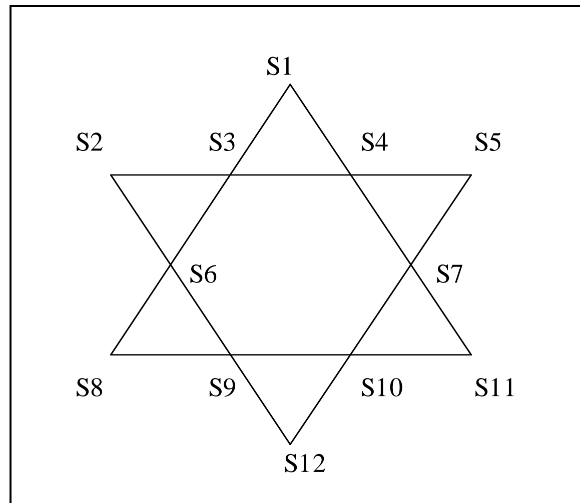

Figure 5.12. Six-point magic star configuration

Figure 5.12 shows the layout of a 6-point star, with each node of the star being assigned a label. One classic magic star problem is to assign the first twelve positive whole integers to the nodes of the star such that the sum of the values at the points is equal to the sum of the values along each line. This can be written as a series of equations, solvable by Gauss-Jordan elimination or similar techniques. The equations would be:

$$
\begin{aligned}
&1+2+\ldots+12 &&= S1+S2+\ldots+S12 \\
&S1 + S2 + S5 + S8 + S11 + S12 &&= Sum \\
&S1 + S4 + S7 + S11 &&= Sum \\
&S1 + S3 + S6 + S8 &&= Sum \\
&S8 + S9 + S10 + S11 &&= Sum \\
&S2 + S3 + S4 + S5 &&= Sum \\
&S2 + S6 + S9 + S12 &&= Sum \\
&S5 + S7 + S10 + S12 &&= Sum \quad \ldots\ldots\ldots\ldots \quad (5.10)
\end{aligned}
$$

For some such problems, it may be known that solutions exist and analytical methods can be employed to find the solution. For other problems, analytical methods may exist but the existence of a solution is not guaranteed. A third class of puzzles has the

property that no solution methods exist. The normal approach to solve such problems would be to identify which class they fall in. Then the solution can be derived analytically, if one exists.

GP was used as an alternative to row-reduction to solve the problem described above.

The problem was modelled in Mathematica, using the parallel GP algorithm as its basis. Since GP produces expressions or programs and the solution being sought was a list of numbers, a conversion of representations was needed. It was decided to model the individuals as permutations that could be applied to a list of 12 numbers. In order to generate the list of numbers represented by an individual, the permutation is applied to (1,2,3,4,5,6,7,8,9,10,11,12) and the resulting list is the solution. Permutations were accomplished by sequences of single-element swaps. These swapping operations were stored in the individual in the form of a tree, that was flattened at evaluation time.

```
SBlock[expr___]:=TestCase[[1]]

Swap[a_, b_]:=Module[
                     {t},
                     t=TestCase[[a]];
                     TestCase[[a]]=TestCase[[b]];
                     TestCase[[b]]=t;
                     TestCase[[12]]
                    ]
```

The function set is composed of **Swap**, which swaps two elements in the list, and **SBlock**, which contains a list of swapping operations. The terminal set contains only random numbers in the range 1-12. The complete list of parameters is listed below in *Table 5.10*.

Parameter	Value
Population Size	**1600**
No of Sub-populations	**16**
Max no of Generations	**51**
Max initial size	**5**
Max size	**17**
Maximum complexity	**50**
Min solution fitness	**1**
Mutation probability	**0.1**
Crossover probability	**0.9**
Terminal set	**{ec, ec, ec, ec}**
Function set	**{sblock, sblock, sblock, sblock, swap, swap, swap, swap}**

Table 5.10. GP Parameters for Magic Star

The fitness function is the only other necessary parameter in order to run the GP algorithm.

```
RawFitness[ex_]:=Module[
                        {summain, sum, diff=0},
                        TestCase=MainCase;
                        ex /. XTrans;
                        summain=Apply[Plus,
                         TestCase[[{1,2,5,8,11,12}]]];
                        sum=Apply[Plus, TestCase[[{1,3,6,8}]]];
                        diff+=Abs[summain-sum];
                        sum=Apply[Plus,
                         TestCase[[{8,9,10,11}]]];
                        diff+=Abs[summain-sum];
                        sum=Apply[Plus, TestCase[[{1,4,7,11}]]];
                        diff+=Abs[summain-sum];
                        sum=Apply[Plus, TestCase[[{2,3,4,5}]]];
                        diff+=Abs[summain-sum];
                        sum=Apply[Plus, TestCase[[{2,6,9,12}]]];
                        diff+=Abs[summain-sum];
                        sum=Apply[Plus,
                         TestCase[[{5,7,10,12}]]];
                        diff+=Abs[summain-sum];
                        diff
                       ]
```

The raw fitness was calculated by first adding together the values at the points of the star, denoted by **summain**. The values along each line are added and these values are then subtracted from **summain**. The differences are gathered together to form the raw fitness. Thus, the fitness function checks an individual to see if it satisfies the criteria of the problem as specified by *Equations 5.10*, and deviations from a perfect solution are penalised proportionately.

The aim of this experiment was to ascertain if GP could solve such a problem with minimum problem-specific information. The GP algorithm was run on a network of 7 486-DX33 computers (6 clients and 1 server).

One run of the algorithm terminated after 72 hours and 237 generations with the perfect solution, which was

```
sblock[4, swap[sblock[sblock[4, swap[5, 8], 2, 6, 12], 4, 4,
 sblock[sblock[6, sblock[7, 6, 6, 9], 6, sblock[10, 6, 2],
 swap[4, 2]],
 1, 6, sblock[4, 5, sblock[12, 6, 9, 6], 6]]], 6],
 swap[swap[11, 4], 6]]
```

When this expression is evaluated, it transforms the **`TestCase`** list into the required set of numbers to assign to the nodes of the star, namely:

{S1, S2, ... , S12} = {6, 4, 3, 11, 8, 12, 7, 5, 9, 10, 2, 1}

The solution is not symmetric so does not lend itself to simple analytical solution methods. Although such methods do exist, it may be easier in some circumstances to model the problem in a prototyping language like Mathematica and execute a GP on it.

Conclusion

All the experiments in this chapter demonstrate the applicability of GP to real-world problems. The advantages of Mathematica modelling are exploited to decrease the setup time and concentrate on the finding of solutions.

Parallelisation of the GP algorithm has the primary advantage of eliminating the constraints that GP placed on memory and computer processing capacity. Thus the sizes of evolved expressions and populations are no longer critical parameters of the GP algorithm. Also, a parallel GP algorithm can be run cooperatively on multiple computers, achieving near-linear speedup of execution. In general, parallelisation of the GP algorithm makes it feasible to solve real-world problems in a prototyping environment like Mathematica.

CONCLUSION

It has been confirmed that Genetic Programming can be useful to solve real-world problems where no analytical solution methodology exists. Symbolic regression is a prime example of such problems and was modelled in both a serial and parallel environment during this study.

Mathematica, already an established mathematical modelling language, was used to implement GP. This implementation took advantage of the extensive function libraries and programming paradigms of Mathematica. It was found that Mathematica is unsuitable for calculations of indeterminate length and time, like GP, due to its internal and temporary storage strategies. Also, Mathematica, being an interpreted language, could never achieve the speed of execution of compiled code. In order to overcome some of these problems, the implementation was parallelised i.e. the GP algorithm was broken into smaller computational segments. The parallel algorithm does not have the restrictions on program parameters which is found in the serial model. Also, speed of execution can be improved in orders of magnitude by executing the algorithm on a network of workstations. This parallel implementation was successfully used to solve some benchmark and real-world optimization problems.

The Mathematica GP implementation is useful because it can be applied to problem domains already modelled in Mathematica. Other problem domains can be modelled in Mathematica with much greater ease than in standard 3GL compiled languages like C++. Although languages like C++ can execute a GP faster than Mathematica, modelling of complex problem domains can be a time-consuming and complicated task. Thus the Mathematica implementation of GP takes advantages of the modelling capabilities of the language. In a non-prototyping production environment, where speed is an important factor, compiled languages would have obvious preference over Mathematica.

Future Directions

Further work could be done on porting the Mathematica implementation to other platforms. Although the functions are platform-independent, the session management

is still based on MS-DOS and compatible operating systems. The scheduler, currently a C++ application, can be written in Java or Mathematica to achieve platform-independence. Scheduling can also be integrated into the algorithm at periodic intervals so that it may optionally be run on a single computer without the need for multi-tasking.

If the code is portable then the algorithm can be executed in parallel on multiple computers networked via the Internet. Data can be shared using Internet-based file system protocols like NFS. Results can be displayed continuously on WWW browsers in the form of Java applets.

The GP algorithm is itself being constantly improved. The Mathematica implementation can be readily extended to cater for changes in GP operators or flow of control. New features like Automatically Defined Functions (ADFs), as discussed exhaustively by Koza, can be easily incorporated since the implementation already caters for multiple sub-expressions within each individual [Koza, 1994].

APPENDIX A : SERIAL ALGORITHM

xtradefs.m

```
ClearAttributes[Divide, Protected]
Divide[_, 0]:=1
SetAttributes[Divide, Protected]

ClearAttributes[Log, Protected]
Log[0]:=0
Log[x_ /; x<0]:=Log[-x]
Log[E^x_]:=x
SetAttributes[Log, Protected]

ClearAttributes[Power, Protected]
Power[0, -1]:=1
SetAttributes[Power, Protected]
```

time.m

```
Time[x_ Second /; x>=3600, Stuff___] :=
  Module[{h, m, s}, s = x; h = Floor[s/3600]; s -= h*3600; m = Floor[s/60];
    s -= m*60; Print[Stuff, h, " Hours, ", m, " Minutes, ", s, " Seconds"]]

Time[x_ Second /; x>=60, Stuff___] :=
  Module[{m, s}, s = x; m = Floor[s/60]; s -= m*60;
    Print[Stuff, m, " Minutes, ", s, " Seconds"]]

Time[x_ Second, Stuff___]:=
  Print[Stuff, x, " Seconds"]
```

