UNIVERSITY OF YORK

Improving Crossover Techniques in a Genetic Program

4th Year Final Report for degree of MEng in Electronic Engineering

Andrew Turner - Y3757996

June 2012

"Ignorance more frequently begets confidence than does knowledge: it is those who know little, and not those who know much, who so positively assert that this or that problem will never be solved by science."

- Charles Darwin

Abstract

This project documents an investigation into the effectiveness of a new form of crossover to be applied to Cartesian Genetic Programming. The assessment of the crossover technique is achieved via four distinct test cases, with the effectiveness analysed in each case. Using the results of these experiments it is shown that this new form of crossover is not beneficial to the search process of Cartesian Genetic Programming.

1 Introduction

This project furthers work published by Janet Clegg [1] surrounding a new crossover technique to be used by Cartesian Genetic Programming. The paper is included in Appendix A for the reader's reference. Cartesian Genetic Programming is a subset of Evolutionary Computation; a group of techniques used to solve optimisation problems via methods inspired by Darwinian evolution. The paper published by Janet Clegg reports a high decrease in convergence time required to find solutions compared to Cartesian Genetic Programming implemented without the new crossover technique. This project first repeats the experiments described within Janet Clegg's paper and then continues the research described by the further work section. The overall aim is to reach a conclusion over the effectiveness of the new crossover technique when applied to Cartesian Genetic Programming.

As a point of interest, the need for further research into the effectiveness of crossover techniques, as applied to Cartesian Genetic Programming, is also discussed by one of the creators of Cartesian Genetic Programming, Julian Miller. Reference is made for the need of this research in his book "Cartesian Genetic Programming" [2], as described in the following extract:

"Crossover operators have received relatively little attention in CGP. Originally a one-point crossover operator was used in CGP (similar to the n-point crossover in genetic algorithms) but was found to be disruptive to the subgraphs within the chromosomes, and had a detrimental affect on the performance of CGP. Some work by Clegg et al. has investigated crossover in CGP (and GP in general). Their approach uses a floating-point crossover operator, similar to that found in evolutionary programming, and also adds an extra layer of encoding to the genotype, in which all genes are encoded as a floating-point number in the range [0,1]. A larger population and tournament selection were also used instead of the (1 + 4) evolutionary strategy normally used in CGP, to try and improve the population diversity. The results of this new approach appear promising when applied to two symbolic regression problems, but further work is required on a range of problems in order to assess its advantages."

The remainder of this chapter shall now describe the reports structure and introduce each chapter. The Evolutionary Computation chapter introduces the general field of Evolutionary Computation and the various techniques which are employed to guide the search process. The Background Literature chapter discusses the literature surrounding Cartesian Genetic Programming; the Genetic Program used throughout this project. The Cartesian Genetic Programming chapter describes, in detail, the structure and operation of a "traditional" Cartesian Genetic Program. The Crossover Techniques chapter discusses a range of crossover techniques including that used within this project. The Investigation chapter discusses the aims and objectives of the project and describes at a high level the experiments which were undertaken. The Possible Test Cases Chapter describes several possible scenarios which could be used to evaluate the new crossover technique, of which a sub set were chosen for the experiments. The Project Timeline chapter describes the original order in which different aspects of the project were to be carried out and how long was to be spent on each stage. The Implementing the New Crossover Technique chapter discusses how the new crossover technique is implemented within the Cartesian Genetic Program.

The design of the author's Cartesian Genetic Program then begins with a Specification chapter defining specific criteria of the program. The Cartesian Genetic Program Production chapter then discusses the implementation of the author’s Cartesian Genetic Program. Finally for the design stage, the Testing Chapter describes the testing strategies which were undertaken to ensure the correct operation of the author’s Cartesian Genetic Program.

Each of the test cases investigated, to assess the effectiveness of the new crossover technique, is described within its own chapter; beginning with Repeating Janet Clegg's Experiments. The following test cases are also described within their own chapters: Test Case 1: Symbolic Regression, Test Case 2: Synthesis of Boolean Logic, Test Case 3: Function Optimisation and finally Test Case 4: Wall Avoider.

The project is then concluded with an Additional Investigations chapter discussing some additional experiments undertaken using the author’s Cartesian Genetic Program, followed by a final Conclusion and a Review of the Project overall.

2 Evolutionary Computation

Evolutionary Computation, and all of its variants, are search methods inspired by Darwinian Evolution. The term search method is used here in relation to searching what is sometimes called the solution space or design space. These are theoretical landscapes which contain all possible solutions to a given problem. These landscapes are navigated with a number of variables; these variables are what the Evolutionary Computational strategies optimise to find the most suitable solution.

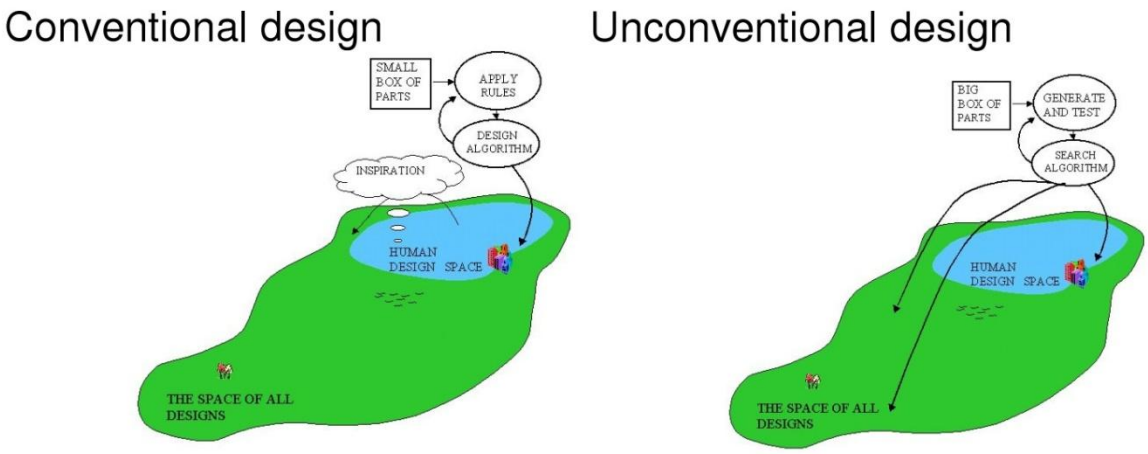

Figure 1 Depiction of Design Space[1]

Figure 1 gives a depiction of this design space, with the left image indicating what is possible with the use of design algorithms; also showing that it takes inspiration to expand the area of the design space currently understood. The right image shows how search algorithms are not constrained to what is referred to as the "human design space" and are capable of accessing a much wider range of solutions. It is interesting to note, that once a new solution is found outside of the "human design space", it is then possible to learn from this solution and widen our overall understanding of the design space, this is akin to inspiration.

Evolutionary Computation begins by creating an initial population of solutions, it is unlikely that any of these solutions will be at all suitable, but it is likely that some will be more suitable than others. These solutions are referred to as genotypes or chromosomes; as this

[1] Sourced from Julian F Millers taught lecture course "Bio-inspired computing" at the University of York 2011

technique has its roots in biology it shares much of the terminology. Once these chromosomes have been generated, the weaker are removed following the concept of natural selection; or more usually in Evolutionary Computation, "survival of the fittest". The remaining chromosomes are then used to produce the next generation, this process shares characteristics of "reproduction" with the use of "recombination" and/or "mutation". Recombination[2] is often referred to as crossover when describing Evolutionary Computation. This process of "survival of the fittest" followed by "reproduction" is iterated until a termination condition is met; this prevents the process continuing indefinitely. All of the processes described in this paragraph are now discussed in further detail.

2.1 Initial Population

The initial population is usually generated by randomly selecting values for the parameters which describe each chromosome. It is important that each chromosome is capable of having its fitness evaluated; in some cases it is necessary to employ a repair algorithm if a randomly generated chromosome might not always represent a valid solution. The initial population can also be seeded with a solution which is thought to be "near" a suitable solution. This technique can be used to improve upon existing solutions.

The size of the population is one of the many parameters which control the evolutionary process and the optimal size of the population depends upon the problem under investigation. It is often difficult to know a suitable population size before trial runs have been completed.

2.2 Survival of the Fittest

Survival of the fittest is the simplest form of natural selection; as it is achieved by looking at each chromosome individually. For this reason survival of the fittest is often the selection technique used to select candidates to generate the next population. The survival of the fittest is employed by assigning a numerical value to each chromosome indicating its fitness; this is achieved by what is commonly called a fitness function. The fitness function is often the most complex component of Evolutionary Computation requiring the most computation time, it is also the most bespoke component of Evolutionary Computation and will have to be re-written for each new problem investigated.

[2] Recombination is akin to sexual reproduction.

2.3 Reproduction

Reproduction is concerned with generating the next population from the current population; there are a wide range of methods for achieving this. In most cases reproduction can be split into three areas; Evolutionary Strategy, Selection, and finally Mutation and/or Crossover; each of which is now discussed.

2.3.1 Evolutionary Strategy

It should be noted, that here the term evolutionary strategy does not refer to the subfield of Evolutionary Computation introduced by Ingo Rechenberg. Here the term evolutionary strategy refers to different techniques which can be used to govern how to create the next generation from the current generation.

There are two main forms of evolutionary strategy, which take the form of $(\mu + \lambda)$-ES and (μ, λ)-ES. The "μ" represents the number of parents that are used to create the next generation and the "λ" represents the number of children which are created. The "+" form represents that the next generation comprises of parents and children, whereas the "," form represents that the next generation comprises of children alone. The "-ES" extension refers to evolutionary strategy.

The "μ" and "λ" values are parameters which control the evolutionary process; they also dictate the population size[3]. The distinguishing features between the "+" form and the "," form is that the "+" form keeps hold of the current best solution(s), where as the "," form does not. An advantage of the "," form is that sometimes it is necessary to allow deviations from what could be local solutions to find the overall global solution. The "," from is also more akin to biology; on which Evolutionary Computation is based.

There are many variations on this arrangement which can be used to change the way in which the evolution takes effect. A common technique which is referred to as "elitism" takes the form of (1 + λ)-ES. This creates a scenario where the next generation is entirely created from the "best" chromosome and that the "best" chromosome is always included in the next population. It is also common practice to always select a child over a parent, to be the next elite chromosome, if they both share equal fitness. This is because the child may

[3] Population size = μ + λ when following the "+" form and population size = λ when following the "," form.

contain different redundant genes[4] to the parent which may become beneficial in future generations; always keeping the parent can cause the search to stagnate.

2.3.2 Selection

Once all of the chromosomes have been assigned a fitness value, a selection of the population can be chosen to seed the next generation. The number of chromosomes which are selected is dictated by the evolutionary strategy described in the previous section. There are many methods used to select the "best" candidates to seed the next generation and these are often referred to as "parent selection methods". Three possible parent selection methods are now described in this section to show how these selection methods can be implemented.

2.3.2.1 Tournament Selection

Tournament selection is a very simple selection strategy where a predefined number of the population is taken at random and the chromosome(s) with the highest fitness are promoted to seed the next generation. This process is looped until the next generation reaches the population size. Again this process relies on predefined parameters i.e. the number of chromosomes in each tournament and how many can be promoted from each tournament.

2.3.2.2 Elitist Selection (or Linear Rank Selection)

Elitist selection is again a very simple selection method, the population is ranked in order of their fitness and a predefined number of the fittest are selected to seed the next generation. The associated disadvantage with Elitist Selection is that it is required that the whole population is sorted each generation; at the cost of time and computational budget.

2.3.2.3 Roulette Wheel (or Proportionate Selection)

With the roulette wheel selection strategy the chromosomes are assigned a probability of being selected which is proportional to their fitness i.e. the fitter chromosomes have a higher chance of being selected. Then a random number source is used to mimic the "roulette wheel" and select a predefined number of chromosomes to be promoted.

[4] Genes which do not influence the operation of the chromosome.

There are again however a number of disadvantages associated with the "roulette wheel" selection method; which are now discussed. It cannot be directly used on minimisation problems (where the best fitness value is zero) as when the solutions all approach zero so do the differences in their fitness. Also as the process converges on a solution the differences in fitness again approach zero and so the selection method "loses direction".

2.3.3 Mutation and/or Crossover

Once the chromosomes which are to seed the next generation have been selected, it is then necessary to generate the next population; this is where the mutation and crossover operators are used. The first thing to note, is that some Evolutionary Computational strategies use mutation or crossover in isolation, whereas others strategies use both. It is also important to note, that mutation can be used to generate the next population from a number of initial chromosomes or it can be applied to a population which has already been generated.

There are again a number of parameters which control this stage of the evolutionary process; these parameters include: the percentage of the population which is generated by mutation, the percentage generated by crossover and to what extent the mutations alter the subject chromosome.

The most common mutation method, "Point Mutation", is to randomly select a gene within a chromosome to be mutated and within that gene randomly select a parameter which describes its operation. This parameter, within the gene, than has its current value changed to another randomly generated valid value. This operation can be applied multiple times to the same chromosome until a desired mutation rate has been achieved.

There are many types of crossover techniques, all of which try to create children with characteristics from both parents. It is also possible for the same parents to produce multiple children by combining their characteristics in different arrangements. For more information on different types of crossover techniques see the chapter entitled Crossover Techniques.

2.4 Termination Condition

As mentioned previously, the process of generating a new population from the old is iterated until a termination condition is reached. There are many different termination conditions which can be used, but it is important that at least one of them is reached after an appropriate length of time. This termination condition can be simply a specific number of iterated generations. This ensures that the process does not continue indefinitely.

Termination conditions can include: a solution is found which meets a given specification, a fixed number of generations has been reached, a real world time scale, computational budget, real world budget (time or funding), the best fitness has not improved for a given number of generations or physically inspecting the best solution. In many cases a selection of these conditions are used as well as conditions which may be more bespoke to the current task.

3 Background Literature

Here a selection of literature surrounding Genetic Programming and its wider context will be given and discussed. This literature makes up a large section of the background reading which was undertaken to understand the field of Cartesian Genetic Programming and where this project lies within it.

The Evolutionary World section describes where Genetic Programming lies within the wider field of Artificial Intelligence and gives the history of developments which led to Genetic Programming. The Cartesian Genetic Programming section discusses this extension to Genetic Programming; again providing its history. It will also discuss crossover when applied to Cartesian Genetic Programming.

The Current Cartesian Genetic Programming Developments section discusses extensions to Cartesian Genetic Programming which have been, and continue to be, developed. The Applications of Cartesian Genetic Programming section gives examples of when Cartesian Genetic Programming has been applied to real world applications.

The final section, Schema Theorem, introduces the Schema Theorem; a widely accepted theory as to why Genetic Algorithms are so powerful. Although this is only a descriptive explanation; no mathematical derivation is given.

3.1 Evolutionary World

At a high level, Genetic Programming sits under the umbrella of Artificial Intelligence. The term “Artificial Intelligence” was first coined by John McCarthy [3] at a conference at Dartmouth College in 1956; the first conference to address the concept of machine intelligence. John McCarthy, who is also the creator of lisp (a programming language often used for artificial intelligence), defines the term “Artificial Intelligence” as: "*the science and engineering of making intelligent machines, especially intelligent computer programs*" [4].

Evolutionary Computation is a sub field of Artificial Intelligence and is a term which encapsulates many related problem solving techniques all inspired by Darwinian Evolution [5]. It is used within the field of Artificial Intelligence due to its ability to actively solve problems without prior knowledge of the problem space and with little to no human

influence. As an interesting point, the notion of using artificial evolution was proposed by Alan Turing in 1948, the essay was dismissed by his employer (the grandson of Charles Darwin) as a “schoolboy essay” [6].

As with many discoveries in science, Evolutionary Computation was innovated independently by different groups around the world. As a result, Evolutionary Computation is often accredited to three separate pioneers who called their respective fields: Evolutionary Programming, Evolutionary Strategies and Genetic Algorithms. These three fields (in their traditional forms) are now discussed in further detail.

3.1.1 Evolutionary Programming

Dr Lawrence J Fogel et al introduced Evolutionary Programming in his book [7] “Artificial Intelligence through Simulated Evolution” in 1966. The book furthered his work achieved during his PhD “On the Organization of Intellect” which was awarded in 1964. Evolutionary Programming’s distinguishing features are that it employs a fixed internal structure and varies only the numerical parameters of the functions used. It also only uses mutation as its main evolutionary operator, rather than mutation and crossover. This is because each member of the population is viewed as a separate species and so is not compatible with other solutions. The next generation is created by mutating the members of the previous generations in what is sometimes referred to as a $(\mu + \mu)$-ES.

3.1.2 Evolutionary Strategies

A German computer scientist called Ingo Rechenberg introduced Evolutionary Strategies (or "Evolutionsstrategie") in his PhD dissertation [8] entitled "Optimierung technischer Systeme nach Prinzipien der biologischen Evolution" in 1971. His work was furthered by himself, Hans-Paul Schwefel et al and is continued to this day. In 2002, Hans-Paul Schwefel and Hans-Georg Beyer produced an article [9] called "Evolutionary Strategies - A comprehensive introduction", which provides substantial information surrounding Evolutionary Strategies; including the history and initial motivations.

Evolutionary Strategies originally used mutation and survival of the fittest for their evolutionary operators, crossover was not used until later [9]; it is still debated whether crossover aids the search process. The mutation used was referred to as Gaussian mutation; where the new value of the mutated gene is most likely to be given a value close to the

original, but can be given an increasingly different value with decreasing probability. This ensures that the mutation is not so strong that the search appears random, but not always too weak that the search gets trapped in local solutions.

3.1.3 Genetic Algorithms

Prof John Henry Holland introduced Genetic Algorithms in his highly influential book [10] "Adaptation in Natural and Artificial Systems"; first published in 1975. Genetic Algorithms operate by optimising a number of predetermined parameters. The fact that the number of parameters is often fixed, leads to simple crossover/recombination implementation; which is often employed alongside mutation.

Genetic Algorithms (as well as Evolutionary Strategies) are a subset of Evolutionary Algorithms, where Genetic Algorithms are the most popular Evolutionary Algorithm. Confusingly in some literature these terms seem to be used interchangeably. A general rule of thumb appears to be that the term "Genetic" implies that crossover/recombination is used more dominantly than mutation.

In Genetic Algorithms, crossover is considered highly important in reaching optimal solutions quickly, and often takes its inspiration from biology by using two parents to produce the child solutions. There is however research which suggests that using more than two parents often offers a greater advantage [11][12].

3.1.3.1 Genetic Programming

Genetic Programming is a specialised version of Genetic Algorithms; where the solutions are also represented as chromosomes comprising of individual genes. Unlike Genetic Algorithms however, the chromosomes represent a tree structure rather than a string of values. This tree structure does not have a fixed internal layout, with the connections also been described by the chromosomes. This results in the evolutionary process evolving the structure of the solution instead of just optimising a predefined structure. The functions at the nodes within the structure can also be varied during the evolutionary process. This is a huge advantage as the way in which problems are approached can be evolved, instead of just optimising a predetermined method. Dr John R Koza is often cited as the main contributor to Genetic Programming, due to his pioneering work published in his book [13]

“Genetic Programming: On the programming of computers by means of natural selection” in 1992.

The evolutionary operators for Genetic Programming are still mutation and crossover/recombination; except now care has to be taken to ensure that the mutation does not change the chromosome to a solution which is no longer valid. Crossover also becomes more complex, due to the chromosomes now be of variable length. As a result crossover is usually achieved by switching whole branches of the tree structures, which results in children with very different characteristics from their parents; quite different from reproduction in biology. It also leads to the strange phenomena of bloat, where the average length of the chromosomes in a population continues to increase as the Genetic Program is ran, with little to no improvement to the overall fitness. These issues are explained and solved in Koza’s book [13] and subsequent books. A book [14] by Riccardo Poli entitled “A Field Guide to Genetic Programming” is an excellent start for someone new to the field and wishes to create a Genetic Program. It covers the issues describe in this section and is free under the Creative Commons license.

3.2 Cartesian Genetic Programming

Cartesian Genetic Programming is a further specialisation of Genetic Programming introduced by Julian Miller and Peter Thomson [15]. The idea was born from using Evolutionary techniques to evolve digital circuits and was originally called Cartesian Genetic Programming in a paper [16]; which analysed and furthered this work in 1999. Cartesian Genetic Programming was formally introduced as a general form of Genetic Programming in 2000 [17].

Unlike regular Genetic Programming, which arrange their nodes in a tree structure, Cartesian Genetic Programs arrange their nodes in a grid structure indexed by coordinates (x and y); hence the term "Cartesian". This structure makes it possible for the re-use of nodes within the evolved programs; leading to increased efficiency as if the same value is needed multiple times it doesn't have to be recalculated. Another artefact of Cartesian Genetic Programming is the presence of redundancy in the chromosomes; these are genes which do not contribute to the overall operation of the chromosome. This redundancy has been shown by Julian Miller et al to be beneficial in the evolutionary process [18]; it is

suggested in the research that the best results were obtained with 95% redundancy in the chromosomes. It is also thought that this redundancy is the cause of the absence (or high reduction) of bloat within Cartesian Genetic Programming when evolving solutions [19]. This is a major advantage as bloat is a drawback of regular Genetic Programming and one which John Koza spent a lot time trying to resolve.

This redundancy is also thought to cause neutral drift [20] within the chromosomes; where the mutation of a gene does not always pose an advantage or disadvantage and hence can make it through to the next generation. This neutral drift is thought to significantly aid the search process. A paper [20] published by T. Yu et al, also propose that this neutrality would help solve needle-in-haystack type problems, where the fitness of a solution can only have a binary representation; correct or incorrect. These types of problems are very hard (likely impossible) to evaluate as no information is provided about the location of the "needles" from testing other locations. The results of T. Yu were questioned by M. Collins [21] who suggested that he could not repeat the results obtained by T. Yu. It is still thought however that neutral drift may beneficial to the evolutionary process [22]; even if it were shown not to aid needle-in-haystack type problems.

Cartesian Genetic Programming uses point mutation and elitism as its evolutionary operators [2], as described in Julian Millers book "Cartesian Genetic Programming". Point mutation is where a randomly chosen parameter of a randomly chosen gene is changed to a different randomly chosen valid state. Elitism is where the fittest member of the population is automatically promoted to the next generation. In Cartesian Genetic Programming the rest of the next generation is populated by mutated versions of the promoted elite chromosome, this is commonly referred to as a $(1 + \lambda)$-ES.

Crossover is thought to be highly important for the evolutionary process in Genetic Programming and is used extensively in the field. However, a study [16] showed one-point crossover to be disruptive to Cartesian Genetic Programming and as a result is not used in the majority of its applications. It is strange that such a close relative of Cartesian Genetic Programming uses crossover extensively and yet it itself does not. Although recently work by Janet Clegg [1] has investigated applying flat crossover to Cartesian Genetic Programming with promising results.

3.3 Current Cartesian Genetic Programming Developments

Cartesian Genetic Programming made its first appearance over ten years ago and has continued to be developed. This section introduces the two main developments which have been applied to Cartesian Genetic Programming: Embedded Cartesian Genetic Programming and Self-Modifying Cartesian Genetic Programming. Embedded Cartesian Genetic Programming adds the ability for the chromosomes to create sub modules within themselves; with the aim to aid the evolutionary process of certain problems. Self-Modifying Cartesian Genetic Programming allows the standard Cartesian Genetic Programming to produce solutions which change over time; similar to how animals grow in biology.