genprog.m

```
Get["time.m"]
Get["xtradefs.m"]

(* Terminals *)
(* Functions *)
(* Parameters *)

MaxComplexity=50

(* Generate random expression *)
GenerateNormal[d_]:=Module[
                     {r, Poss, PossPar},
                     If[
                        d>1,
                        Poss=Join[Functions, Terminals];
                        PossPar=Parameters,
                        Poss=Terminals;
                        PossPar={}
                       ];
                     While[
                           Length[PossPar]<Length[Poss],
                           PossPar=Append[PossPar,0]
                          ];
                     r=Random[Integer, {1, Length[Poss]}];
                     Switch[
                            PossPar[[r]],
                            0,
                            Poss[[r]],
                            1,
```

```
                             Poss[[r]][Generate[d-1]],
                             2,
                             Poss[[r]][Generate[d-1], Generate[d-1]],
                             3,
                             Poss[[r]][Generate[d-1], Generate[d-1],
Generate[d-1]]
                            ]
                     ]

Generate[d_]:=Module[
                      {y},
                      y=GenerateNormal[d];
                      While[
                            LeafCount[y]>MaxComplexity,
                            y=GenerateNormal[d]
                           ];
                      y
                     ]

CrossoverProbability=0.9

(* Get list of all indices of internal points in expression *)
RemoveZero[x_]:=If[Position[x, 0]=={}, x, {}]
Points[x_]:=Union[Map[RemoveZero, Position[x, _]], {}]

GetInternal[{x___}]:=x

(* Perform crossover operation on two expressions *)
Cross1[x_, y_]:=Module[
                        {spot1, spot2, point1, point2, temp1, temp2},
                        If[
                           Random[]<CrossoverProbability,
                           point1=Points[x];
                           spot1=Random[Integer, {1, Length[point1]}];
                           point2=Points[y];
                           spot2=Random[Integer, {1, Length[point2]}];
                           temp1=x[[GetInternal[point1[[spot1]]]]];
                           temp2=y[[GetInternal[point2[[spot2]]]]];
                           { If[
                                point1[[spot1]]=={},
                                temp2,
                                ReplacePart[x, temp2, point1[[spot1]]]
                               ],
                             If[
                                point2[[spot2]]=={},
                                temp1,
                                ReplacePart[y, temp1, point2[[spot2]]]
                               ]
                           },
                           {x, y}
                          ]
                       ]

MutationProbability=0.1

MaxSize=17

(* Perform mutation operation on an expression *)
Mutate[x_]:=Module[
                    {spot1, point1, y, xold},
                    xold=x;
                    If[
                       Random[]<MutationProbability,
                       y=Generate[MaxInitialSize];
                       point1=Points[x];
                       spot1=Random[Integer, {1, Length[point1]}];
                       If[
                          point1[[spot1]]=={},
                          y,
                          ReplacePart[x, y, point1[[spot1]]]
                         ],
                       If[
```

```
                         ((Depth[x]<MaxSize) &&
(LeafCount[x]<MaxComplexity)),
                         x,
                         xold
                        ]
                     ]
                  ]

(* RawFitness *)

StandardizedFitness[x_]:=RawFitness[x]

AdjustedFitness[x_]:=N[1/(1+StandardizedFitness[x])]

(* List of fitnesses of expressions in current generation *)
Fitnesses={}

(* Make cumulative fitnesses vector *)
CalcFitnessSum:=Module[{},
   FitSum=Table[Apply[Plus, Take[Fitnesses, i]], {i, 1, Length[Fitnesses]}];
                       FitSum=Insert[FitSum, 0, 1];
                      ]

(* Bisection algorithm search for roulette wheel fitness choice *)
Search[x_] :=
  Module[{Mid, Start=1, Stop=Length[FitSum]},
         While[Start+1 != Stop,
               Mid = Floor[(Start+Stop)/2];
               If[FitSum[[Mid]] > x,
                  Stop=Mid,
                  Start=Mid
                 ]
              ];
         Start
        ]

(* Create new generation from previous one *)
NewGen[x_] := Module[
  {maxwheel, newgen, lenx},
  newgen={};
  maxwheel=Apply[Plus, Fitnesses];
  lenx=Length[x];
  CalcFitnessSum;
  Do[
    Module[
      {spot, index, isum},
      spot=Random[]*maxwheel;
      index=Search[spot];
      newgen=Append[newgen, x[[index]]]
     ],
    {i, 1, lenx}
  ];
  newgen
]

(* Perform crossover on all expressions in new generation *)
Crossover[x_] := Module[
  {newx, oldx, n2, leno, origlen},
  oldx=x;
  newx={};
  leno=Length[oldx];
  origlen=leno;
  While[
    leno>0,
    If[
      leno==1,
      newx=Append[newx, First[oldx]];
      oldx=Rest[oldx],
      n2=Cross1[oldx[[1]], oldx[[2]]];
      If[((Depth[n2[[1]]]<=MaxSize) && (LeafCount[n2[[1]]]<=MaxComplexity)),
         newx=Append[newx, n2[[1]]],
         newx=Append[newx, oldx[[1]]]
```

```
         ];
       If[((Depth[n2[[2]]]<=MaxSize) && (LeafCount[n2[[2]]]<=MaxComplexity)),
          newx=Append[newx, n2[[2]]],
          newx=Append[newx, oldx[[2]]]
         ];
       oldx=Drop[oldx, 2];
     ];
     leno=Length[oldx]
   ];
   newx
]

MaxGenerations=51
PopulationSize=250

(* Solution, SolutionFitness, SolutionSet *)

(* Update best-of-run individual *)
CheckSolution[gen_, x_]:=Module[
                          {minf, maxf},
                          Fitnesses=AdjustedFitness /@ x;
                          minf=Position[Fitnesses, Min[Fitnesses]][[1,1]];
                          maxf=Position[Fitnesses, Max[Fitnesses]][[1,1]];
                          If[
                             SolutionFitness<Fitnesses[[maxf]],
                             Solution=x[[maxf]];
                             SolutionFitness=Fitnesses[[maxf]]
                            ];
                          SolutionSet=Append[SolutionSet,
                           {gen, Fitnesses[[maxf]], x[[maxf]],
                                 Fitnesses[[minf]], x[[minf]]}];
                          Print["G", gen, ": max ", Fitnesses[[maxf]],
                                        "    min ", Fitnesses[[minf]]];
                         ]

MinFitness=0.99

(* Generation, Population, TotTime *)

XTrans={}

(* Apply Genetic algorithm *)
ApplyGen := Module[
  {onetime, poplog},
  Off[Get::noopen];
  Get["restart.log"];
  On[Get::noopen];
  newpop=Population;
(*  Print["G", Generation, ": calculating fitnesses ..."]; *)
(*  Print["G", Generation, ": done ... ", Timing[CheckSolution[Generation,
newpop]][[1]]]; *)

  While[
    (SolutionFitness<MinFitness) && (Generation<MaxGenerations),
   onetime=Timing[
    Print["G", Generation, ": creating mating pool ..."];
    Print["G", Generation, ": done ... ",
Timing[newpop=NewGen[newpop]][[1]]];
    Print["G", Generation, ": performing crossover ..."];
    Print["G", Generation, ": done ... ",
Timing[newpop=Crossover[newpop]][[1]]];
    Print["G", Generation, ": performing mutation ..."];
    Print["G", Generation, ": done ... ", Timing[newpop=Map[Mutate,
newpop]][[1]]];
    Generation++;
    Population=newpop;
    Print["G", Generation, ": calculating fitnesses ..."];
    Print["G", Generation, ": done ... ", Timing[CheckSolution[Generation,
newpop]][[1]]];
    Print["G", Generation, ": best-of-run fitness so far = ",
SolutionFitness];
   ][[1]];
```

```
    Time[onetime, "G", Generation, ": total time for Generation change = "];
    TotTime+=onetime;
    Time[TotTime, "G", Generation, ": total time so far = "];

    Print["Saving state of system..."];
    Save["restart.log", PopulationSize];
    Save["restart.new", ContinueGen];
    RenameFile["restart.log", "restart.old"];
    RenameFile["restart.new", "restart.log"];
    DeleteFile["restart.old"];

    poplog=OpenAppend["pop.log"];
    WriteString[poplog, ","];
    Write[poplog, {Generation, Fitnesses}];
    Close[poplog];
    Print["Finished saving state of system..."];
  ];
  {Solution /. XTrans, SolutionFitness}
]

MaxInitialSize=6

(* Initialise Genetic algorithm *)
Initialize:=Block[
                  {poplog},
                  Population=Table[Generate[MaxInitialSize],
                                   {PopulationSize}];
                  SolutionFitness=0;
                  SolutionSet={};
                  Generation=0;
                  TotTime=0;
                  Print["G", Generation, ": calculating fitnesses ..."];
                  Print["G", Generation, ": done ... ",
                        Timing[CheckSolution[Generation, Population]][[1]]];
                  Print["G", Generation, ": best-of-run fitness so far = ",
                        SolutionFitness];

                  Off[DeleteFile::nffil];
                  DeleteFile["pop.log"];
                  DeleteFile["restart.log"];
                  DeleteFile["restart.new"];
                  DeleteFile["restart.old"];
                  On[DeleteFile::nffil];

                  poplog=OpenAppend["pop.log"];
                  WriteString[poplog, "pop={"];
                  Write[poplog, {Generation, Fitnesses}];
                  Close[poplog];

                  Save["restart.log", PopulationSize];
                  Save["restart.log", ContinueGen];

                  Information[Population];
                  GInformation;
                 ]

(* Start run of algorithm *)
StartGen:=Timing[
                 CheckAbort[
                            ApplyGen,
                            {Solution /. XTrans, SolutionFitness}
                           ]
                ]

ContinueGen[gen_]:=Module[{},
                          MaxGenerations=gen;
                          MinFitness=2;
                          Save["restart.log", MaxGenerations];
                          Save["restart.log", MinFitness];
                          StartGen
                         ]
```

```
GInformation:=Module[{},
                     Print[""];
                     Print["Population Size       : ", PopulationSize];
                     Print["Max no of Generations : ", MaxGenerations];
                     Print["Max initial size      : ", MaxInitialSize];
                     Print["Max size              : ", MaxSize];
                     Print["Min solution fitness  : ", MinFitness];
                     Print["Mutation probability  : ", MutationProbability];
                     Print["Crossover probability : ",
CrossoverProbability];
                     Print["Terminal set          : ", Terminals];
                     Print["Function set          : ", Functions];
                    ]
```

stats.m

```
ShowSample:=ListPlot[MapThread[List, {XPoints, YPoints}]]

ShowCurve:=Module[
                  {t},
                  t=MapThread[List, SolutionSet];
                  ListPlot[MapThread[List, {Join[t[[1]], t[[1]]],
                                     Join[t[[2]], t[[4]]]}],
                                     PlotRange->{{0, 51}, {0, 1}}]
                 ]

ShowSolution:=Plot[Solution /. XTrans, {x, -2, 2}]

ShowFit:=Show[ShowSample, ShowSolution,
              PlotRange->{{-2, 2}, {-2, 10}},
              PlotLabel->Solution /. XTrans, AxesLabel->{x, ""},
              Frame->True
             ]

Stats[s_String]:=Module[{},
                        Display[StringJoin[s, ".sam"], ShowSample];
                        Display[StringJoin[s, ".sol"], ShowSolution];
                        Display[StringJoin[s, ".fit"], ShowFit];
                        Display[StringJoin[s, ".scu"], ShowCurve];
                       ]
```

hist.m

```
<<Graphics`Graphics`
<<Graphics`Animation`

Run["copy pop.log+pop.m pop.ful /Y > nul"]

<<pop.ful

popfit=MapThread[List, pop][[2]]

Histogram[x_, opts___]:=
   Module[{data, fl, figs},
          data=Table[0, {10}];
          figs=Map[Floor, popfit[[x+1]]*10];
          figs=Map[If[#==0, 1, #]&, figs];
          Map[(data[[#]]++)&, figs];
          BarChart[data, BarLabels->Table[i, {i, 0, 0.9, 0.1}],
                   PlotRange->{{0, 11}, {0, PopulationSize}},
                   PlotLabel->StringJoin["Generation ", ToString[x]],
                   opts]
         ]

HistTable:=Table[Histogram[i, DisplayFunction->Identity],
                 {i, 0, Length[pop]-1}]
```

```
AnimateHist:=ShowAnimation[HistTable];
```

restart.m

```
<<xtradefs.m

<<stats.m

<<restart.log
```

APPENDIX B : SCHEDULER

```
#define WIN31

#include <dir.h>

#include <owl.h>

// ------------------------------------------------------------------ \\
// Class declaration for a general item of data in a linked list

class Thing
{
public:
   Thing () {};
   Thing           *Next, *Prev;
};

// ------------------------------------------------------------------ \\
// Class declaration and definition for a general linked list

class ThingList
{
public:
   ThingList ();
   void AddThing ( Thing *p );
   Thing *PopThing ();
protected:
   Thing           *Head, *Tail;
};

ThingList::ThingList ()
{
   Head=NULL;
   Tail=NULL;
}

void ThingList::AddThing ( Thing *p )
{
   p->Prev=Tail;
   if (Head==NULL)
      Head=p;
   else
      Tail->Next=p;
   p->Next=NULL;
   Tail=p;
}

Thing *ThingList::PopThing ()
{
   if (Head==NULL)
      return NULL;
   if (Head==Tail)
   {
      Thing *p=Head;
      Tail=Head=NULL;
      return p;
   }
   else
   {
      Thing *p=Head;
      Head=Head->Next;
      Head->Prev=NULL;
      return p;
   }
}

// ------------------------------------------------------------------ \\
// Declaration and definition for a list of job

class Job : public Thing
```

```
{
public:
   Job ( char *n );
   char *GetName ();
protected:
   char            Name[80];
};

Job::Job ( char *n )
{
   lstrcpy (Name, n);
}

char *Job::GetName ()
{
   return Name;
}

// ---------------------------------------------------------------------- \\
// Declaration and definition for a list of jobs

class JobList : public ThingList
{
public:
   JobList ( PTDialog ptd );
   JobList ( PTDialog ptd, int );
   void Refresh ();
   void AddJob ( Job *p );
protected:
   PTDialog        Parent;
};

JobList::JobList ( PTDialog ptd, int )
{
   Parent=ptd;
}

JobList::JobList ( PTDialog ptd )
{
   struct ffblk    ffblk;
   int          done;

   Parent=ptd;

   done = findfirst("*.*", &ffblk, FA_DIREC);
   while (!done)
   {
      if ((ffblk.ff_name[0]=='P') &&
         (ffblk.ff_name[1]=='O') &&
         (ffblk.ff_name[2]=='P') &&
         (ffblk.ff_name[lstrlen (ffblk.ff_name)-3]=='L') &&
         (ffblk.ff_name[lstrlen (ffblk.ff_name)-2]=='O') &&
         (ffblk.ff_name[lstrlen (ffblk.ff_name)-1]=='G') &&
         (ffblk.ff_name[3]!='.'))
      {
         ffblk.ff_name[lstrlen (ffblk.ff_name)-4]=0;
        Job *p=new Job (ffblk.ff_name);
        AddJob (p);
      }
      done = findnext(&ffblk);
   }

   Refresh ();
}

void JobList::Refresh ()
{
   Job             *p=(Job *)Head;

   Parent->SendDlgItemMsg (102, LB_RESETCONTENT, 0, 0);
   while (p!=NULL)
   {
```

```
      if (p!=NULL)
        Parent->SendDlgItemMsg (102, LB_ADDSTRING, 0, (long)(p->GetName()));
      p=(Job *)p->Next;
   }
}

void JobList::AddJob ( Job *p )
{
   AddThing (p);
}

// -------------------------------------------------------------------- \\
// Declaration and definition for a list of migration jobs

class MigrateJobList
{
public:
   MigrateJobList ( PTDialog ptd, int nos );
   ~MigrateJobList ();
   void AddJob ( char *s );
   void Refresh ();
   char *GetJob ();
   BOOL MoreJobs ();
   void ClearJob ( char *s );
protected:
   unsigned char   *Matrix, *List;
   PTDialog        Parent;
   unsigned long   Size;
   char            tJob[256];
   unsigned long GetPos ( int r, int c );
};

MigrateJobList::MigrateJobList ( PTDialog ptd, int nos )
{
   Parent=ptd;
   Size=nos;
   Matrix=new unsigned char [GetPos (nos-1, nos)+1];
   memset (Matrix, 0, GetPos (nos-1, nos)+1);
   List=new unsigned char [nos];
   memset (List, 0, nos);
}

MigrateJobList::~MigrateJobList ()
{
   delete Matrix;
   delete List;
}

unsigned long MigrateJobList::GetPos ( int r, int c )
{
   unsigned long   Pos, r1=r, c1=c;
   Pos=(((r1-1)*(2*(Size-1)-r1+2))/2)+c1-r1-1;
   return Pos;
}

void MigrateJobList::AddJob ( char *s )
{
   s++;
   unsigned long Code=atol (s);
   unsigned long r=(Code / Size)+1;
   unsigned long c=(Code % Size)+1;
   Matrix[GetPos (r, c)]=1;
}

char *MigrateJobList::GetJob ()
{
   for ( int a=1; a<Size; a++ )
      for ( int b=a+1; b<=Size; b++ )
        if ((List[a-1]==0) && (List[b-1]==0) && (Matrix[GetPos (a, b)]==1))
        {
           List[a-1]=1;
           List[b-1]=1;
```

```
            Matrix[GetPos (a, b)]=0;
            char   u[256];
            lstrcpy (tJob, "M");
            ultoa ((a-1)*Size+(b-1), u, 10);
            lstrcat (tJob, u);
            return tJob;
         }
   return NULL;
}

BOOL MigrateJobList::MoreJobs ()
{
   if (memchr (Matrix, 1, GetPos (Size-1, Size)+1)==NULL)
      return FALSE;
   else
      return TRUE;
}

void MigrateJobList::ClearJob ( char *s )
{
   s++;
   unsigned long Code=atol (s);
   unsigned long r=(Code / Size)+1;
   unsigned long c=(Code % Size)+1;
   List[r-1]=0;
   List[c-1]=0;
}

void MigrateJobList::Refresh ()
{
   Parent->SendDlgItemMsg (102, LB_RESETCONTENT, 0, 0);
   for ( int a=1; a<Size; a++ )
      for ( int b=a+1; b<=Size; b++ )
        if (Matrix[GetPos (a, b)]==1)
        {
            char   u[256];
            lstrcpy (tJob, "M");
            ultoa ((a-1)*Size+(b-1), u, 10);
            lstrcat (tJob, u);
            Parent->SendDlgItemMsg (102, LB_ADDSTRING, 0, (long)tJob);
        }
}

// ------------------------------------------------------------------ \\
// Declaration and definition for a processor

class Processor : public Thing
{
public:
   Processor ( char *n );
   char *GetJob ();
   void SetJob ( char *s );
   char *GetName ();
   char *GetCurrentJob ();
   void KillCurrentJob ();
protected:
   char            Name[80];
   char            aJob[80];
   char            TempJob[80];
};

Processor::Processor ( char *n )
{
   lstrcpy (Name, n);
}

char *Processor::GetJob ()
{
   struct ffblk    ffblk;
   int           done;
   char            attr[256];
```

```
   lstrcpy (attr, Name);
   lstrcat (attr, "\\*.*");
   done = findfirst(attr, &ffblk, 0);
   lstrcpy (TempJob, ffblk.ff_name);

   if (done)
      return NULL;
   else
      return TempJob;
}

void Processor::SetJob ( char *s )
{
   char            t[100];

   lstrcpy (t, Name);
   lstrcat (t, "\\");
   lstrcat (t, s);
   HFILE f=_lcreat (t, 0);
   _lclose (f);

   lstrcpy (aJob, s);
}

char *Processor::GetName ()
{
   return Name;
}

char *Processor::GetCurrentJob ()
{
   return aJob;
}

void Processor::KillCurrentJob ()
{
   lstrcpy (aJob, "idle");
}

// ------------------------------------------------------------------- \\
// Declaration and definition for a list of processors

class ProcessorList : public ThingList
{
public:
   ProcessorList ( PTDialog ptd );
   void Start ();
   void Refresh ();
   BOOL            RunComplete, GenComplete;
protected:
   PTDialog        Parent;
   void AddProcessor ( Processor *p );
   JobList         *jl;
   MigrateJobList  *ml;
   enum            { Processing, Checking, Migrating } RState;
   void RefreshProcessing ();
   void RefreshMigrating ();
   void RefreshChecking ();
};

// find all processors, initialise job list and start processing
ProcessorList::ProcessorList ( PTDialog ptd )
{
   struct ffblk    ffblk;
   int          done;

   GenComplete=RunComplete=FALSE;

   Parent=ptd;

   done = findfirst("*.*", &ffblk, FA_DIREC);
   while (!done)
```

```
   {
      if (((ffblk.ff_attrib & FA_DIREC)>0) &&
          (ffblk.ff_name[0]=='P') &&
         (ffblk.ff_name[1]=='R') &&
         (ffblk.ff_name[2]=='O') &&
         (ffblk.ff_name[3]=='C'))
      {
         Processor *p=new Processor (ffblk.ff_name);
        AddProcessor (p);
      }
      done = findnext(&ffblk);
   }

   jl=new JobList (Parent);

   Start ();
}

// create job lists and assign tasks to each processor
void ProcessorList::Start ()
{
   Processor       *p=(Processor *)Head;
   Job             *aJob;
   char            t[256];
   char            Generation[10];
   char            SolutionFitness[256];
   int             NoOfMigrationPairs;
   int           NoOfSubpopulations;
   char            MigrationPair[20];

   Parent->SendDlgItemMsg (101, LB_RESETCONTENT, 0, 0);
   Parent->SendDlgItemMsg (102, LB_RESETCONTENT, 0, 0);
   Parent->SendDlgItemMsg (103, LB_RESETCONTENT, 0, 0);

   while (p!=NULL)
   {
      aJob=(Job *)jl->PopThing ();
      if (aJob!=NULL)
      {
        p->SetJob (aJob->GetName ());
        lstrcpy (t, p->GetName ());
        lstrcat (t, "::");
         lstrcat (t, aJob->GetName ());
        Parent->SendDlgItemMsg (101, LB_ADDSTRING, 0, (long)t);
        p=(Processor *)p->Next;
        delete aJob;
      }
      else
         p=NULL;
   }

   jl->Refresh ();
   RState=Processing;

   fstream f ("pop.inf", ios::in);
   f >> Generation;
   f >> SolutionFitness;
   f >> NoOfSubpopulations;
   f >> NoOfMigrationPairs;

   ml=new MigrateJobList (Parent, NoOfSubpopulations);

   for ( int a=0; a<NoOfMigrationPairs; a++ )
   {
      f >> MigrationPair;
      ml->AddJob (MigrationPair);
   }

   Parent->SendDlgItemMsg (301, WM_SETTEXT, 0, (long)Generation);
   Parent->SendDlgItemMsg (302, WM_SETTEXT, 0, (long)SolutionFitness);
}
```

```
// refresh job status for each processor
void ProcessorList::Refresh ()
{
   if (RState==Processing)
      RefreshProcessing ();
   else if (RState==Checking)
      RefreshChecking ();
   else
      RefreshMigrating ();
}

// add processor to list
void ProcessorList::AddProcessor ( Processor *p )
{
   AddThing (p);
}

// refresh evolution jobs for each processor and update display
void ProcessorList::RefreshProcessing ()
{
   Processor       *p=(Processor *)Head;
   char            *aJobName;
   BOOL            StillComputing=FALSE, JobChange=FALSE;
   char            t[256];
   Job             *aJob;

   Parent->SendDlgItemMsg (101, LB_RESETCONTENT, 0, 0);
   while (p!=NULL)
   {
      aJobName=p->GetJob ();
      if (aJobName!=NULL)
      {
        StillComputing=TRUE;
        lstrcpy (t, p->GetName ());
        lstrcat (t, "::");
         lstrcat (t, p->GetCurrentJob ());
        Parent->SendDlgItemMsg (101, LB_ADDSTRING, 0, (long)t);
      }
      else
      {
        aJob=(Job *)jl->PopThing ();
        if (aJob!=NULL)
         {
           Parent->SendDlgItemMsg (103, LB_ADDSTRING, 0, (long)(p-
>GetCurrentJob ()));
           p->SetJob (aJob->GetName ());
           lstrcpy (t, p->GetName ());
           lstrcat (t, "::");
           lstrcat (t, aJob->GetName ());
           Parent->SendDlgItemMsg (101, LB_ADDSTRING, 0, (long)t);
           delete aJob;
           JobChange=TRUE;
           StillComputing=TRUE;
         }
        else
        {
           if (lstrcmp (p->GetCurrentJob (), "idle")!=0)
            {
              Parent->SendDlgItemMsg (103, LB_ADDSTRING, 0, (long)(p-
>GetCurrentJob ()));
              p->KillCurrentJob ();
            }
           lstrcpy (t, p->GetName ());
           lstrcat (t, "---idle");
           Parent->SendDlgItemMsg (101, LB_ADDSTRING, 0, (long)t);
        }
      }
      p=(Processor *)p->Next;
   }
   if (JobChange)
      jl->Refresh ();
   if (!StillComputing)
```

```
   {
      ((Processor *)(Head))->SetJob ("MSTART");
      lstrcpy (t, ((Processor *)(Head))->GetName ());
      lstrcat (t, "::MSTART");
      Parent->SendDlgItemMsg (101, LB_RESETCONTENT, 0, 0);
      Parent->SendDlgItemMsg (101, LB_ADDSTRING, 0, (long)t);
      delete jl;
      RState=Checking;
   }
}

// check for best solution
void ProcessorList::RefreshChecking ()
{
   char            *CurrJob;
   CurrJob=((Processor *)(Head))->GetJob ();

   if (CurrJob==NULL)
   {
      RState=Migrating;
      ml->Refresh ();
   }
   else if (lstrcmpi (CurrJob, "DONE")==0)
   {
      delete ml;
      RunComplete=TRUE;
   }
}

// refresh migration jobs for each processor and update display
void ProcessorList::RefreshMigrating ()
{
   Processor       *p=(Processor *)Head;
   char            *aJobName;
   BOOL            StillComputing=FALSE, JobChange=FALSE;
   char            t[256];
   char            *aJob;

   Parent->SendDlgItemMsg (101, LB_RESETCONTENT, 0, 0);
   while (p!=NULL)
   {
      aJobName=p->GetJob ();
      if (aJobName!=NULL)
      {
        StillComputing=TRUE;
        lstrcpy (t, p->GetName ());
        lstrcat (t, "::");
         lstrcat (t, p->GetCurrentJob ());
        Parent->SendDlgItemMsg (101, LB_ADDSTRING, 0, (long)t);
      }
      else
      {
        if (lstrcmp (p->GetCurrentJob (), "idle")!=0)
           ml->ClearJob (p->GetCurrentJob ());
        aJob=ml->GetJob ();
        if (aJob!=NULL)
         {
           if (lstrcmpi (p->GetCurrentJob (), "idle")!=0)
              Parent->SendDlgItemMsg (103, LB_ADDSTRING, 0, (long)(p-
>GetCurrentJob ()));
           p->SetJob (aJob);
           lstrcpy (t, p->GetName ());
           lstrcat (t, "::");
           lstrcat (t, aJob);
           Parent->SendDlgItemMsg (101, LB_ADDSTRING, 0, (long)t);
           JobChange=TRUE;
           StillComputing=TRUE;
         }
        else
        {
           if (lstrcmp (p->GetCurrentJob (), "idle")!=0)
           {
```

```
              Parent->SendDlgItemMsg (103, LB_ADDSTRING, 0, (long)(p-
>GetCurrentJob ()));
              p->KillCurrentJob ();
             }
           lstrcpy (t, p->GetName ());
           lstrcat (t, "---idle");
           Parent->SendDlgItemMsg (101, LB_ADDSTRING, 0, (long)t);
          }
      }
      p=(Processor *)p->Next;
   }
   if (JobChange)
      ml->Refresh ();
   if (!StillComputing)
   {
      if (ml->MoreJobs ()==FALSE)
      {
         delete ml;
        GenComplete=TRUE;
      }
   }
}

// ----------------------------------------------------------------------- \\
// Declaration and definition of Windows Interface

class MDialog : public TDialog
{
public:
   MDialog ( PTWindowsObject AParent, LPSTR AName, PTModule AModule=NULL );
   virtual void IdleAction ();
protected:
   BOOL            Stopped;
   DWORD        TotTime, StartTime;
   virtual LPSTR GetClassName () { return "GPNetDialog"; };
   virtual void HandleExit ( RTMessage ) = [ ID_FIRST + 201 ];
   virtual void Start ( RTMessage ) = [ ID_FIRST + 202 ];
   virtual void Stop ( RTMessage ) = [ ID_FIRST + 203 ];
   ProcessorList   *pl;
   void MakeTime ( DWORD t, char *s );
};

MDialog::MDialog ( PTWindowsObject AParent, LPSTR AName, PTModule AModule) :
  TDialog (AParent, AName, AModule)
{
   pl=NULL;
   Stopped=FALSE;
   TotTime=0;
};

void MDialog::IdleAction ()
{
   if (pl!=NULL)
   {
      pl->Refresh ();
      if (pl->GenComplete==TRUE)
      {
        TotTime+=GetTickCount ()-StartTime;
        delete pl;
        if (Stopped==FALSE)
          {
             StartTime=GetTickCount ();
           pl=new ProcessorList (this);
          }
        else
           pl=NULL;
      }
      else if (pl->RunComplete==TRUE)
      {
        TotTime+=GetTickCount ()-StartTime;
        delete pl;
         pl=NULL;
```

```
      }
      else
      {
        char      s[256];
        DWORD     t=TotTime+(GetTickCount ()-StartTime);
          MakeTime (t, s);
          SendDlgItemMsg (303, WM_SETTEXT, 0, (long)s);
      }
   }
}

void MDialog::HandleExit ( RTMessage )
{
   CloseWindow ();
}

void MDialog::Start ( RTMessage )
{
   if (pl==NULL)
   {
      StartTime=GetTickCount ();
      pl=new ProcessorList (this);
      Stopped=FALSE;
   }
}

void MDialog::Stop ( RTMessage )
{
   if (pl!=NULL)
      Stopped=TRUE;
}

void MDialog::MakeTime ( DWORD t, char *s )
{
   DWORD           secs, mins, hours;

   t=t/1000;
   hours=t/3600;
   t-=hours*3600;
   mins=t/60;
   t-=mins*60;
   secs=t;
   wsprintf (s, "H: %lu M: %lu S: %lu", hours, mins, secs);
}

// ------------------------------------------------------------------ \\
// Declaration and definition of Application container

class MApplication : public TApplication
{
public:
   MApplication ( LPSTR AName, HINSTANCE AnInstance, HINSTANCE
APrevInstance,
               LPSTR ACmdLine, int ACmdShow ) :
     TApplication (AName, AnInstance, APrevInstance, ACmdLine, ACmdShow) {};
protected:
   virtual void InitMainWindow ();
   virtual void IdleAction ();
   DWORD           TickTimer;
};

void MApplication::InitMainWindow ()
{
   MainWindow = new MDialog (NULL, "GPNetDialog");
   TickTimer=GetTickCount ();
}

void MApplication::IdleAction ()
{
   if ((GetTickCount ()-TickTimer)>=1000)
   {
      ((MDialog *)(MainWindow))->IdleAction ();
```

```
      TickTimer=GetTickCount ();
   }
}

// ------------------------------------------------------------------ \\
// ------------------------------------------------------------------ \\
// Main program body
// ------------------------------------------------------------------ \\

int PASCAL WinMain ( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  LPSTR lpCmdLine, int nCmdShow )
{
   MApplication M ("GPNet", hInstance, hPrevInstance, lpCmdLine, nCmdShow);
   M.Run ();
   return M.Status;
}
```