3.3.1 Embedded Cartesian Genetic Programming

When a human designs software or digital circuits it is common practice to split the problem up into smaller sub-sections, which are simpler to design, then use these sub-sections to realise the larger solution. For instance, if a processor were to be designed one would not start at a logic gate level, one would design adders, shift registers etc and then use these new components in the final design. The same concept could be used (and is used) for Evolutionary Computation. When Evolutionary Computation is employed, there are a number of given (often simple) allowed functions; these are the same as the logic gates in the given example. The Evolutionary Computational strategy could then be allowed to make its own new functions, from these given simple functions. It is thought that if this were possible, Evolutionary Computation could then tackle larger scale problems; something which it has struggled with in the past.

Embedded Cartesian Genetic Programming [23] was introduced by James Walker and Julian Miller in 2004 and was designed to achieve module acquisition as described in the previous paragraph. The module acquisition operates by randomly selecting two points in the chromosome and transforming that section into a new function which is used in its place. This function is then available for future use as a function by any other gene; if mutation brings it in to operation. After the best chromosomes are selected for the next generation, only the functions contained within these chromosomes are available as modules; this stops the increase of unwanted modules. Mutation still effects the modules but in a more controlled manor. Module acquisition is not allowed within other modules to stop the

growth of nested modules getting too large. Module outputs can never be mapped to module inputs; as this would create a "junk" module. There is also a mutation operation which causes the contents of the module to be inserted in its place in the chromosome. Julian Millers book provides detailed coverage of how Embedded Cartesian Genetic Programming operates [24].

In order to test Embedded Cartesian Genetic Programming, the task of evolving even parity functions using the logic gates: AND, OR, NAND and NOR was used [23]. This makes for a suitable experiment as calculating parity functions is extremely modular and therefore suited to Embedded Cartesian Genetic Programming. As an extra difficulty, it is known that implementing even parity functions is far simpler with the use of XOR or EXNOR functions; therefore these were not provided. The results of this experiment showed that for small parity problems the overhead of the module acquisition hindered the evolutionary process. However, for larger parity problems it proved a large advantage generating even parity function up to twenty times faster than regular Cartesian Genetic Programming.

3.3.2 Self-Modifying Cartesian Genetic Programming

To understand the motivation behind Self-Modifying Cartesian Genetic Programming the concept of genotypes and phenotypes must be introduced. The genotype (or chromosome) is a description of a solution to a given problem rather than the solution itself, this description has to be decoded to produce the solution; this solution is the phenotype. In many forms of Evolutionary Computation the mapping between the genotype and phenotype is direct; the mapping is slightly more abstract however in Cartesian Genetic Programming.

In biological systems the mapping between the genotype and the phenotype changes over time e.g. as animals develop from a few cells to their final form. This ability to change the phenotype over time was the inspiration for Self-Modifying Cartesian Genetic Programming. Self-Modifying Cartesian Genetic Programming was introduced in 2007 [25] by Simon Harling et al and is discussed in detail in Julian Millers book [26]. The genotype changes the phenotypes over time with the addition of new functions which when executed change the original structure of the genotype. The new structure may contain further self-modifying functions which are then executed on the next evaluation of the phenotype. This results in a

series of phenotypes all of which can have different functionality. This technique is useful when trying to solve a series of computational problems rather than a single instance. Another feature of Self-Modifying Cartesian Genetic Programming is its ability to acquire more inputs and produce more outputs as it develops.

A strong advantage of Self-Modifying Cartesian Genetic Programming is its ability to find general solutions. As proof of this, the task of evolving even parity functions using logic gates was used; see the section on Embedded Cartesian Genetic Programming for further details as to why this is such an interesting problem. The Self-Modifying Cartesian Genetic Program successfully generated a program which could calculate the parity of an arbitrary sized bit string (tested to 24 bits) by iterating the self-modifying process until the corresponding iteration was reached. From this a general solution was derived to the even bit parity problem; these results were published in 2010 [27].

It was also shown that the use of Self-Modifying Cartesian Genetic Programming on problems which do not benefit from the self-modifying aspect, were not hindered by its presence [28]. Julian Miller indicates in his book that Self-Modifying Cartesian Genetic Programming could "be a suitable replacement for the classical Cartesian Genetic Programming model" [26]. At the time of writing this, a new version of Self-Modifying Cartesian Genetic Programming is under development which aims to simplify and optimise this technique.

3.4 Applications of Cartesian Genetic Programming

Cartesian Genetic Programming is relatively new in the field of Evolutionary Computation, but there are still many examples of its application; a selection of which are described in this section.

3.4.1 Co-Evolution

Gul Muhammad Khan et al produced a paper [29] describing the application of Cartesian Genetic Programming to the co-evolution of two agents in a predator-prey relationship. The behaviour of these agents was implemented by a biologically inspired neural network, which had the ability to adapt itself during its “life time". It was found that the agents were able to learn from their environment and pass on this knowledge to the next generation. This result

inspired them to continue their work in trying “to see if it is possible to evolve a general capability for learning”.

Another example of co-evolution was shown by Joseph A. Rothermich and Julian F. Miller [30] in an experiment which investigated the emergence of multicellular organisms using evolution. They created a scenario where “cells” existed in a world containing areas of “food”. The “cells” and “food” released chemical signals which could be detected by the other “cells”. The “cells” were given the option of dividing (at a small energy cost) into two of the same cell; with slight mutation. It was found that over time the cells which did divide were more prominent than those which did not. This was thought to be because one of many cells were more likely to find “food” and survive than a single “cell”; hence more likely to pass on their genes.

3.4.2 Hardware Implementations

The implementation of Genetic Programs on graphics processor units has recently become a topic of interest in the Evolutionary Computational world. This is due to the significant decrease in computational time when multiple processes can be run in parallel. Simon Harding [31] used graphics processor units to evolve noise reduction filters which were notably better than standard median filters; which are often used for this application.

Work by Zdenek Vasicek and Lukas Sekanina [32] also found that Cartesian Genetic Programs implemented on FPGA’s significantly decreased computational time; 30-40 times compared to highly optimised software implementations. Their work surrounded the solving of symbolic regression problems and the implementation of digital logic circuits.

3.4.3 Synthesis of Boolean Logic

The widest application of Cartesian Genetic Programming has been towards the evolution of digital circuits [16]; such as even parity generators previously described. This work has also been extended by Zbysek Gajda and Lukas Sekanina [33] by including polymorphic logic gates; gates which change their function depending upon a control signal. Their work was motivated by the limitations of current polymorphic circuit design techniques and concluded that a combination of conventional design and Cartesian Genetic Programming created the most efficient circuits.

3.5 Schema Theorem

The Schema Theorem is a widely excepted explanation as to why Genetic Algorithms are so powerful when applied to solving optimisation problems. The Schema Theorem was proposed by the founder of Genetic Algorithms John Holland [10]; see the Genetic Algorithms section. For a critical review of whether the Schema Theorem can truly explain the effectiveness of Genetic Algorithms see [34] by Lee Altenberg.

The concept behind Schema Theorem is that all chromosomes fall into multiple schemata (sets of similar chromosomes). When each chromosome is evaluated, all of the other possible chromosomes belonging to the sets which contain that chromosome are to some extent also evaluated. This adds a level of implicit parallelism to the search process. As the weaker chromosomes are removed by the selection method employed, the range of different schema is reduced and the average fitness of each schema increased. It is then likely that the use of mutation and/or crossover will improve the fitness of each chromosome. Genetic Algorithms therefore navigate the search space using what appear to be random methods, but actually perform far better than a simple random search.

4 Cartesian Genetic Programming

This chapter describes the inner workings of a Cartesian Genetic Program; for a brief history and related background literature see the chapter entitled Background Literature. The information given in this chapter is mainly taken from the Cartesian Genetic Programming section of Julian Miller's book [15].

Cartesian Genetic Programs are often described as "directional acyclic graphs" i.e. a graph structure which is unidirectional and which does not contain any feedback. The graph in this case is a two dimensional grid of nodes indexed by x and y coordinates; hence "Cartesian". The functionality of each of these nodes is described by its corresponding gene which uses integer values to represent each parameter. These parameters describe: where the node gathers its inputs, the operation performed by the node and where the user can obtain the global outputs. Each node can obtain their inputs from previous nodes or from the global inputs. Cartesian Genetic Programs are often depicted by Figure 2, taken again from Julian Millers book [15].

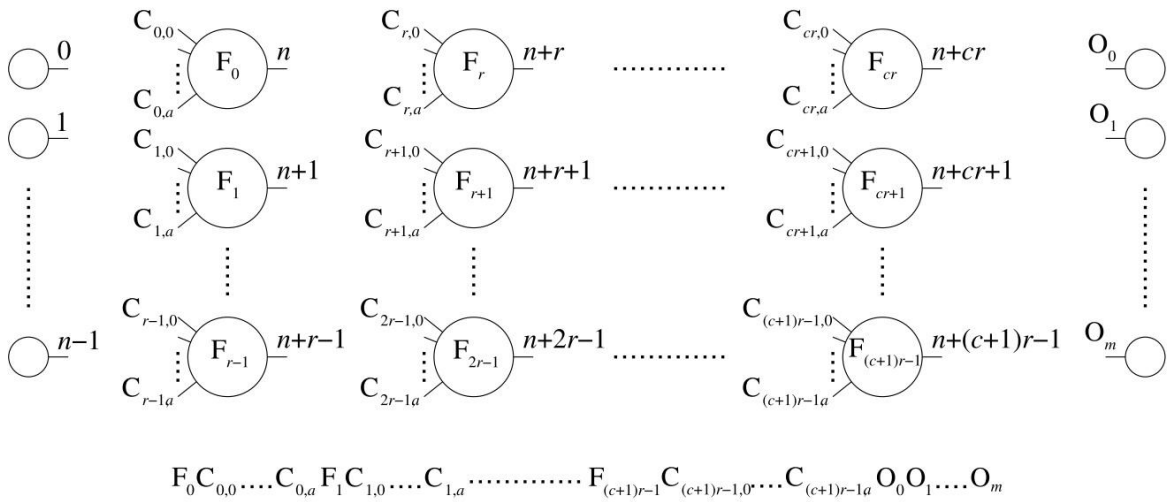

Figure 2 Depiction of the Cartesian Genetic Program structure

Figure 2 shows how each node is indexed by its corresponding position in the graph structure and that each node can only acquire inputs from previous columns or the global inputs. The lower equation in Figure 2 shows the general form of a Cartesian Genetic Programs chromosome (or genotype); with each node been assigned a function (F) and a list

of inputs (C). The outputs (O) are then be taken from any of the internal nodes. The parameter which describes the function of each node is called the "Function Gene"; this is an integer value which is used as an index in a look-up-table. The parameters which describe where each node obtains their inputs are referred to as “Connection Genes"; these are also integer values which index other nodes.

The Cartesian Genetic Program uses three parameters to describe its structure: number of columns, number of rows and levels back. The number of rows and columns describe the "shape" of the graph; with their product dictating the maximum number of functional nodes within the program. Quite often however, the number of rows is set to one, as this structure can implement any arrangement possible with multiple rows (provided the number of functional nodes remains the same). The levels back parameter describes how many columns back each node can acquire its inputs; it therefore controls the connectivity of the program. Setting levels back to equal the number of columns is described as been fully connected.

The following subsections describe how to: initiate the first generation, decode the evolved chromosomes, evaluate the fitnesses and finally generate the future generations.

4.1 Creating the Initial Population

An important property of all Evolutionary Computation is that the described solutions (or chromosomes) can always be evaluated. To achieve this, the chromosomes of Cartesian Genetic Programs are only allowed parameter values within certain ranges. These ranges must be adhered to when initialising the first population and when generating new future populations.

As with many forms of Evolutionary Computation, Cartesian Genetic Programs create their initial population by assigning random values to the chromosomes parameters. The "Connection Genes" for each node is taken as a random value between zero and the number of previous nodes plus the number of global inputs. This ensures that the acyclic criterion is preserved. The "Function Gene" for each of the nodes is taken as a random value between zero and the number of possible functions available. The outputs are taken as a random value between the number of inputs and the number of nodes. Following these rules ensures that valid chromosomes are always generated.

4.2 Decoding the Chromosomes

The decoding of the chromosomes is normally achieved in a two part process; first derive which nodes are active in generating the outputs and then calculate the outputs for given inputs. Active nodes are those nodes whose presence is necessary in generating the outputs. In a Cartesian Genetic Programs chromosome there can be many inactive nodes and so conducting all of the internal calculations would be a waste of computational time.

One method of evaluating which nodes are active is to take the output nodes and store their inputs in an "active node list". The nodes placed on the "active node list" also have their input nodes added to the list. This process is then repeated for each node which is placed on the list until the inputs are reached. The generated "active node list" is then the complete set of nodes necessary for generating the outputs.

Once the "active node list" has been generated, the Cartesian Genetic Program can be used to generate outputs for a range of inputs. The mapping between the inputs and outputs, or just the outputs themselves, is often what is used to generate the fitness for each chromosome. The outputs can be calculated by first creating a blank matrix of the same dimensions as the Cartesian Genetic Program. This "output matrix" is to store the intermediate values generated by each active node. The first active node then generates its outputs from the global inputs using its corresponding function; this output is then saved in the "output matrix". Subsequent active nodes can then gather their inputs from, and store their outputs to, the "output matrix". The final outputs can then be looked up from the complete “output matrix".

4.3 Creating the Next Generation

Conventionally Cartesian Genetic Programs implement a (1 + λ)-ES to create the next generation. The λ members of the population are generated by mutating the single elite member of the previous generation. When selecting the next elite chromosome, children are chosen over the parent if they have equal fitness.

The mutation method used by Cartesian Genetic Programming is called "Point Mutation"; see section "Mutation and/or Crossover" for further details. Cartesian Genetic Programming follows the same rules used when generating the initial population as it does when mutating a specific parameter of a given gene. This ensures that valid solutions are always generated.

5 Crossover Techniques

As described in the section entitled Mutation and/or Crossover, crossover can be used to generate the next generation from a selected sample of the previous generation. It can be used in conjunction with or without the mutation operator. As this project continues the work of Janet Clegg [1][35], the same crossover technique is the main focus of this project. The crossover technique used by Clegg is called BLX-0 or Flat Crossover and is described in this section along with other common crossover strategies.

5.1 Point Crossover

Point crossover is achieved by splitting each parent chromosome into sections; by the placement of a point(s). The children are then generated by taking different sections from each of the parents to form a new chromosome.

5.1.1 Single-Point Crossover

Single-Point Crossover is point crossover, where the parent chromosomes are split into two sections. When the crossover point always splits the parent chromosomes into equal sections, it is referred to as Mid-Point Crossover; see Figure 3. When the single crossover point is selected at random it is referred to as "Simple Crossover".

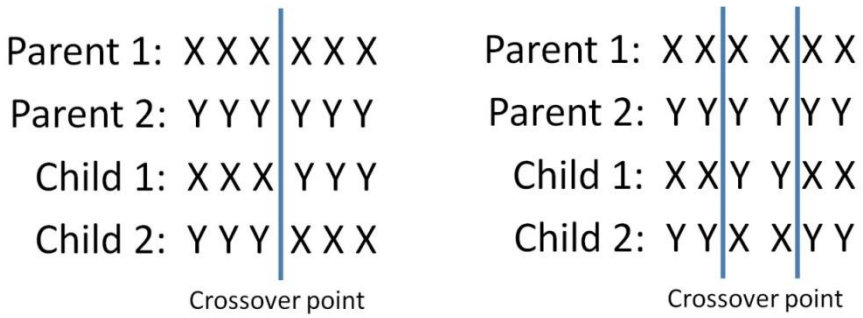

Figure 3 Depiction of "Mid-Point Crossover" (left) and "Two-Point Crossover" (right)

5.1.2 Two-Point Crossover

Two-Point Crossover is an extension on single point crossover, which employs two crossover points instead of one. Again the crossover points can be set to split the parent chromosomes into equal sections, or can be picked at random; Two-Point Crossover is also shown in Figure 3.

5.1.3 Cut and Splice Crossover

Cut and Slice Crossover is similar to Single-Point Crossover, only different random crossover points are chosen for each parent. This results in variable length children chromosomes, for this reason it is not often used. Cut and Splice Crossover is shown in Figure 4.

Figure 4 Depiction of "Cut and Splice Crossover"

5.2 Uniform or Discrete Crossover

Uniform or Discrete Crossover is employed by splitting the parent chromosomes into many sections; usually of even length (but this is not essential). The children then take their subsections from either parent with a given probability; usually 0.5 to ensure they contain an even amount of genetic material from both parents. See Figure 5 for an example of Uniform Crossover.

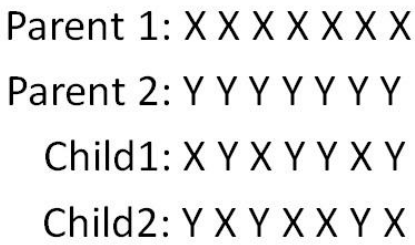

Figure 5 Depiction of "Uniform Crossover"

5.3 BLX-0 or Flat Crossover

BLX-0 or Flat Crossover is the first crossover technique which does not simply pick the child's chromosomes from the parents. Instead it calculates new values based upon the parents chromosomes; for this reason it cannot be used when the chromosomes are represented as binary strings.

Flat Crossover was first defined by Nicholas J. Radcliffe in 1991 [36] and was "affectionately known as Top Hat". The technique takes the two floating point representations of the gene

parameters and randomly selects a value between the two. The child's parameters are calculated by[5]:

$$C_i^k = P_i^1 + r_i(P_i^2 - P_i^1) \qquad if\ P_i^1 < P_i^2$$

$$C_i^k = P_i^2 + r_i(P_i^1 - P_i^2) \qquad if\ P_i^2 < P_i^1$$

$$C_i^k = P_i^1 = P_i^2 \qquad if\ P_i^1 = P_i^2$$

Where the parameters of parent one are indexed by $(P_1^1, P_2^1, \dots, P_n^1)$ and the parameters of parent two are indexed by $(P_1^2, P_2^2, \dots, P_n^2)$. Child one's parameters are then indexed by $(C_1^1, C_2^1, \dots, C_n^1)$ and child two's by $(C_1^2, C_2^2, \dots, C_n^2)$. Finally r_i is a random value such that $0 < r_i < 1$. If for example, there were two children to be produced k would index each child with a value of 1,2 and i would index all of the corresponding parameters.

It should be noted that for this type of crossover, the chromosomes are represented in a floating point form; as opposed to the usual integer form. This means that all of the values which describe each node (input locations and/or functionality) are floating point values between zero and one. For this reason an additional decoding level is required to convert back into the corresponding integer form. The equations to calculate the function index and node index from the floating point form are shown here[6]:

$$Function\ Index = floor(gene_i * func_{total})$$

$$Node\ Index = floor(gene_i * nodeterm_j)$$

Where $gene_i$ indexes each gene by i, $func_{total}$ is the total number of functions and $nodeterm_j$ is the number of possible input nodes available to the current node under inspection. The floor operator truncates the given arguments e.g. the floor of 4.3 would be four.

5.4 BLX-α or Arithmetic Crossover

BLX-α or Arithmetic Crossover is another example of a crossover technique which cannot be applied to chromosomes represented as a binary string. It was created by L. J. Eshelman et

[5] This form of the Flat Crossover equations was taken from a BSc cause taught by Janet Clegg at the University of York.
[6] Also taken from Janet Clegg's taught lecture series.

al [37] in 1993 and represents a more general form of Flat Crossover. Similarly with Flat Crossover randomly selecting parameter values for the children's chromosomes somewhere between the two parent values, Arithmetic Crossover picks a value between its parent's values plus a small margin. This margin is dictated by α and by setting the value of α to zero it implements Flat Crossover.

Each child's parameter can then be calculated using the following:

$$\Delta = \alpha(Y - X)$$

$$C = rand(X - \Delta, Y + \Delta)$$

Where X and Y represent the two parent parameter values and C is the calculated child's parameter value. α is set by the user to vary how much the child's parameter values can lay outside the parents. The rand operation then selects a random value between the given parameters. This form of the equation holds for when X < Y.

A disadvantage of this technique is that for the case where α is not equal to zero, it is possible for the child's parameter to be assigned a value outside of a valid range; in this case a repair algorithm has to be employed to fix/prevent this occurrence.

6 The Investigation

This chapter describes the overall aims of the project and the objectives set to achieve these given aims. A timeline created to meet these objectives, described using a Gantt chart, can be found in the chapter entitled Project Timeline.

The Aims section outlines the aims of the project at a high level and the Objectives section describes in more detail how these aims will be achieved. The Procedure section describes the process used to evaluate the effectiveness of the new crossover technique; as used by Janet Clegg et al [1]. The Hardware and Software Requirements section then discusses the tools needed to implement the given objectives. Finally the Risk Assessment discusses the possible risks endangering the project along with possible contingencies.

6.1 Aims

Here the aims of the overall project are stated. These aims are split into primary and secondary subheadings. Primary Aims 1 and 2 and Secondary Aim 1 make reference to a paper published by Janet Clegg et al [1]. The author's project is an extension of the work described in Janet Clegg's paper and therefore the paper is included in Appendix A for the reader's reference.

6.1.1 Primary Aims

1) Evaluate whether the crossover technique used by Janet Clegg offers a statistically significant decrease in convergence time when solving a range of problems using Cartesian Genetic Programming.
2) Evaluate whether the floating point representation and tournament selection scheme required for the new crossover technique changes the behaviour of the Cartesian Genetic Program.

6.1.2 Secondary Aims

1) To study further the effects of the parameters governing the effectiveness of Cartesian Genetic Programming, more specifically when implementing the new crossover technique.

2) To apply Cartesian Genetic Programming to a range of problems to which it has not previously been applied.

6.2 Objectives

This section describes the objectives of this project; split into primary and secondary subsections. By completing the given objectives the previously stated aim will be met.

6.2.1 Primary Objectives

1) Investigate at least three different search problems using Cartesian Genetic Programming with and without the new crossover technique and evaluate/compare their relative effectiveness using multiple statistical methods.
2) Apply Cartesian Genetic Programming to the same search problems using the floating point chromosome representation necessary for the new crossover technique, but without the use of the crossover technique. Compare the results obtained with the integer form Cartesian Genetic Programming.
3) Apply the floating point form of Cartesian Genetic Programming to the same search problems using the tournament selection method necessary for the new crossover technique; but without the use of the new crossover technique. Compare the results obtained with the floating point form of the Cartesian Genetic Programming without the tournament selection method.

6.2.2 Secondary Objectives

1) When evaluating the search problems, systematically optimise the Cartesian Genetic Program's evolutionary parameters for all experiments.
2) Evaluate what can be learnt from the optimised parameters which appear to be most suitable.
3) Where possible, and if time permits, select search problems to which Cartesian Genetic Programming has not yet been applied.
4) If time permits, and the results are worthy, publish the results of this project in a scientific journal article.

6.3 Procedure

This section describes the planned procedure for carrying out the given objectives. It was decided that once a generalised Cartesian Genetic Program was created, a selection of test

cases (search problems) were to be evaluated. For each of these test cases the following methods would be compared:

1) Cartesian Genetic Programming without tournament selection using an integer chromosome representation.
2) Cartesian Genetic Programming without tournament selection using a floating point chromosome representation.
3) Cartesian Genetic Programming with tournament selection using a floating point chromosome representation.
4) Cartesian Genetic Programming using BLX-0 crossover, which requires a floating point chromosome representation and tournament selection.