APPENDIX C : PARALLEL GP

time.m

```
(* Genetic Programming *)

(* Time output routines *)

(* H. Suleman *)
(* 24 October 1995 *)

BeginPackage["Genetic`Time`"]

Time::usage = "Time[x] outputs the time taken in seconds, minutes and hours.
               Time[x, Stuff] outputs Stuff followed by time taken."

Begin["`Private`"]

Time[x_ Second /; x>=3600, Stuff___] :=
  Module[{h, m, s}, s = x; h = Floor[s/3600]; s -= h*3600; m = Floor[s/60];
    s -= m*60; Print[Stuff, h, " Hours, ", m, " Minutes, ", s, " Seconds"]]

Time[x_ Second /; x>=60, Stuff___] :=
  Module[{m, s}, s = x; m = Floor[s/60]; s -= m*60;
    Print[Stuff, m, " Minutes, ", s, " Seconds"]]

Time[x_ Second, Stuff___]:=
  Print[Stuff, x, " Seconds"]

End[]

Protect[Time]

EndPackage[]
```

xtradefs.m

```
(* Genetic Programming *)

(* Extra definition routines *)

(* H. Suleman *)
(* 24 October 1995 *)

BeginPackage["Genetic`ExtraDefinitions`"]
EndPackage[]

ClearAttributes[Divide, Protected]
Divide[_, 0]:=1
SetAttributes[Divide, Protected]

ClearAttributes[Mod, Protected]
Mod[_, 0]:=0
SetAttributes[Mod, Protected]

ClearAttributes[Log, Protected]
Log[0]:=0
Log[x_ /; x<0]:=Log[-x]
(* Log[E^x_]:=x *)
SetAttributes[Log, Protected]

ClearAttributes[Power, Protected]
Power[0, x_ /; x<0]:=1
SetAttributes[Power, Protected]
```

```
ClearAttributes[Unequal, Protected]
Unequal[_String, EndOfFile]:=True
SetAttributes[Unequal, Protected]
```

default.m

```
(* Genetic Programming *)

(* Default parameters and user-defined functions *)

(* H. Suleman *)
(* 9 June 1996 *)

BeginPackage["Genetic`Parameters`"]

MaxComplexity = 50

PopulationSize = 40

MaxInitialSize = 5

NoOfSubpopulations = 4

MaxGenerations = 51

MaxSize = 17

MinFitness = 0.99

CrossoverProbability = 0.9

MutationProbability = 0.1

MigrationPercentage = 0.1

MigrationDeviation = 0.05

MigrationProbability = 4

Epoch=20

LengthOfMember = 1

XTrans={PPlus->Plus, PMinus->Minus, PTimes->Times, PDivide->Divide, PMod-
>Mod}

Functions={PPlus, PTimes, PMinus, PDivide}

Parameters={2, 2, 1, 2}

Terminals={ec}

ReTouch[expr_]:=expr /. ec:>Random[Real, {1, 10}]

RawFitness[expr_]:=N[((expr /. XTrans)-Sqrt[2])^2]

StandardizedFitness[expr_]:=RawFitness[expr]

AdjustedFitness[expr_]:=Module[
                                {answer},
                                answer=N[1/(1+StandardizedFitness[expr])];
                                AdjustedFitness[expr]=answer;
                                answer
                               ]

EndPackage[]
```