As the crossover technique used by Janet Clegg requires the use of a floating point chromosome representation, 1) and 2) would evaluate whether this new encoding is damaging to the search process. Another requirement of the new crossover technique is the use of tournament selection, 1) and 3) would therefore evaluate the effect of a tournament selection scheme. The two previous comparisons would evaluate whether the conditions necessary for the new crossover technique are damaging, beneficial or neutral to the search process. Once this was known, the new crossover technique, 4), could be compared to the integer form of Cartesian Genetic Programming. This would enable a more insightful evaluation than a simple comparison between Cartesian Genetic Programming with and without the new crossover technique. All the comparisons described here were planned to be completed for a range of test cases so significant conclusions can be drawn.

When each version of the Cartesian Genetic Program is applied to each test case, the parameters which govern the search process are systematically optimised; in order to ensure a fair comparison between the different strategies. The process of optimising the parameters is described in Appendix B. It should be noted that these values are likely not to be optimum, as finding the optimum parameters for Evolutionary Computation is a search process in its own right; however this process should produce suitable values.

6.4 Hardware and Software Requirements

As this project is software/research related there are no hardware requirements; except the use of a PC. The freely available Eclipse IDE for Java [38] was chosen as the main

development software; with Mathworks MATLAB [39] also used as a "scratch-pad". It is understood that most Evolutionary Strategies are implemented using languages which are known to be computationally efficient. Java was chosen however, due to the simple description of a Genetic Program in an object oriented language and because of personal past experience. A Dropbox [40] account was created to save documents and code on a remote server. This server could then be synced with local folders on multiple computers, ensuring that all files are backed up and enabling coding from multiple locations without having to deal with code management.

6.5 Risk Assessment

As this is a software/research related project, the main risks surround implications for the project itself rather than real world "health and safety" concerns. The possible risks thought to endanger the project include: loosing digital files, poor code management, project overrun, project underrun and encountering severe difficulties. Each of which shall now be discussed with respect to the severity and likeliness of the risk; along with the prevention tactics used to avoid/accommodate these risks.

6.5.1 Loosing Files

The possibility of losing digital files was very high due to the ease of deletion and the possibility of computer failure. The severity of losing work becomes heightened as the project progresses; due to the fact there is more to lose. Losing work towards the end of the project would be massively detrimental and possibly an un-correctable situation. The likeliness of losing work is quite high; if sufficient care is not taken when storing all digital files. The methods used to prevent the losing of digital files were: saving all work to an online Dropbox account [40] and conducting weekly backups to an external hard drive.

6.5.2 Poor Code Management

Specific code management is essential for all but the smallest projects. The management of code has two parts: preventing the loss of completed works and dealing with version control. The losing of completed code followed the same method outline as the previous section, Losing Files. Version control is important because during code development it would quite likely that changes to the code may result in a program which no longer operates correctly; in these situations it can be highly beneficial to revert back to a working

version. The likeliness of needing to revert to a previous state during development is very high. To enable this functionality, code was downloaded to the current machines hard drive during editing and only re-uploaded when fully operational. If at any point the code saved to the online Dropbox was found to be in error, it would be reverted to the last weeks backup; with the maximum loss of seven days work.

6.5.3 Project Overrun

A project overrun is the situation where the workload to be completed is no longer possible in the available time frame. The severity of the project not been completed on time was considered very high as there was a fixed deadline to be adhered to. Project overruns are very common due to the difficulty of correctly anticipating how long each aspect of the project will take; and anticipating all aspects of the project. In order to prevent the project overrunning, the project timeline was constructed to include activities which would be reduced and/or removed if necessary.

6.5.4 Project Underrun

A project underrun is the opposite of a project overrun; it becomes apparent that there is an insufficient workload to be completed and the project prematurely comes to an end. This situation is far less severe than a project overrun; as a final report will be completed before the deadline. It does however indicate poor project management and that more could have been achieved in the available time. The likeliness of an underrun occurring is far less than an overrun, but still possible due to the inaccuracies in predicting how long each aspect of the project will take. A project underrun is prevented by including activities in the project timeline which would be added if time become available.

6.5.5 Encountering Difficulties

As with all projects, it was possible that a problem would be reached which the author could not resolve; this may be: coding related, logic related, research related or mathematical. This situation could be quite damaging if the difficulty encountered relates to the core of the project. The likelihood of encountering a major difficulty is quite high due to the testing nature of final year projects. If a difficulty is encountered the following steps were to be taken: search the internet for related issues, search the library for related topics and finally

refer to the project supervisors. If the difficulty remains than the project would have to be adapted to avoid its constraint.

7 Possible Test Cases

This chapter describes a range of different possible test cases for which the new crossover technique proposed by Janet Clegg [1] could be evaluated. The evaluation would be via a comparison with normal Cartesian Genetic Programming. There are two main themes of test case shown here; those which optimise a set of parameters (7.1, 7.7 and 7.10) and those which evolve programs which solves a given problem (7.2, 7.3, 7.4, 7.5, 7.6, 7.8 and 7.9). One of the advantages of Cartesian Genetic Programming is its ability to be applied to evolving programs rather than simply optimising parameters; this is why most of the test cases fall into this category. The test case 7.9 involves evolving a finite state machine; an application of Cartesian Genetic Programming which is relatively unexplored in the literature.

7.1 Function Optimisation

Many real world (and theoretical) problems can be reduced to the task of optimising a number of predefined parameters. Cartesian Genetic Programming can be applied to optimising parameters by an arrangement where the outputs of a particular chromosome are then the given inputs to a function. The function is then run, and the returned result used to determine the fitness.

For this test case the functions to be optimised are multi-dimensional graphs, where the aim is to find the set of coordinates which locate the minimum point. The inputs to the Cartesian Genetic Program are a set of arbitrary constants. The possible Cartesian Genetic Programs functions are a set of mathematical operands with their outputs limited to plus/minus one. The generated outputs for the given chromosome are scaled to meet the range of possible graph function inputs. When finding the minimum point in the graphs, the lowest returned value represents the fittest chromosome.

Possible functions for this test case are: the Griewank Function, the Shekel Function and the Rosenbrock function; these functions are now described in further detail.

The Shekel Function was developed as a test function by J. Shekel for function optimization techniques [41]. The function is described by the following equation and matrices of constants:

$$f(x) = -\sum_{i=1}^{M} \frac{1}{(x - A_i)(x - A_i)^T + C_i}$$

$0 \leq x_i \leq 10$

$$A = \begin{bmatrix} 4 & 4 & 4 & 4 \\ 1 & 1 & 1 & 1 \\ 8 & 8 & 8 & 8 \\ 6 & 6 & 6 & 6 \\ 3 & 7 & 3 & 7 \\ 2 & 9 & 2 & 9 \\ 5 & 5 & 3 & 3 \\ 8 & 1 & 8 & 1 \\ 6 & 2 & 6 & 2 \\ 7 & 3.6 & 7 & 3.6 \end{bmatrix} \qquad C = \begin{bmatrix} 0.1 \\ 0.2 \\ 0.2 \\ 0.4 \\ 0.4 \\ 0.6 \\ 0.3 \\ 0.7 \\ 0.5 \\ 0.5 \end{bmatrix}$$

The Shekel function has its global minimum value of -10.5364098167 at the coordinates 4.00075, 4.00059, 3.99966 and 3.99951. For the readers interest a limited view of the Shekel Function is shown in Figure 6.

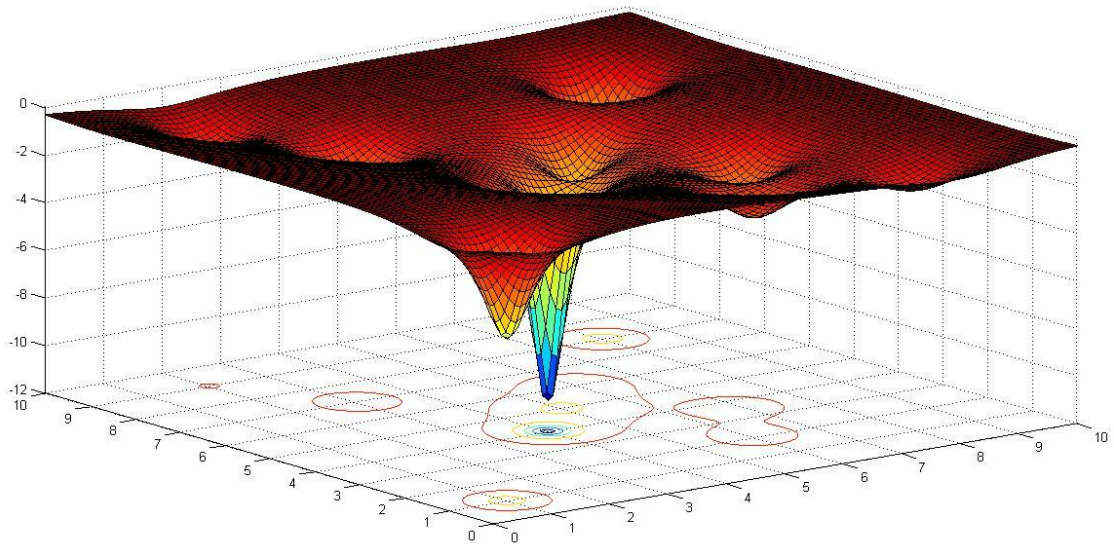

Figure 6 The Shekel Function limited to two variables

The Griewank function was developed by A.O. Griewank [42] and is an interesting equation as the number of minima grows exponentially with the number of dimensions used. The function is described by the following equation:

$$f(x_1, \ldots, x_n) = 1 + \left(\frac{1}{4000}\right)\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right)$$

$-600 \leq x_i \leq 600$

The Griewank function has its global minimum value of zero at the point where all of the coordinates are also zero. The value of n determines the number of variables used. The Griewank function is shown in Figure 7 with n set as two.

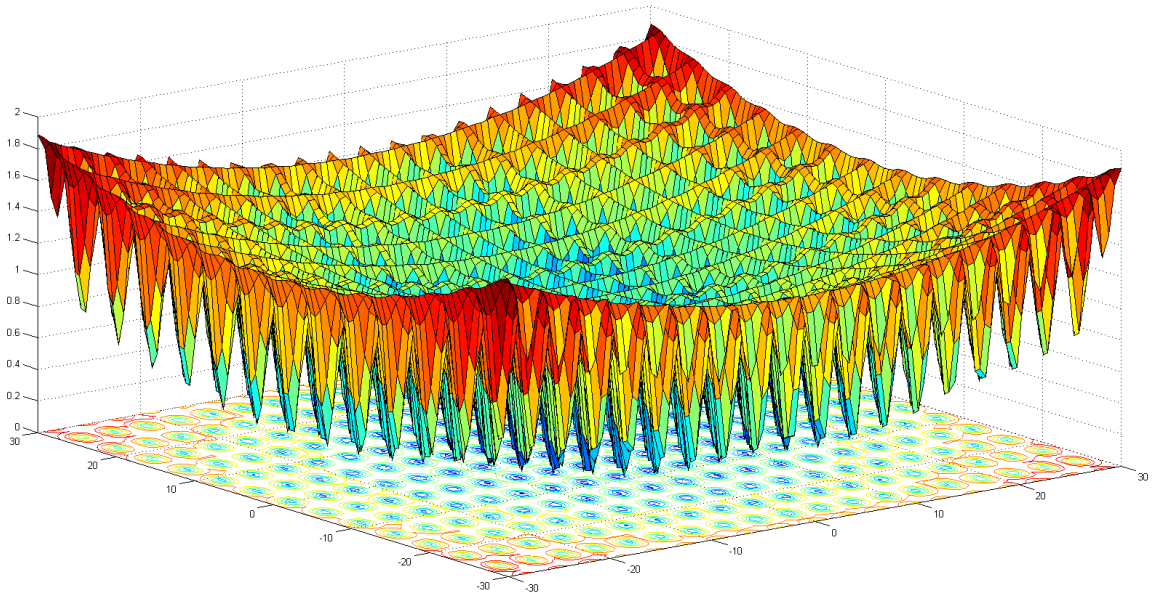

Figure 7 The Griewank function

The Rosenbrock function[7] is a commonly used optimization landscape introduced by Howard H. Rosenbrock in 1960 [43]. The function has an interesting characteristic of its main valley been easily found, but the minima of the whole function very difficult; due to the near flat bottom of the valley. The two variable function is described by the following equation:

[7] Also communally referred to as the Rosenbrock's valley or the Rosenbrock's banana function.

$$f(x,y) = (1-x)^2 + 100(y-x^2)^2$$

$-2 \leq x \leq 2$
$-2 \leq y \leq 2$

The Rosenbrock function has a minimum value of zero at the position (x,y) = (1,1) and is shown in Figure 8.

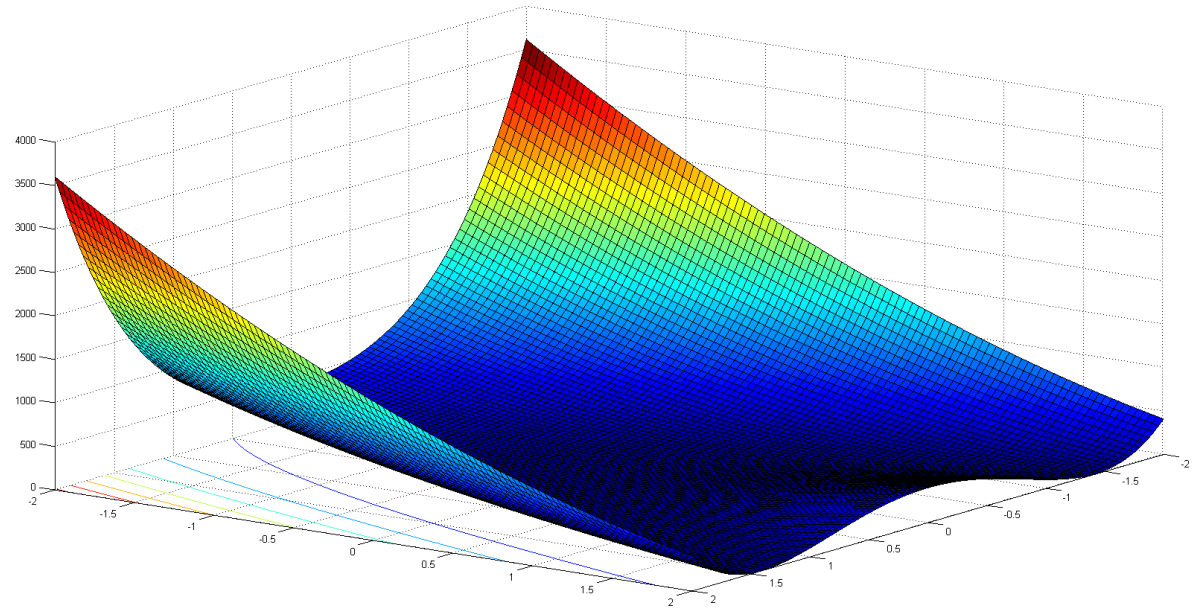

Figure 8 The Rosenbrock function

A possible extension to this investigation is to mark the positions inspected by the Cartesian Genetic Program on the graph being evaluated. The results may just be interesting to observe, but they may also show additional information on how evolution has approached the problem.

7.2 Symbolic Regression (Curve Fitting)

It is often necessary to plot a line of best fit to a given sample of data. This process can: show trends, describe whole data sets with one equation and sometimes reveal hidden relationships within the data which offer insight into the inner workings of the data source. Cartesian Genetic Programming (and more generally Genetic Programming) is well suited to symbolic regression, as it is perfectly adapted to trying different combinations of mathematical operation in order to find the best solution. This is different from Genetic Algorithms, which in its simplest form only vary the weightings of predefined mathematical

operations. The fitness is often taken as the sum of the differences between the predicted values and the actual values; resulting in zero representing a perfect solution.

For this test case the two equations used by Janet Clegg [1], originally taken from Koza's work [13], would be evaluated. These equations would be used to generate a data set for which the Cartesian Genetic Program would try to fit its own equations too. It is also possible to test these equations with the addition of noise; to mimic a real world situation with non-perfect data.

$$f(x) = x^6 - 2x^4 + x^2$$

$$f(x) = x^5 - 2x^3 + x$$

The inputs to the Cartesian Genetic Program would be the independent variables used in the equation or experiment. The functions used within the Cartesian Genetic Programming nodes would be simple mathematical operands: addition, subtraction, multiplication and division.

7.3 Synthesis of Boolean logic

Designing the necessary network of Boolean logic gates can be tedious for all but the simplest truth tables. There are algorithms which can be followed to arrive at solutions, but they are "long winded", use only certain logic gates and often do not lead to the “best” solution.

From the start, Cartesian Genetic Programming used the synthesis of Boolean logic test case to demonstrate its effectiveness [16]. This type of problem requires that the algorithm produces the network of logic gates necessary to implement a given truth table; something Cartesian Genetic Programming is perfectly adapted to do. It therefore makes a suitable test case for historic reasons and for the fact it utilises a strong quality of Cartesian Genetic Programming.

Here the test case would be to implement the logic for a given truth table e.g. that of a two bit multiplier. The inputs to the Cartesian Genetic Program would be the sets of inputs from the truth table and the outputs of the Cartesian Genetic Program would be the outputs of that configuration of logic gates. The fitness function would award points for every

incorrectly implemented line of the truth table. More complex fitness functions can be implemented e.g. by awarding points for using fewer logic gates thus promoting smaller circuits. The node functions themselves would be Boolean logic gates.

7.4 Wall Follower

A "Wall Follower" is a well known standard maze solving algorithm which reacts only to its immediate surroundings; therefore having no memory of its previous positions. This test case is to evolve the logic network which takes inputs from its surroundings and then has to navigate around a "world" gaining points when following walls. The "world" in this case is a two dimensional grid with squares representing free space or walls, see Figure 9 (the blue square represents the starting position). The "Wall Follower" would be able to "see" the eight squares surrounding its current position and react by moving: up, down, left or right. These possible moves are mapped to the two binary outputs of the Cartesian Genetic Program by assigning binary representations to the moves e.g. 11 could represent a move to the right. The "Wall Follower" would not be able to walk through walls and would be allowed a given number of moves to gain as many points as possible. Each section of the wall would only contain one available point; to prevent the solution of simply moving back and forth along the same path. Although in this scenario we are not trying to solve a maze, the "Wall Follower Algorithm" should provide a solution to this task[8].

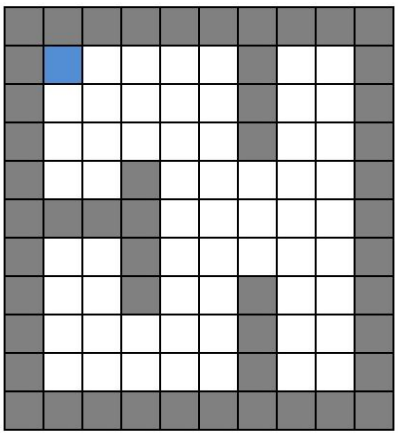

Figure 9 Possible layout of the "wall followers" world

This process is similar to the Synthesis of Boolean logic, except the correct truth table is not known in advance; instead this is also being evolved along with its implementation. A

[8] Specifically mazes with no internal loops.

possible advantage of this test case is that it would make an interesting example to show in the final presentation; the effectiveness of the current solution can be shown at different generations via an animation.

7.5 Wall Avoider

The "Wall Avoider" test case is similar to the "Wall Follower", except the aim is now to navigate across the "world" without "bumping" into any walls. Points are awarded for every movement which is made towards the "finish line" (along the x-axis). The game is terminated: if the finish line is reached, after a given number of movements, or if an attempt is made to walk through a wall. An example of a possible "world" is shown in Figure 10, where the yellow checkers represent the "finish line" and the blue square represents the starting position.

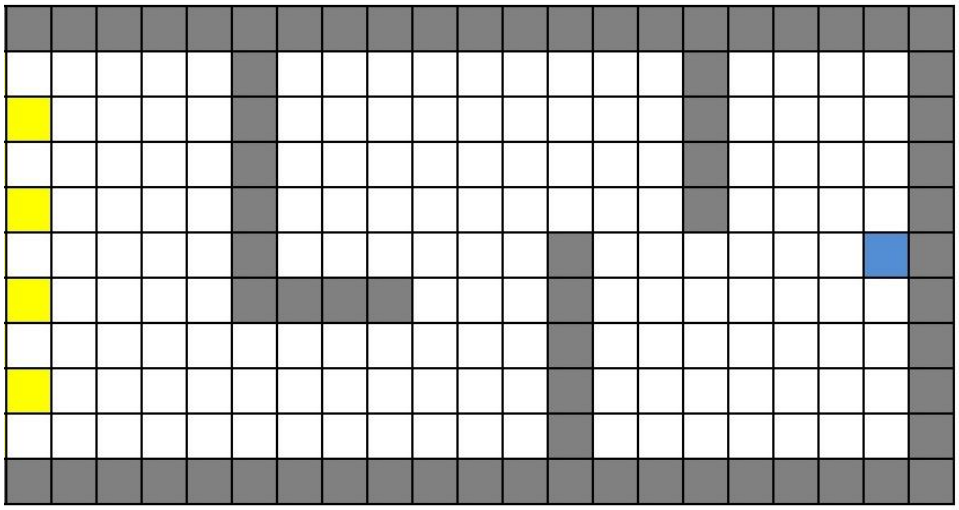

Figure 10 Possible layout of the "Wall Avoiders" world

In order to make this task possible, it is thought that the crudest memory is necessary; this memory is of the previous move undertaken. Without memory, the "L" shape in the "world" is thought to be an impossible challenge. Although it is likely that other sections are also too difficult without a more extensive knowledge of the surrounding "world". To implement this crude memory the previous move (or previous output) is to be fed back to the available inputs of the Cartesian Genetic Program. It is also assumed that when the game is started the previous move was a step towards the "finish line" (left in the shown figure). This test case may require a large number of generations before a solution is found, but it is thought that a solution will eventually be found (at least to some extent).

7.6 Fibonacci/Prime Number Sequence Predictor

This test case is similar to the symbolic regression test case, except here the curve to be fitted is the Fibonacci or prime number sequence. Determining a prime number by its index alone is one of the great problems surrounding mathematics. Prime numbers find application in cryptography algorithms, hash tables and pseudorandom number generators.

The task of predicting prime numbers using Cartesian Genetic Programming has already been attempted by James Walker and Julian Miller [44]. In this paper two tasks were undertaken: the prediction of the prime numbers un-consecutively and consecutively. Each of these two tasks was attempted using two different methods. The first method was symbolic regression using the functions: addition, subtraction, multiplication, protected division, and protected modulus. The second method was to evolve a digital circuit using multi-chromosome Cartesian Genetic Programming [45]; where the genotype is comprised of multiple chromosomes each responsible for a single output. The outputs of this digital circuit were used as the coefficients of a binary representation of the predicted prime number.

Cartesian Genetic Programming has also been applied to the task of predicting the Fibonacci Series undertaken by Simon Harding, Julian Miller and Wolfgang Banzhaf [28]. This paper used an adapted form of Cartesian Genetic Programming called Self Modifying Cartesian Genetic Programming; see the section on Self-Modifying Cartesian Genetic Programming for further details. Although this process could also be achieved using symbolic regression.