operator.m

```
(* Genetic Programming *)

(* Genetic operator routines *)

(* H. Suleman *)
(* 9 June 1996 *)

(* Get parameters *)
Needs["Genetic`Parameters`", "default.m"]

BeginPackage["Genetic`Operators`"]

Cross1::usage = "Cross1[x, y] performs crossover on x and y to produce {x1,
y1}."

Crossover::usage = "Crossover[x] performs crossover on the population in
list x."

Mutate::usage = "Mutate[x] randomly mutates expression x."

Begin["`Private`"]

(* Get list of all indices of internal points in expression *)
RemoveZero[x_]:=If[Position[x, 0]=={}, x, {}]
Points[x_]:=Union[Map[RemoveZero, Position[x, _]], {}]
GetInternal[{x___}]:=x

(* Perform crossover operation on two expressions *)
Cross1[x_, y_]:=Module[
                       {spot1, spot2, point1, point2, temp1, temp2},
                       If[
                          Random[]<Genetic`Parameters`CrossoverProbability,
                          point1=Points[x];
                          spot1=Random[Integer, {1, Length[point1]}];
                          point2=Points[y];
                          spot2=Random[Integer, {1, Length[point2]}];
                          temp1=x[[GetInternal[point1[[spot1]]]]];
                          temp2=y[[GetInternal[point2[[spot2]]]]];
                          { If[
                               point1[[spot1]]=={},
                               temp2,
                               ReplacePart[x, temp2, point1[[spot1]]]
                              ],
                            If[
                               point2[[spot2]]=={},
                               temp1,
                               ReplacePart[y, temp1, point2[[spot2]]]
                              ]
                          },
                          {x, y}
                         ]
                      ]

(* perform crossover on corresponding elements in lists *)
Cross1[x_ /; Head[x]==List, y_ /; Head[y]==List]:=
   Module[
          {z, pos, xnew, ynew},
          pos=Random[Integer, {1, Length[x]}];
          z=Cross1[x[[pos]], y[[pos]]];
          xnew=x;
          ynew=y;
          xnew[[pos]]=z[[1]];
          ynew[[pos]]=z[[2]];
          {xnew, ynew}
         ]

(* Perform mutation operation on an expression *)
Mutate[x_]:=Module[
                   {spot1, point1, y, xold, xnew},
```

```
                    xold=x;
                    xnew=x;
                    If[
                       Random[]<Genetic`Parameters`MutationProbability,
                       y=Genetic`Initialization`Generate[Random[Integer, {1,
Genetic`Parameters`MaxInitialSize}]];
                       point1=Points[xnew];
                       spot1=Random[Integer, {1, Length[point1]}];
                       xnew=If[
                               point1[[spot1]]=={},
                               y,
                               ReplacePart[x, y, point1[[spot1]]]
                              ];
                       If[
                          ((Depth[xnew]<=Genetic`Parameters`MaxSize) &&

(LeafCount[xnew]<=Genetic`Parameters`MaxComplexity)),
                          xnew,
                          xold
                         ],
                       xold
                      ]
                   ]

(* perform mutation on an element within a list *)
Mutate[x_ /; Head[x]==List]:=
   Module[
          {z, xnew},
          z=Random[Integer, {1, Length[x]}];
          xnew=x;
          xnew[[z]]=Mutate[x[[z]]];
          xnew
         ]

(* Perform crossover on all expressions in new generation *)
Crossover[x_] := Module[
  {newx, oldx, n2, leno, origlen},
  oldx=x;
  newx={};
  leno=Length[oldx];
  origlen=leno;
  While[
    leno>0,
    If[
      leno==1,
      newx=Append[newx, First[oldx]];
      oldx=Rest[oldx],
      n2=Cross1[oldx[[1]], oldx[[2]]];
      If[((Depth[n2[[1]]]<=Genetic`Parameters`MaxSize) &&
          (LeafCount[n2[[1]]]<=Genetic`Parameters`MaxComplexity)),
         newx=Append[newx, n2[[1]]],
         newx=Append[newx, oldx[[1]]]
        ];
      If[((Depth[n2[[2]]]<=Genetic`Parameters`MaxSize) &&
          (LeafCount[n2[[2]]]<=Genetic`Parameters`MaxComplexity)),
         newx=Append[newx, n2[[2]]],
         newx=Append[newx, oldx[[2]]]
        ];
      oldx=Drop[oldx, 2];
    ];
    leno=Length[oldx]
  ];
  newx
]

End[]

Protect[Cross1, Crossover, Mutate]

EndPackage[]
```

initial.m

```
(* Genetic Programming *)

(* Initialization routines *)

(* H. Suleman *)
(* 9 June 1996 *)

(* Get time routines *)
Needs["Genetic`Time`", "time.m"]

(* Get extra definitions for basic arithmetic operations *)
Needs["Genetic`ExtraDefinitions`", "xtradefs.m"]

(* Get parameters *)
Needs["Genetic`Parameters`", "default.m"]

(* Get file locking routines *)
Needs["Genetic`Shares`", "shares.m"]

BeginPackage["Genetic`Initialization`", {"Genetic`Parameters`"}]

Generate::usage = "Generate[x] generates a random expression of depth x."

Initialize::usage = "Initialize initialises the various parameters and
populations."

GInformation::usage = "GInformation[] lists information about the current
parameters."

GPopInformation::usage = "GPopInformation[popname] lists information about
the state and best individual in the current population."

CheckSolution::usage = "CheckSolution calculates fitnesses and checks for
solutions."

CheckGlobalSolutions::usage = "CheckGlobalSolutions checks if the local
solution betters the global one."

InitNames::usage = "InitNames initialises the table of name prefixes of
populations."

Begin["`Private`"]

(* Make lists of terminals+functions, parameters, etc. *)
MakePossibilities:=Module[
                          {},
                          Genetic`Parameters`GPossibilities=Join[Terminals,
                                                Functions];
                          Genetic`Parameters`GPossParameter=Join[
                                                Table[0, {Length[Terminals]}],
                                                Parameters
                                               ];

Genetic`Parameters`GPossLength=Length[Genetic`Parameters`GPossibilities];
                          Genetic`Parameters`GTermLength=Length[Terminals];
                         ]

(* Generate random expression *)
GenerateNormal[d_]:=Module[
                     {r},
                     If[
                        d>1,
                        r=Random[Integer, {1,
Genetic`Parameters`GPossLength}],
                        r=Random[Integer, {1,
Genetic`Parameters`GTermLength}]
                       ];
                     Switch[
                            Genetic`Parameters`GPossParameter[[r]],
```

```
                              0,
                              Genetic`Parameters`GPossibilities[[r]],
                              1,

Genetic`Parameters`GPossibilities[[r]][GenerateNormal[d-1]],
                              2,

Genetic`Parameters`GPossibilities[[r]][GenerateNormal[d-1],
                                                    GenerateNormal[d-1]],
                              3,

Genetic`Parameters`GPossibilities[[r]][GenerateNormal[d-1],
                                                    GenerateNormal[d-1],
                                                    GenerateNormal[d-1]],
                              4,

Genetic`Parameters`GPossibilities[[r]][GenerateNormal[d-1],
                                                    GenerateNormal[d-1],
                                                    GenerateNormal[d-1],
                                                    GenerateNormal[d-1]],
                              5,

Genetic`Parameters`GPossibilities[[r]][GenerateNormal[d-1],
                                                    GenerateNormal[d-1],
                                                    GenerateNormal[d-1],
                                                    GenerateNormal[d-1],
                                                    GenerateNormal[d-1]]
                             ]
                      ]

(* Generate an expression of given depth and maxcomplexity *)
Generate[d_]:=Module[
                      {y},
                      y=GenerateNormal[d];
                      While[
                            ((Depth[y]<d) || (LeafCount[y]>MaxComplexity)),
                            y=GenerateNormal[d]
                           ];
                      ReTouch[y]
                      ]

(* Update best-of-run individual and fitnesses in population *)
CheckSolution[gen_, x_, popname_]:=Module[
                          {minf, maxf, AverageFitness},
                          Genetic`Parameters`Fitnesses=AdjustedFitness /@ x;
                          minf=Position[Genetic`Parameters`Fitnesses,
Min[Genetic`Parameters`Fitnesses]][[1,1]];
                          maxf=Position[Genetic`Parameters`Fitnesses,
Max[Genetic`Parameters`Fitnesses]][[1,1]];
                          Genetic`Parameters`Solution=x[[maxf]];

Genetic`Parameters`SolutionFitness=Genetic`Parameters`Fitnesses[[maxf]];

                          AverageFitness=Apply[Plus,
Genetic`Parameters`Fitnesses]/PopulationSize;

Genetic`Parameters`SolutionSet=Append[Genetic`Parameters`SolutionSet,
                           {gen, Genetic`Parameters`Fitnesses[[maxf]],
x[[maxf]],
                                 Genetic`Parameters`Fitnesses[[minf]],
x[[minf]],
                                 AverageFitness}];
                          Print[popname, "-G", gen, ":  min=",
                                Genetic`Parameters`Fitnesses[[minf]], "
ave=",
                                AverageFitness, "  max=",
Genetic`Parameters`Fitnesses[[maxf]]];
                         ]

(* Check global populations *)
CheckGlobal[popname_]:=Module[
                              {info},
```

```
                              Print["** checking ", popname];

                              (* process population *)
                              BeginPackage["Genetic`Parameters`",
"Global`"];
                              Get[StringJoin[popname, ".log"]];
                              EndPackage[];

                              info=Last [Genetic`Parameters`SolutionSet];

                              If[

Genetic`Parameters`GMaxSolutionFitness<info[[2]],
                                 Genetic`Parameters`GMaxSolution=info[[3]];
                                 Print["** found better solution : ",
info[[2]]];

Genetic`Parameters`GMaxSolutionFitness=info[[2]];
                                 Genetic`Parameters`GMaxSolutionPop=popname;
                                ];

                              If[

Genetic`Parameters`GMinSolutionFitness>info[[4]],
                                 Genetic`Parameters`GMinSolution=info[[5]];
                                 Print["** found worse solution : ",
info[[4]]];

Genetic`Parameters`GMinSolutionFitness=info[[4]];
                                 Genetic`Parameters`GMinSolutionPop=popname;
                                ];


Genetic`Parameters`GAveSolutionFitness+=info[[6]];

Genetic`Parameters`TotTime+=Genetic`Parameters`TimeTaken;
                              Genetic`Parameters`NoOfIndividuals+=
                               Length[Genetic`Parameters`Population];
                             ]

(* Check for global solutions among all populations *)
CheckGlobalSolutions:=Module[
                             {f, mp},

                             Print["** Checking global status"];

                             BeginPackage["Genetic`Parameters`", "Global`"];
                             Get["pop.log"];
                             EndPackage[];

                             Genetic`Parameters`GMaxSolution=1;
                             Genetic`Parameters`GMaxSolutionFitness=0;
                             Genetic`Parameters`GMaxSolutionPop="pop";

                             Genetic`Parameters`GMinSolution=1;
                             Genetic`Parameters`GMinSolutionFitness=1;
                             Genetic`Parameters`GMinSolutionPop="pop";

                             Genetic`Parameters`GAveSolutionFitness=0;
                             Genetic`Parameters`NoOfIndividuals=0;

                             Map[CheckGlobal,
Genetic`Parameters`PopulationNames];

                             Genetic`Parameters`GAveSolutionFitness=
                              N[Genetic`Parameters`GAveSolutionFitness/
                                Genetic`Parameters`NoOfIndividuals];

                             If[
                                Genetic`Parameters`GMaxSolutionFitness>
                                Genetic`Parameters`GlobalSolutionFitness,
```

```
                              Genetic`Parameters`GlobalSolutionFitness=
                               Genetic`Parameters`GMaxSolutionFitness;
                              Genetic`Parameters`GlobalSolution=
                               Genetic`Parameters`GMaxSolution;
                             ];

                           Genetic`Parameters`GlobalSolutionSet=
                            Append[Genetic`Parameters`GlobalSolutionSet,
                            {Genetic`Parameters`Generation,
                             Genetic`Parameters`GMaxSolutionFitness,
                             Genetic`Parameters`GMaxSolution,
                             Genetic`Parameters`GMinSolutionFitness,
                             Genetic`Parameters`GMinSolution,
                             Genetic`Parameters`GAveSolutionFitness}];

                           DeleteFile["pop.log"];
                           Save["pop.log", Genetic`Parameters`TotTime];
                           Save["pop.log",
Genetic`Parameters`GlobalSolutionSet];
                           Save["pop.log",
Genetic`Parameters`GlobalSolution];
                           Save["pop.log",
Genetic`Parameters`GlobalSolutionFitness];

                           Off[DeleteFile::nffil];
                           DeleteFile["pop.inf"];
                           On[DeleteFile::nffil];

                           mp=Select[
                                     Genetic`Parameters`MigrationPairs,
                                     (Random[Integer,
MigrationProbability]==0)&
                                    ];

                           f=OpenWrite["pop.inf"];
                           Write[f, Genetic`Parameters`Generation];
                           Write[f,
CForm[Genetic`Parameters`GlobalSolutionFitness]];
                           (* Write[f,
CForm[Genetic`Parameters`MinFitness]]; *)
                           Write[f,
CForm[Genetic`Parameters`NoOfSubpopulations]];
                           Write[f, Length[mp]];
                           Map[
                               (Write[f, TextForm[StringJoin["M",
ToString[#]]]])&,
                               mp
                              ];
                           Close[f];

                           Print["** Finished global checks"];
                          ]

(* Initialise a population *)
InitializePop[popname_]:=Block[
                  {poplog, i, j},
                  If[
                     LengthOfMember==1,
                     Genetic`Parameters`Population=Table[
                       Generate[
                                Mod[
                                    i,
                                    Genetic`Parameters`MaxInitialSize
                                   ]+1
                               ],
                               {i, 1, Genetic`Parameters`PopulationSize}
                       ],
                     Genetic`Parameters`Population=Table[
                       Table[
                         Generate[
                                  Mod[
                                      i,
```

```
                                     Genetic`Parameters`MaxInitialSize
                                    ]+1
                                ],
                                {j, 1, Genetic`Parameters`LengthOfMember}
                        ],
                        {i, 1, Genetic`Parameters`PopulationSize}
                      ]
                   ];

                 Genetic`Parameters`Population[[1]]=
                    {PPlus[PTimes[Evaluate[Genetic`Parameters`gq0],
Genetic`Parameters`dt1],
                     PTimes[Evaluate[Genetic`Parameters`gq1],
Genetic`Parameters`dt2]],
                     PPlus[PTimes[Evaluate[Genetic`Parameters`gq0x],
Genetic`Parameters`dx1],
                     PTimes[Evaluate[Genetic`Parameters`gq1x],
Genetic`Parameters`dx2]]};
                 Genetic`Parameters`SolutionFitness=0;
                 Genetic`Parameters`SolutionSet={};
                 Genetic`Parameters`Generation=0;
                 Genetic`Parameters`TimeTaken=0;
                 Print[popname, "-G", Genetic`Parameters`Generation, ":
calculating fitnesses ..."];
                 Print[popname, "-G", Genetic`Parameters`Generation, ":
done ... ",
                       Timing[CheckSolution[Genetic`Parameters`Generation,
Genetic`Parameters`Population, popname]]
                        [[1]]
                      ];
                 Print[popname, "-G", Genetic`Parameters`Generation, ":
best-of-run fitness so far = ",
                       Genetic`Parameters`SolutionFitness];

                 Off[DeleteFile::nffil];
                 DeleteFile[StringJoin[popname, ".plg"]];
                 DeleteFile[StringJoin[popname, ".log"]];
                 DeleteFile[StringJoin[popname, ".new"]];
                 DeleteFile[StringJoin[popname, ".old"]];
                 On[DeleteFile::nffil];

                 poplog=OpenAppend[StringJoin[popname, ".plg"]];
                 WriteString[poplog, "pop={"];
                 Write[poplog, {Genetic`Parameters`Generation,
Genetic`Parameters`Fitnesses}];
                 Close[poplog];

                 Save[StringJoin[popname, ".log"],
Genetic`Parameters`Population];
                 Save[StringJoin[popname, ".log"],
Genetic`Parameters`Fitnesses];
                 Save[StringJoin[popname, ".log"],
Genetic`Parameters`Generation];
                 Save[StringJoin[popname, ".log"],
Genetic`Parameters`TimeTaken];
                 Save[StringJoin[popname, ".log"],
Genetic`Parameters`Solution];
                 Save[StringJoin[popname, ".log"],
Genetic`Parameters`SolutionFitness];
                 Save[StringJoin[popname, ".log"],
Genetic`Parameters`SolutionSet];

                 Information[Genetic`Parameters`Population];
                 GPopInformation [popname];
                ]

(* Initialise table of name prefixes, migration pairs *)
InitNames := Module[
                   {dim, row1, row2, row3, col1, col2, col3, names={},
                    popnos, pop, migt, miglen},

                   Genetic`Parameters`PopulationNames=Table[
```

```
                                          StringJoin["pop", ToString[i]],
                                          {i, 1,
Genetic`Parameters`NoOfSubpopulations}
                                         ];

                    If[
                       Genetic`Parameters`NoOfSubpopulations==1,
                       Genetic`Parameters`MigrationPairs={};
                       Return[]
                      ];

                    Genetic`Parameters`MigrationPairs=Table[
                          dim=Sqrt[Genetic`Parameters`NoOfSubpopulations];
                          row2=Floor[(pop-1)/dim];
                          col2=Mod[(pop-1), dim];
                          row1=Mod[row2-1, dim]; row3=Mod[row2+1, dim];
                          col1=Mod[col2-1, dim]; col3=Mod[col2+1, dim];
                          popnos={row1*dim+col1+1, row1*dim+col2+1,
row1*dim+col3+1,
                                  row2*dim+col1+1,
row2*dim+col3+1,
                                  row3*dim+col1+1, row3*dim+col2+1,
row3*dim+col3+1};
                          Map[({pop, #})&, popnos],
                          {pop, 1, Genetic`Parameters`NoOfSubpopulations}
                         ];
                    (*Print[Genetic`Parameters`MigrationPairs];*)
                    Genetic`Parameters`MigrationPairs=
                     Flatten[Genetic`Parameters`MigrationPairs, 1];
                    (*Print[Genetic`Parameters`MigrationPairs];*)
                    Genetic`Parameters`MigrationPairs=
                     Map[Sort, Genetic`Parameters`MigrationPairs];
                    (*Print[Genetic`Parameters`MigrationPairs];*)
                    Genetic`Parameters`MigrationPairs=
                     Union[Genetic`Parameters`MigrationPairs];
                    (*Print[Genetic`Parameters`MigrationPairs];*)
                    migt=Genetic`Parameters`MigrationPairs;
                    Genetic`Parameters`MigrationPairs=
                     Map[((#[[1]]-1)*
                         Genetic`Parameters`NoOfSubpopulations+
                         #[[2]]-1)&,
                         migt
                        ];
                   ]

(* Initialise all parameters and populations *)
Initialize::nofunc="a list of Functions must be defined first"
Initialize::noterm="a list of Terminals must be defined first"
Initialize::noperm="a list of the no of Parameters in each function must
                    be defined"
Initialize:=Module[
                   {Proc, DelList},
                   If[NameQ["Functions"],,
                           Message[Initialize::nofunc];
                           Return[]];
                   If[NameQ["Terminals"],,
                           Message[Initialize::noterm];
                           Return[]];
                   If[NameQ["Parameters"],,
                           Message[Initialize::noparm];
                           Return[]];

                   Off[DeleteFile::nffil];
                   DeleteFile["calced.m"];
                   DeleteFile["pop.inf"];
                   DelList=FileNames["logfile.*"];
                   If[DelList!={}, DeleteFile[DelList]];
                   DelList=FileNames["*.plg"];
                   If[DelList!={}, DeleteFile[DelList]];
                   DelList=FileNames["*.log"];
                   If[DelList!={}, DeleteFile[DelList]];
                   DelList=FileNames["backup.*"];
```

```
                   If[DelList!={}, DeleteFile[DelList]];
                   On[DeleteFile::nffil];

                   Map[
                       (DeleteDirectory[#, DeleteContents->True])&,
                       FileNames["PROC*"]
                      ];

                   Genetic`Parameters`GlobalSolution=1;
                   Genetic`Parameters`GlobalSolutionFitness=0;
                   Genetic`Parameters`GlobalSolutionSet={};
                   Genetic`Parameters`TotTime=0;
                   Save["pop.log", Genetic`Parameters`GlobalSolution];
                   Save["pop.log",
Genetic`Parameters`GlobalSolutionFitness];
                   Save["pop.log", Genetic`Parameters`GlobalSolutionSet];
                   Save["pop.log", Genetic`Parameters`TotTime];

                   MakePossibilities;
                   Save["calced.m", Genetic`Parameters`GPossibilities];
                   Save["calced.m", Genetic`Parameters`GPossParameter];
                   Save["calced.m", Genetic`Parameters`GTermLength];
                   Save["calced.m", Genetic`Parameters`GPossLength];

                   InitNames;

                   Save["calced.m", Genetic`Parameters`PopulationNames];
                   Save["calced.m", Genetic`Parameters`MigrationPairs];

                   Genetic`Parameters`PopulationSize=
                    Genetic`Parameters`PopulationSize/
                    Genetic`Parameters`NoOfSubpopulations;

                   GInformation;

                   Map[InitializePop, Genetic`Parameters`PopulationNames];

                   CheckGlobalSolutions;
                  ]

GInformation:=Module[{},
                     $Output=Append[$Output, OpenWrite["params.txt"]];
                     SetOptions[$Output[[2]], FormatType->TextForm];
                     Print[""];
                     Print["Population Size       : ",
Genetic`Parameters`PopulationSize*

Genetic`Parameters`NoOfSubpopulations];
                     Print["No of Subpopulations  : ",
Genetic`Parameters`NoOfSubpopulations];
                     Time[Genetic`Parameters`TotTime, "Total time taken
: "];
                     Print["Max no of Generations : ",
Genetic`Parameters`MaxGenerations];
                     Print["Max initial size      : ",
Genetic`Parameters`MaxInitialSize];
                     Print["Max size              : ",
Genetic`Parameters`MaxSize];
                     Print["Maximum complexity    : ",
Genetic`Parameters`MaxComplexity];
                     Print["Min solution fitness  : ",
Genetic`Parameters`MinFitness];
                     Print["Mutation probability  : ",
Genetic`Parameters`MutationProbability];
                     Print["Crossover probability : ",
Genetic`Parameters`CrossoverProbability];
                     Print["Terminal set          : ",
Genetic`Parameters`Terminals];
                     Print["Function set          : ",
Genetic`Parameters`Functions];
                     Print[""];
                     Close[$Output[[2]]];
```

```
                     $Output=Take[$Output, 1];
                    ]

GPopInformation[popname_]:=Module[{},
                     Print[""];
                     Print["Population name       : ", popname];
                     Print["Current generation    : ",
Genetic`Parameters`Generation];
                     Print["Current best fitness  : ",
Genetic`Parameters`SolutionFitness];
                     Print[""];
                     Print["Current best individual ***"];
                     Print[Genetic`Parameters`Solution];
                     Print[""];
                    ]

End[]

EndPackage[]
```