For this project the test case would be achieved using straight forward symbolic regression and not by any of the other methods described. It would therefore be a similar test case to Symbolic Regression (Curve Fitting) but to a more real world (and interesting) application.

7.7 Travelling Salesman

The "Travelling Salesman" is a famous NP-hard computer science optimization problem developed around the 1950's; although its exact history appears to be unknown. Just for fun, see Figure 11 for its appearance in the Webcomic XKCD [46].

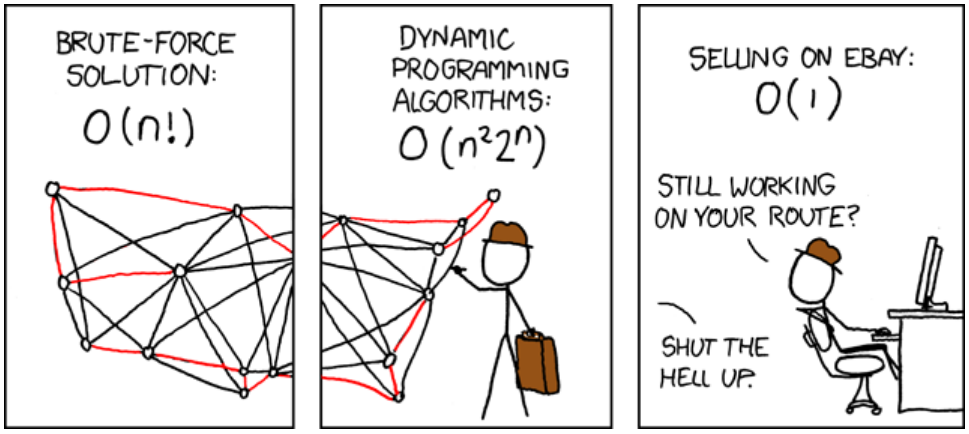

Figure 11 A Travelling Salesman Solution as depicted by the XKCD Webcomic

It is possible to apply Cartesian Genetic Programming to the "Travelling Salesman Problem" by setting the number of outputs to equal the number of cities. These outputs are then mapped to an ordered list of the cities. The Cartesian Genetic Program outputs are then sorted into numerical order, thus creating a new order of cities[9]. This process is shown in Figure 12.

The inputs to the Cartesian Genetic Program are a range of arbitrary constants and the node functions would be a set of mathematical operands. In Figure 12 the outputs are limited to be between zero and one; but this does not have to be the case.

The "Travelling Salesman" problems would be taken from [47], a group in association with the Heidelberg University who have a substantial archive of "Travelling Salesman" problems; along with their current best known solutions.

[9] This method was originally derived by Julian F Miller during a taught 4th year lecture course at the University of York in 2011. The concept was then investigated by a fellow student and friend Matthew Glenister for his assessment of this module.

CGP OUTPUTS	City Number
0.05	1
0.98	2
0.48	3
0.87	4
0.21	5
0.55	6
0.64	7
0.29	8
0.10	9
0.74	10

CGP OUTPUTS	City Number
0.05	1
0.10	9
0.21	5
0.29	8
0.48	3
0.55	6
0.64	7
0.74	10
0.87	4
0.98	2

Figure 12 Depiction of the ordering process used for generating "Travelling Salesman" permutations

7.8 Even Parity

This test case is similar to the test case described in the Synthesis of Boolean logic section, except here the truth table to be realised is now that of the parity bit needed to ensure even parity. As described in the Embedded Cartesian Genetic Programming section, calculating the parity for a bit string is quite a difficult task when only using the logic gates: AND, OR, NAND and NOR. Therefore to provide a higher level of difficulty only these gates are provided for this test case.

An advantage of this test case is the size of the bit string can be increased and the time taken by the Cartesian Genetic Program to converge on a solution recorded for each length. A reasonable comparison can then be made between normal Cartesian Genetic Programming and the new crossover technique over a range of problem complexities.

7.9 Artificial Ant

The Artificial Ant Problem [48] was developed by D Jefferson et al in 1991 and used by Koza in his book [13]. The problem to be solved is to navigate an "ant" around its "world" so as to gather the maximum amount of "food" in a limited number of movements. The "world" in this case is a two-dimensional toroidal grid with some locations containing "food"; see Figure 13 taken from [48].

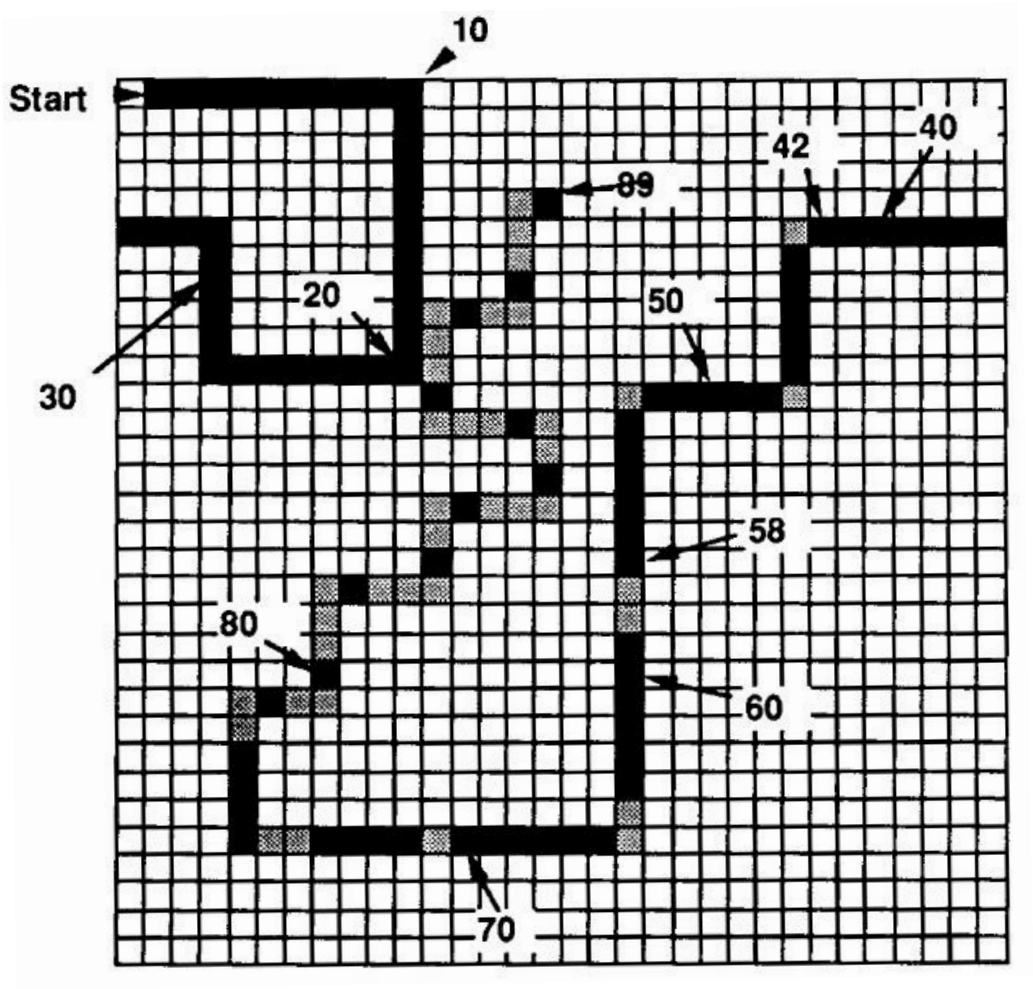

Figure 13 A possible layout of a "World" used for the "Artificial Ant Problem" with the most preferred route shown

The "ant" operates in a sense-and-act loop, where its only sense is to "see" the state of the square ahead, see Figure 14 also taken from [48], and its possible actions are: move forward one step, turn right (without moving), turn left (without moving) or do nothing. As can be seen in Figure 13, the optimum path to be followed increases in difficulty; the points that are awarded at various positions along this route are also shown.

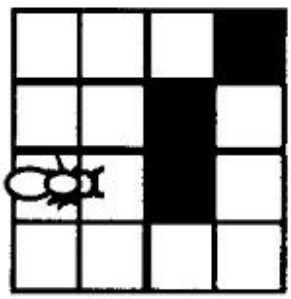

Figure 14 Depiction of the "ants" ability to see one square ahead

D Jefferson describes two different methods of solving the "Artificial Ant Problem"; the first using finite state machines and the second using artificial neural networks. It is the finite state machine representation which would be undertaken by this test case. This approach would be achieved by defining seven outputs for the Cartesian Genetic Program. The first two of these bits would describe the next action of the "ant" e.g. 10 could decode to "turn right". The remaining five outputs would be fed back as inputs to the Cartesian Genetic Program, these would represent the next state of the finite state machine; see Figure 15. This approach is thought to produce a representation which can be converted into a finite state machine.

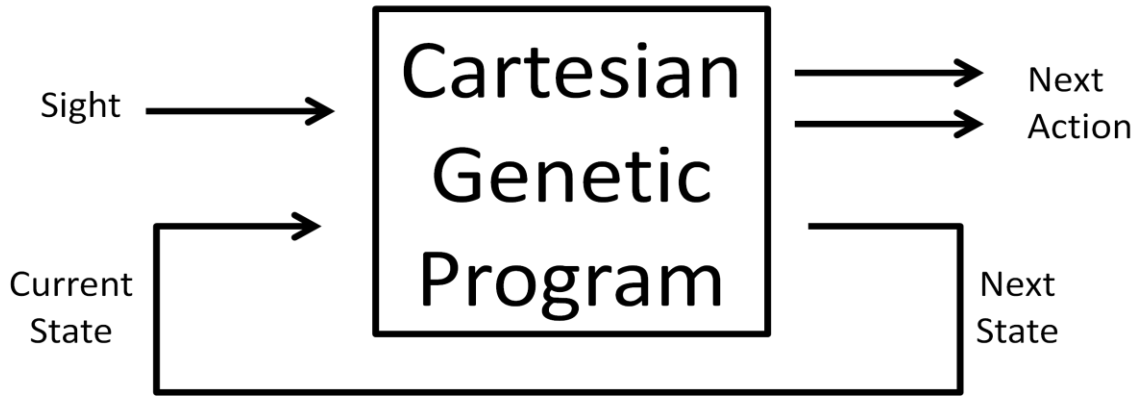

Figure 15 Possible Architecture to implement a Finite State Machine

The implementation of sequential circuits using Cartesian Genetic Programming has been indirectly studied before [49] by J Walker et al; whilst automating code generation for MOVE processors. Julian Miller also described using the artificial ant test case in the original formal paper on Cartesian Genetic Programming [17]; this paper however did not decode the results into a finite state machine. It therefore appears that the evolution of finite state machines using Cartesian Genetic Programming has been relatively uninvestigated. The creation and implementation of finite state machines using Evolutionary Computation has however been undertaken using other strategies. B Ali et al gives a detailed example of using Genetic Algorithms to design sequential logic circuits [50]. Interestingly, Simon Lucas describes the use of Cartesian Genetic Programming for evolving Finite State Transducers (a close relative of Finite State Machines) as further work in a published paper [51].

7.10 Game of life

As a final (and probably far too ambitious) test case, the evolution of starting arrangements to be used in Conway's "Game of Life" [52] would be investigated. The "Game of Life" is one of the simplest forms of cellular automaton; a form of computing where the next state of each component (or cell) is dependent on its current state and the state of neighbouring cells. Using simple rules, cellular automaton has been shown to create complex structures and has been an area of interest for a number of decades. Figure 16 shows a standard depiction of the "Game of Life" taken from a web based Java Script implementation of the "Game of Life" [53].