genmain.m

```
(* Genetic Programming *)

(* Main routines *)

(* H. Suleman *)
(* 28 May 1996 *)

(* Get normal distribution functionality *)
Needs["Statistics`NormalDistribution`"];

(* Get time routines *)
Needs["Genetic`Time`", "time.m"]

(* Get extra definitions for basic arithmetic operations *)
Needs["Genetic`ExtraDefinitions`", "xtradefs.m"]

(* Get parameters *)
Needs["Genetic`Parameters`", "default.m"]

(* Get initialization routines *)
Needs["Genetic`Initialization`", "initial.m"]

(* Get file locking routines *)
Needs["Genetic`Shares`", "shares.m"]

(* Get genetic operators *)
Needs["Genetic`Operators`", "operator.m"]

BeginPackage["Genetic`Main`", {"Genetic`Parameters`",
                               "Genetic`Initialization`",
                               "Genetic`Operators`",
                               "Statistics`NormalDistribution`"}]

CreateNewGeneration::usage = "CreateNewGeneration[oldgen] creates a new
generation from the old generation using fitness-proportionate
reproduction."

StartRun::usage = "Starts the run of the genetic algorithm."

RegisterProc::usage = "Registers a processor."

Begin["`Private`"]

(* Make cumulative fitnesses vector *)
CalcFitnessSum:=Module[{fitsum, i},
   fitsum=Table[Apply[Plus, Take[Fitnesses, i]],
                {i, 1, Length[Fitnesses]}];
```

```
                        fitsum=Insert[fitsum, 0, 1];
                        fitsum
                       ]

(* Bisection algorithm search for roulette wheel fitness choice *)
Search[x_, fitsum_] :=
  Module[{Mid, Start=1, Stop=Length[fitsum]},
         While[Start+1 != Stop,
               Mid = Floor[(Start+Stop)/2];
               If[fitsum[[Mid]] > x,
                  Stop=Mid,
                  Start=Mid
                 ]
              ];
         Start
        ]

(* Create new generation from previous one *)
CreateNewGeneration[x_] := Module[
  {maxwheel, newgen, lenx, fitsum, i},
  newgen={};
  maxwheel=Apply[Plus, Fitnesses];
  lenx=Length[x];
  fitsum=CalcFitnessSum;
  Do[
    Module[
      {spot, index},
      spot=Random[]*maxwheel;
      index=Search[spot, fitsum];
      newgen=Append[newgen, x[[index]]]
    ],
    {i, 1, lenx}
  ];
  newgen
]

(* Get a sub-population filename *)
GetPopFile:=Module[
                   {OrigDirectory, t},
                   OrigDirectory=Directory[];
                   SetDirectory[Genetic`Parameters`Processor];
                   t=FileNames[];
                   SetDirectory[OrigDirectory];
                   If[
                      Length[t]==0,
                      "NOFILES",
                      If[
                         SameQ[t[[1]], "DONE"],
                         "NOFILES",
                         t[[1]]
                        ]
                     ]
                  ]

(* perform migration between source and dest populations *)
MigratePop[source_, dest_]:=Module[
                         {maxwheel1, fitsum1, Fitnesses1, Population1,
                          TimeTaken1, SolutionSet1, Solution1,
                          SolutionFitness1, maxwheel, fitsum, noofx,
                          fname, i},

                         Print["Migrating pops  :   ", source, " & ", dest];

(*                          If[Random[Integer, MigrationProbability]!=0,
                             Return[]
                            ];
*)
                         BeginPackage["Genetic`Parameters`", "Global`"];
                         Get[StringJoin[source, ".log"]];
                         EndPackage[];

                         Population1=Population;
```

```
Fitnesses1=Fitnesses;
TimeTaken1=TimeTaken;
Solution1=Solution;
SolutionFitness1=SolutionFitness;
SolutionSet1=SolutionSet;

maxwheel1=Apply[Plus, Fitnesses];
fitsum1=CalcFitnessSum;

BeginPackage["Genetic`Parameters`", "Global`"];
Get[StringJoin[dest, ".log"]];
EndPackage[];

maxwheel=Apply[Plus, Fitnesses];
fitsum=CalcFitnessSum;

noofx=Random[
             NormalDistribution[
MigrationPercentage,
                                  MigrationDeviation
                                 ]
            ];

(* noofx=Random[Real, MigrationDeviation*
                    MigrationPercentage*2];
noofx-=MigrationDeviation*MigrationPercentage;
noofx+=MigrationPercentage; *)

If[noofx<0, noofx=0];
If[noofx>1, noofx=1];
noofx*=Length[Population1];
noofx=Floor[noofx];

Do[
   Module[
          {spot1, index1, spot, index, temp},
          spot1=Random[]*maxwheel1;
          index1=Search[spot1, fitsum1];
          spot=Random[]*maxwheel;
          index=Search[spot, fitsum];

          temp=Population[[index]];
Population[[index]]=Population1[[index1]];
          Population1[[index1]]=temp;

          temp=Fitnesses[[index]];
          Fitnesses[[index]]=Fitnesses1[[index1]];
          Fitnesses1[[index1]]=temp;
         ],
    {i, 1, noofx}
   ];

fname=StringJoin[dest, ".new"];
Save[fname, Population];
Save[fname, Fitnesses];
Save[fname, Generation];
Save[fname, TimeTaken];
Save[fname, Solution];
Save[fname, SolutionFitness];
Save[fname, SolutionSet];
RenameFile[StringJoin[dest, ".log"],
           StringJoin[dest, ".old"]];
RenameFile[fname, StringJoin[dest, ".log"]];
DeleteFile[StringJoin[dest, ".old"]];

Population=Population1;
Fitnesses=Fitnesses1;
TimeTaken=TimeTaken1;
Solution=Solution1;
SolutionFitness=SolutionFitness1;
```

```
                         SolutionSet=SolutionSet1;

                         fname=StringJoin[source, ".new"];
                         Save[fname, Population];
                         Save[fname, Fitnesses];
                         Save[fname, Generation];
                         Save[fname, TimeTaken];
                         Save[fname, Solution];
                         Save[fname, SolutionFitness];
                         Save[fname, SolutionSet];
                         RenameFile[StringJoin[source, ".log"],
                                    StringJoin[source, ".old"]];
                         RenameFile[fname, StringJoin[source, ".log"]];
                         DeleteFile[StringJoin[source, ".old"]];
                        ]

(* MigratePop[pairs_]:=Module[
                           {},
                           Print["Migrating populations ", pairs];
                           Map[
                               (MigrateMembers[ #1[[1]], #1[[2]] ])&,
                               pairs
                              ];
                          ]
*)


(* perform migration based on parameters *)
Migrate[popf_]:=Module[
                       {OrigDirectory, FullNum, firstpop, secondpop},

                       If[
                          SameQ[StringDrop[popf, 1], "START"],
                          CheckGlobalSolutions;
                          If[

Genetic`Parameters`GlobalSolutionFitness>=MinFitness,
                             OrigDirectory=Directory[];
                             SetDirectory[Genetic`Parameters`Processor];
                             Save["DONE", MinFitness];
                             SetDirectory[OrigDirectory]
                            ],
                          FullNum=ToExpression[StringDrop[popf, 1]];
                          firstpop=Floor[FullNum/NoOfSubpopulations]+1;
                          secondpop=Mod[FullNum, NoOfSubpopulations]+1;
                          MigratePop[StringJoin["POP", ToString[firstpop]],
                                     StringJoin["POP", ToString[secondpop]]]
                         ];

                       OrigDirectory=Directory[];
                       SetDirectory[Genetic`Parameters`Processor];
                       DeleteFile[popf];
                       SetDirectory[OrigDirectory];
                      ]

(* Apply Genetic algorithm *)
ApplyGen := Module[
   {popfile, onetime, poplog, mig, OrigDirectory},

   BeginPackage["Genetic`Parameters`", "Global`"];
   Get["calced.m"];
   EndPackage[];

        Print["Waiting for processor start flag ..."];
        popfile=GetPopFile;

        While[
              SameQ[popfile, "NOFILES"],
              Pause[1];
              popfile=GetPopFile
             ];
```

```
        If[
           SameQ[StringTake[popfile, 1], "M"],
           Migrate[popfile];
           Return[]
          ];

        (* process population *)
        BeginPackage["Genetic`Parameters`", "Global`"];
        Get[StringJoin[popfile, ".log"]];
        EndPackage[];

        onetime=Timing[
         Print[popfile, "-G", Generation, ": mating pool ... ",
Timing[newpop=CreateNewGeneration[Population]][[1]]];
         Print[popfile, "-G", Generation, ": crossover   ... ",
Timing[newpop=Crossover[newpop]][[1]]];
         Print[popfile, "-G", Generation, ": mutation    ... ",
Timing[newpop=Map[Mutate, newpop]][[1]]];
         Generation++;
         Population=newpop;
         Print[popfile, "-G", Generation, ": fitnesses   ... "];
         Print[popfile, "-G", Generation, ": done        ... ",
Timing[CheckSolution[Generation, newpop, popfile]][[1]]];
         Print[popfile, "-G", Generation, ": best-of-run   = ",
SolutionFitness];
        ][[1]];
        Time[onetime, popfile, "-G", Generation, ": time for gen  = "];
        TimeTaken+=onetime;

        Save[StringJoin[popfile, ".new"], Population];
        Save[StringJoin[popfile, ".new"], Fitnesses];
        Save[StringJoin[popfile, ".new"], Generation];
        Save[StringJoin[popfile, ".new"], TimeTaken];
        Save[StringJoin[popfile, ".new"], Solution];
        Save[StringJoin[popfile, ".new"], SolutionFitness];
        Save[StringJoin[popfile, ".new"], SolutionSet];
        RenameFile[StringJoin[popfile, ".log"], StringJoin[popfile,
".old"]];
        RenameFile[StringJoin[popfile, ".new"], StringJoin[popfile,
".log"]];
        DeleteFile[StringJoin[popfile, ".old"]];

        poplog=OpenAppend[StringJoin[popfile, ".plg"]];
        WriteString[poplog, ","];
        Write[poplog, {Generation, Fitnesses}];
        Close[poplog];
        Print[popfile, "-G", Generation, ": system saved ..."];

        OrigDirectory=Directory[];
        SetDirectory[Genetic`Parameters`Processor];
        DeleteFile[popfile];
        SetDirectory[OrigDirectory];
]

(* Start run of algorithm *)
StartRun[x_]:=Module[
              {result, log, i},

              Do[
                 log=StringJoin["LOGFILE.", ToString[x]];
                 $Output=Append[$Output, OpenAppend[log]];
                 SetOptions[$Output[[2]], FormatType->TextForm];

                 Genetic`Parameters`Processor=StringJoin["PROC",
ToString[x]];

                 CheckAbort[
                            ApplyGen,
                            0
                           ];

                 Close[$Output[[2]]];
```

```
                 $Output=Take[$Output, 1],
                 {i, 1, Genetic`Parameters`Epoch}
                ];

            ]

RegisterProc[x_]:=Module[
                         {proc},
                         proc=StringJoin["PROC", ToString[x]];
                         CreateDirectory[proc];
                        ]

End[]

EndPackage[]
```

stats.m

```
(* Genetic Programming *)

(* Statistics routines *)

(* H. Suleman *)
(* 30 October 1996 *)

Needs["Graphics`Graphics`"];

Needs["Graphics`Animation`"];

BeginPackage["Genetic`Stats`", {"Graphics`Graphics`",
                                "Graphics`Animation`",
                                "Graphics`Graphics3D`" }]

GlobalCurve::usage = "GlobalCurve[] shows the global fitness curve."

GlobalHistogram::usage = "GlobalHistogram produces a set of
               histograms for the entire population."

MaxHistogram::usage = "MaxHistogram produces a set of 3-D histograms
                       showing the progress of the solution fitness
                       in each subpopulation."

AveHistogram::usage = "AveHistogram produces a set of 3-D histograms
                       showing the average fitness in each subpopulation."

CalcHistogram::usage = "CalcHistogram calculates the global histograms
                        and 3D histograms."

HistogramData={};

Histogram3DMax={};
Histogram3DAve={};

Begin["`Private`"]

GlobalCurve:=Module[
                    {t, MaxG, MinG, AveG},

                    BeginPackage["Genetic`Parameters`"];
                    Get["pop.log"];
                    EndPackage[];

                    t=MapThread[List, Genetic`Parameters`GlobalSolutionSet];

                    MaxG=ListPlot[MapThread[List, {t[[1]], t[[2]]}],
                                  PlotRange->{{0, Max[t[[1]]]}, {0, 1}},
                                  PlotStyle->{RGBColor[1,0,0]},
                                  Frame->True,
```

```
                                  FrameLabel->{"Generation       Fit(ness):
red=max green=min blue=ave",
                                               "Fit"},
                                  PlotLabel->"Global Fitness Curve",
                                  PlotJoined->True,
                                  DisplayFunction->Identity];

                    MinG=ListPlot[MapThread[List, {t[[1]], t[[4]]}],
                                  PlotRange->{{0, Max[t[[1]]]}, {0, 1}},
                                  PlotStyle->{RGBColor[0,1,0]},
                                  Frame->True,
                                  FrameLabel->{"Generation       Fit(ness):
red=max green=min blue=ave",
                                               "Fit"},
                                  PlotLabel->"Global Fitness Curve",
                                  PlotJoined->True,
                                  DisplayFunction->Identity];

                    AveG=ListPlot[MapThread[List, {t[[1]], t[[6]]}],
                                  PlotRange->{{0, Max[t[[1]]]}, {0, 1}},
                                  PlotStyle->{RGBColor[0,0,1]},
                                  Frame->True,
                                  FrameLabel->{"Generation       Fit(ness):
red=max green=min blue=ave",
                                               "Fit"},
                                  PlotLabel->"Global Fitness Curve",
                                  PlotJoined->True,
                                  DisplayFunction->Identity];

                    Show [{MaxG, MinG, AveG},
                          DisplayFunction->$DisplayFunction];
                 ]

GetPopNumber[x_]:=ToExpression[StringTake[x, {4, StringLength[x]-4}]]

CalcHistogram:=Module[
                        {t, data, popfit, figs, gen, popsize=0, numgen,
                         popfiles, first=1, maxes, popnumber, inFile,
                         outFile},

                        popfiles=FileNames["pop*.plg"];
                        popfiles=Sort[
                                      popfiles,
                                      (Less[GetPopNumber[#1],
GetPopNumber[#2]])&
                                     ];
                        Histogram3DMax=Table[0, {Length[popfiles]}];
                        Histogram3DAve=Table[0, {Length[popfiles]}];

                        Map[
                            (Print["copying file ", #];
(*                            cmdline="copy ";
                             cmdline=StringJoin[cmdline, #];
                             cmdline=StringJoin[cmdline, "+pop.m pop.ful /Y
> nul"];
                             Run[cmdline];*)

                             inFile=OpenRead["pop1.plg"];
                             outFile=OpenWrite["pop.ful"];
                             While[
                                   i=Read[inFile, String];
                                    Not[SameQ[i, EndOfFile]],
                                   WriteString[outFile, i, "\n"]
                                  ];
                             Close[inFile];
                             WriteString[outFile, "}"];
                             Close[outFile];

                             Print["reading in data"];
                             BeginPackage["Genetic`Parameters`"];
                             Get["pop.ful"];
                             EndPackage[];
```

```
                             Print["separating data"];
                             popfit=MapThread[List,
Genetic`Parameters`pop][[2]];
                             numgen=Max[MapThread[List,
Genetic`Parameters`pop][[1]]];

                             If[
                                first==1,
                                data=Table[Table[0, {10}], {numgen}];
                                first=0
                               ];

                             Print["discretizing data"];
                             Do[
                                figs=Map[Floor, popfit[[gen]]*10];
                                figs=Map[If[#==0, 1, #]&, figs];
                                Map[(data[[gen, #]]++)&, figs],
                                {gen, 1, numgen}
                               ];

                             Print["extracting maximums"];
                             maxes={};
                             Do[
                                maxes=Append[maxes, Max[popfit[[gen]]]],
                                {gen, 1, numgen}
                               ];
                             popnumber=ToExpression[
                                       StringDrop[StringDrop[#, 3], -4]
                                                    ];
                             Histogram3DMax[[popnumber]]=maxes;

                             Print["extracting averages"];
                             maxes={};
                             Do[
                                maxes=Append[maxes,
                                       Apply[Plus,
                                        popfit[[gen]]]/Length[popfit[[gen]]]
                                            ],
                                {gen, 1, numgen}
                               ];
                             Histogram3DAve[[popnumber]]=maxes;

                             popsize+=Length[popfit[[1]]])&,

                             popfiles
                            ];

                            Print["generating global graphs"];
                            HistogramData=
                              Table[
                                    BarChart[data[[gen]],
                                      BarLabels->Table[i, {i, 0, 0.9, 0.1}],
                                      PlotRange->{{0, 11}, {0, popsize}},
                                      PlotLabel->StringJoin["Global
Generation ",
                                                  ToString[gen]],
                                      DisplayFunction->Identity],
                                    {gen, 1, numgen}
                                   ];

                            Print["generating maximum graphs"];
                            Histogram3DMax=MapThread[List, Histogram3DMax];
                            Histogram3DMax=Map[Partition[#,
                                                Sqrt[Length[popfiles]]]&,
                                               Histogram3DMax];
                            Histogram3DMax=
                              Table[
                                    BarChart3D[Histogram3DMax[[gen]],
                                      PlotRange->{Automatic, Automatic,
{0,1}},
```

```
                                      PlotLabel->StringJoin["Max of
Generation ",
                                                  ToString[gen]],
                                      ViewPoint->{4,1,4},
                                      DisplayFunction->Identity],
                                     {gen, 1, numgen}
                                    ];

                            Print["generating average graphs"];
                            Histogram3DAve=MapThread[List, Histogram3DAve];
                            Histogram3DAve=Map[Partition[#,
                                                Sqrt[Length[popfiles]]]&,
                                               Histogram3DAve];
                            Histogram3DAve=
                              Table[
                                    BarChart3D[Histogram3DAve[[gen]],
                                      PlotRange->{Automatic, Automatic,
{0,1}},
                                      PlotLabel->StringJoin["Ave of
Generation ",
                                                  ToString[gen]],
                                      ViewPoint->{4,1,4},
                                      DisplayFunction->Identity],
                                     {gen, 1, numgen}
                                    ];
                       ]

MyOpenTempCounter=1;
MyOpenTemporary:=Module[
                        {front="TF"},
                        front=StringJoin[front,
ToString[MyOpenTempCounter++]];
                        OpenWrite[front]
                       ]

MyRasterFunction = Module[
                          {fname = MyOpenTemporary},
                          Display[fname, #];
                          Close[fname]
                         ]&

GlobalHistogram:=Module[
                        {},
                        If[HistogramData=={}, CalcHistogram];
                        ShowAnimation[HistogramData,
                                      RasterFunction->MyRasterFunction]
                       ]

MaxHistogram:=Module[
                     {},
                     If[Histogram3DMax=={}, CalcHistogram];
                     ShowAnimation[Histogram3DMax,
                                   RasterFunction->MyRasterFunction]
                    ]

AveHistogram:=Module[
                     {},
                     If[Histogram3DAve=={}, CalcHistogram];
                     ShowAnimation[Histogram3DAve,
                                   RasterFunction->MyRasterFunction]
                    ]

Stats[s_String]:=Module[{},
                        Display[StringJoin[s, ".scu"], GlobalCurve];
                       ]

End[]

EndPackage[]
```

BIBLIOGRAPHY

Abell, M. L. and Braselton, J. P., *The Mathematica Handbook*, AP Professional, 1992.

Andre, D., *Artificial Evolution of Intelligence: Lessons from natural evolution - An illustrative approach using Genetic Programming*, BS Honors thesis, Stanford University Symbolic Systems Program, 1994.

Andre, D., *Learning and Upgrading Rules for an OCR System using Genetic Programming*, in *Proceedings of the First IEEE Conference on Evolutionary Computation*, Volume 1, pp 462-467, IEEE Press, 1994.

Andre, D., Bennett, F. H. III and Koza, J. R., *Discovery by Genetic Programming of a Cellular Automata Rule that is Better than any Known Rule for the Majority Classification Problem*, in *Genetic Programming 1996: Proceedings of the First Annual Conference, Stanford University*, MIT Press, 1996.

Andrews, M. and Prager, R., *Genetic Programming for the Acquisition of Double Auction Market Strategies*, in *Advances in Genetic Programming*, pp 355-368, edited by Kinnear, K. E. Jr., MIT Press, 1994.

Cantu-Paz, E., *A Summary of Research on Parallel Genetic Algorithms*, IlliGAL (Illinois Genetic Algorithms Laboratory) Report No. 95007, 1995.

Darwin, C., *The Origin of Species* - a variorum text edited by Morse Peckham, University of Pennsylvania, Philadelphia, 1959.

Freeman, A., *Simulating Neural Networks with Mathematica*, p271-273, Addison-Wesley, 1994.

Goodman, E. D., *An Introduction to GALOPPS (Genetic Algorithm Optimized for Portability and Parallelism)*, GARAGe (Genetic Algorithms Research and Applications Group) Technical Report #96-07-01, Michigan State University, 1996.

Hajek, M., *Optimization of Fuzzy Rules by Using a Genetic Algorithm*, in *Proceedings of ICARCV, The Third International Conference on Automation, Robotics and Computer Vision, Singapore*, pp 2111-2115, 1994.

Hartl, D. L., Freifelder, D. and Snyder, L. A., *Basic Genetics*, Jones and Bartlett Publishers Inc., Boston, 1988.

Haynes, T., Wainwright, R., Sen, S. and Schoenefeld, D., *Strongly Typed Genetic Programming in Evolving Cooperation Strategies*, in *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pp 271-278, Kaufman, san Francisco, 1995.

Heitkotter, J., *The Hitch-Hiker's Guide to Evolutionary Computation*, 1995.

Holland, J. H., *Adaptation in Natural and Artificial Systems*, MIT Press, 1992.

Keith, M. J. and Martin, M. C., Genetic Programming in C++: Implementation Issues, in *Advances in Genetic Programming,* pp 285-310, edited by Kinnear, K. E. Jr., MIT Press, 1994.

Kinnear, K. E. Jr., *Advances in Genetic Programming*, pp 3-19, MIT Press, 1994.

Koza, J. R., *Genetic Programming: On the Programming of Computers by means of Natural Selection*, MIT Press, 1992.

Koza, J. R., *Introduction to Genetic Programming*, in *Advances in Genetic Programming,* pp 21-42, edited by Kinnear, K. E. Jr., MIT Press, 1994.

Koza, J. R., *Genetic Programming II: Automatic Discovery of Reusable Programs*, MIT Press, 1994.

Koza, J. R. and Andre, D., *Parallel Genetic Programming on a Network of Transputers*, Stanford University Computer Science Department Technical Report stan-cs-tr-95-1542, 1995.

Koza, J. R., Bennett, F. H. III, Andre, D. and Keane, M. A., *Toward Evolution of Electronic Animals Using Genetic Programming*, in *Artificial Life V: Proceedings of the Fifth International Workshop on the Synthesis and Simulation of Living Systems*, MIT Press, Cambridge, 1996.

Koza, J. R., Bennett, F. H. III, Andre, D. and Keane, M. A., *Evolution of a Low-Distortion, Low-Bias 60 Decibal Op Amp with Good Frequency Generalization using Genetic Programming*, in *Late Breaking Papers at the Genetic Programming 1996 Conference, Stanford University,* Stanford University Bookstore, pp 94-100, 1996.

Levine, D., *A Parallel Genetic Algorithm for the Set Partitioning Problem*, in *Proceedings of INFORMS (Institute for Operations Research and the Management Sciences), New Orleans*, 1995.

Maeder, R., *Programming in Mathematica*, Addison-Wesley, 1991.

Michalewicz, Z., *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag, New York, 1992.

Nachbar, R. B., *Genetic Programming*, pp 36-47, in *The Mathematica Journal* 5(3), Miller Freeman Publications, 1995.

Oakley, H., *Two Scientific Applications of Genetic Programming: Stack Filters and Non-Linear Equations Fitting to Chaotic Data*, in *Advances in Genetic Programming,* pp 369-389, edited by Kinnear, K. E. Jr., MIT Press, 1994.

Punch, B., Zongker, D. and Goodman, E., *The Royal Tree Problem, a Benchmark for Single and Multi-population Genetic Programming*, in *Advances in Genetic Programming II,* edited by Kinnear, K. E. Jr., MIT Press, 1996.

Reynolds, C. W., *Evolution of Obstacle Avoidance Behavior: Using Noise to Promote Robust Solutions*, in *Advances in Genetic Programming,* pp 221-241, edited by Kinnear, K. E. Jr., MIT Press, 1994.

Ryan, C., *Pygmies and Civil Servants*, in *Advances in Genetic Programming,* pp 243-263, edited by Kinnear, K. E. Jr., MIT Press, 1994.

Schach, R., *Software Engineering*, Aksen Associates Incorporated Publishers, 1995.

Spencer, G. *Automatic Generation of Programs for Walking and Crawling*, in *Advances in Genetic Programming,* pp 335-353, edited by Kinnear, K. E. Jr., MIT Press, 1994.

Toth, G. J. and Lorincz, A., *Genetic Algorithm with Migration on Topology Conserving Maps*, in *Proceedings of ICANN '93 Amsterdam, The Netherlands*, pp 605-608, Springer-Verlag, London, 1993.

Wickham-Jones, T., *Mathematica Graphics*, TELOS/Springer-Verlag, 1994.

Wolfram, S., Mathematica: *A System for Doing Mathematics by Computer*, Addison-Wesley, 1991.

Wolfram, S., *Mathematica Reference Guide*, Addison-Wesley, 1992.