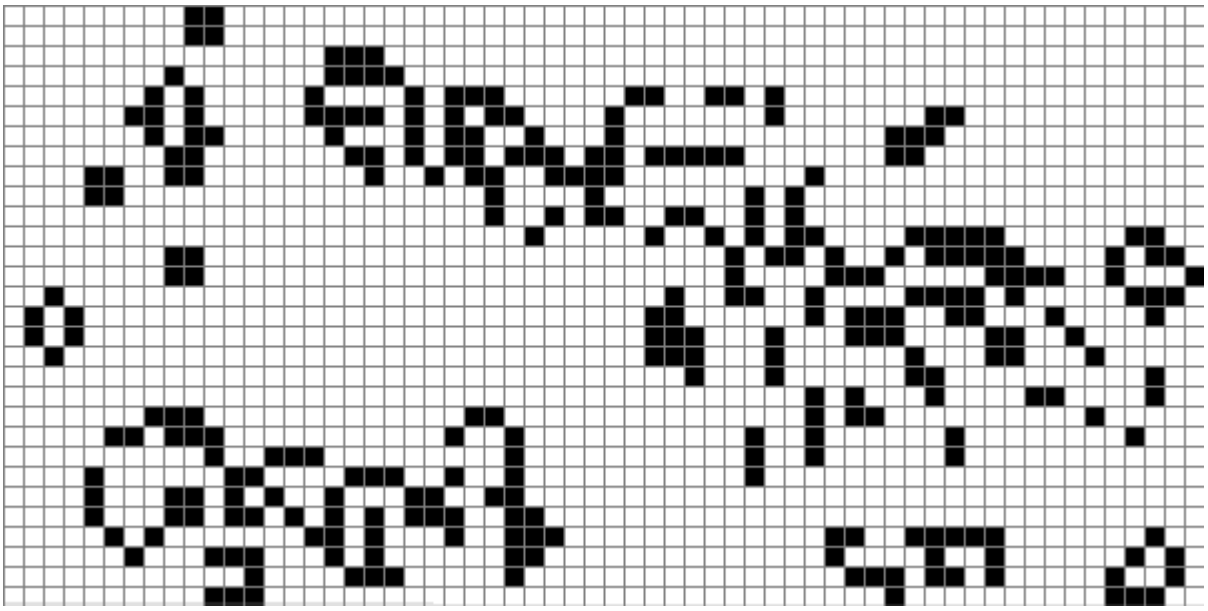

Figure 16 Depiction of Conway's Game of Life

A paper written by D Kazakov et al [54] used Genetic Algorithms to try and identify sets of rules to govern the "game of life". The aim was to discover rules which were most likely to support "interesting" life. The work assumed a relationship between the entropy of the system and the appearance of "interesting" life. High entropy systems were considered too chaotic to support "interesting" life and low entropy systems were also thought to contain nothing of interest. It was therefore assumed that a level of entropy somewhere between the two extremes would be an indication of the most "interesting" life. The investigation calculated local entropy values around the grid, identifying areas of high entropy in an otherwise low entropy "world".

Another application of Evolutionary Computation applied to the “game of life” is to evolve initial arrangements which produce “interesting” behaviour. These typically follow the original rules described by Conway:

1. Survivals. Each live cell with two or three neighbouring cells survives for the next generation.
2. Deaths. Each live cell with four or more neighbours dies (is removed) from overcrowding. Every cell with one or fewer neighbours dies from isolation.
3. Births. Each dead or empty cell adjacent to exactly three neighbours –no more, no less–comes to life.

Two examples of this type of work are: E Sapin et al [55] who successfully evolved configurations which created “Glider Guns” (a structure which creates further structures) and Hector Alfaro et al [56] who successfully re-discovered many previously known structures.

This test case would follow the same approach as D Kazakov et al [54], the difference would be however, to try and identify “interesting” structures using the standard rules. This would be an ambitious test case and is likely not to be completed as a result. It would however make for a interesting investigation as it combines two large fields in artificial intelligence: evolutionary Computation and Cellular Automata.

8 Project Timeline

This Chapter describes the timeline which was originally constructed to guide the project. The timeline described was not "set in stone" as it was considered more than likely that different stages of the project may take more or less time to complete than anticipated. It was however, intended to ensure the project was completed on time and give a sense of how much work was needed to be completed. The weekly timeline to be followed is shown in Figure 17; each stage of which is now discussed in further detail.

8.1 Research and Reading

This stage of the project was intended for research and reading of related works. Only two weeks were allocated for this stage of the project as a general background had already been achieved during previous studies. This stage was split into four subsections; this was intended to give an indication of the type of topics researched, although other areas were also to be investigated.

8.2 Initial Report

The writing of the initial report was planned to begin as early as possible; this was so an initial structure could be formed which would then guide the research. It was intended that the initial report would constitute much of the same information as used in the first few chapters of the final report. The initial report was intended to be continuously developed until the deadline is reached; shown in red on the timeline.

8.3 General Code Design & Production

The general code was defined as the code used throughout all of the test cases; a generalised Cartesian Genetic Program. The code was to be designed so that it could easily be adapted for each test case and for the new crossover technique. The design was intended to start by defining a detailed specification; which would then be used to form a class diagram. Genetic Programs are considered fairly simple to create; therefore three weeks were allotted for this stage.

8.4 Test Case 1, 2 and 3

It was thought that as the project progressed, the author's level of competency at implementing new test cases would improve. It was also thought, that as the project progressed more challenging test cases would be selected. For this reason the same time period was allocated for each test case. Each test case was to be: designed, coded and tested as a regular Cartesian Genetic Program, before being implemented using the new crossover format and re-tested. The evolutionary parameters for both methods would then be optimised for the particular test case followed by multiple runs of solving the given test case. Each test case section ends by writing up the observed results; this ensures that the creation of the final report was completed throughout the project and not left until the end.

If it was found that the project was too ambitious and there was not enough time available to complete the proposed work, the number of test cases would be reduced. All of the aims of the project could still be (in part) achieved when using fewer test cases and would be far more conclusive than attempting many test cases to a low standard. If however it was found that there was ample time to complete the proposed workload, further test cases could then be completed.

8.5 Publish Results

It was a personal aim to attempt to publish the results of this project; assuming the results were worthy of publishing. This would require the writing and submission of an academic paper to a relevant journal and would therefore take time to complete. Two weeks were allocated to the research and production of this paper, but as this stage was not essential it could be removed if the project began to over-run.

8.6 Finish Report

This stage comprised the completing and submission of the final report. It was intended that the final report would be completed throughout the project e.g. the design and coding chapters to be completed during the design and coding stage and each test case written upon completion. This time was allocated to writing the concluding chapters of the report and completing the final edit before submission.

8.7 Presentation

Two weeks were allocated for the production of the final presentation. This also included time for the preparation of the viva.

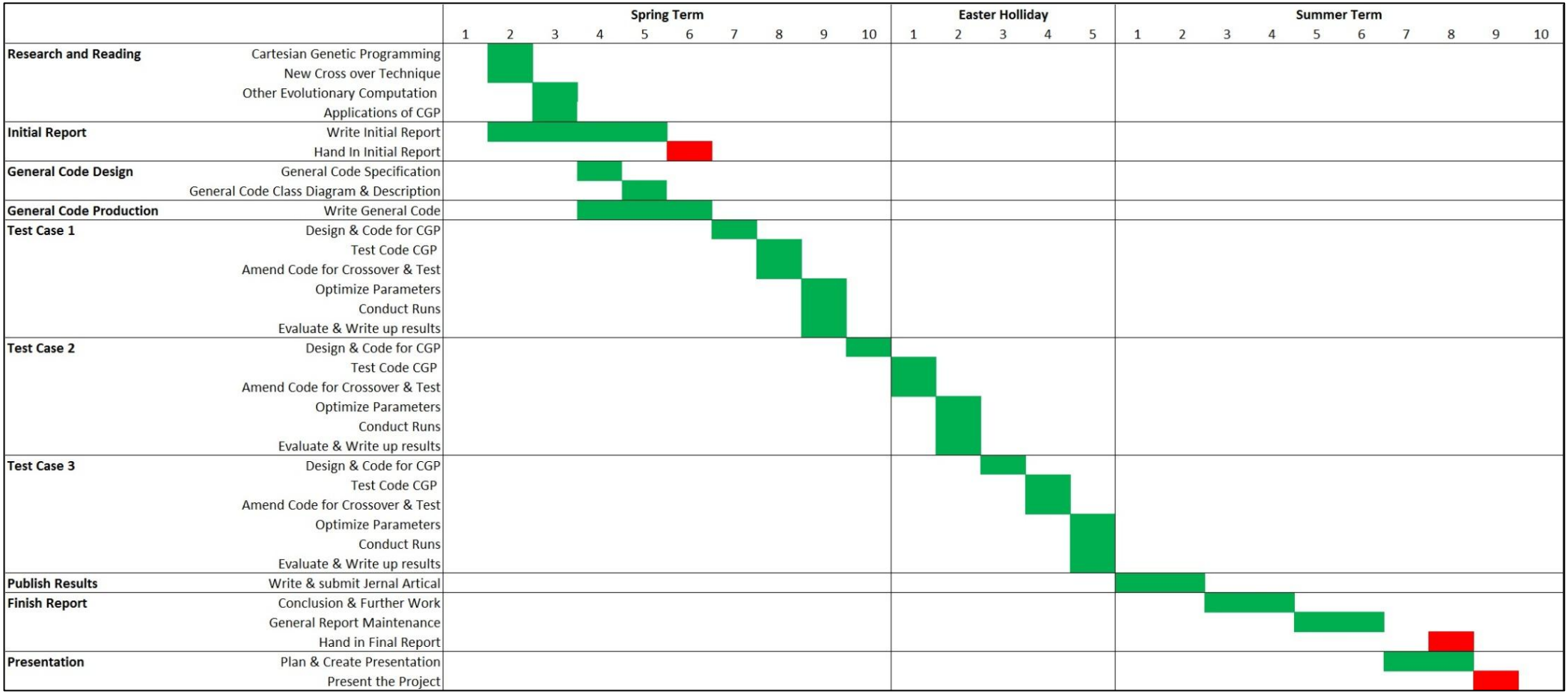

Figure 17 Project Timeline

9 Implementing the New Crossover Technique

This chapter explains how the new crossover technique is implemented throughout this project. The crossover method used is BLX-0 (BLX-α implemented with α set to zero) as used by Janet Clegg [1] and described in the Crossover Techniques Chapter.

The implementation of BLX-0 crossover requires the use of a selection scheme to choose which parents are to be used to generate the child chromosomes. This is unlike most implementations of Cartesian Genetic Programs, which use elitism and asexual reproduction to generate the next population. The selection scheme used by Janet Clegg, and therefore by the author for this project, is tournament selection. Tournament selection is where a predefined number of the current population are selected to enter a “knock-out” tournament, where the winners are selected as the parents. In the authors (and Janet Clegg's) implementation, a simple tournament is used. All of the chromosomes are entered into the same round of the tournament and the two chromosomes with the best fitnesses are the winners. Interestingly, there currently appears to be no literature describing if tournament selection offers an advantage or disadvantage to Cartesian Genetic Programs; although via email Julian Miller stated that in his experience he has never found tournament selection to offer an advantage.

BLX-0 crossover also requires that the chromosomes are represented in a floating point form; as described in the Crossover Techniques Chapter.

The implementation used is shown via a flow chart in Figure 18. The sequence of events shown is undertaken each generation to create the next population from the current population. The process is also described here:

1. Start with the current population
2. Calculate a fitness for each member of the population
3. Promote the best μ member of the current population directly to the next population
4. Select a predefined number of the current population to be entered into a tournament
5. Select the best two members of the tournament to be the parents
6. Pick a random floating point number between zero and one
7. If (random number < crossover percentage) use crossover to create two children and add them to the next population
8. If (random number >= crossover percentage) Add the two parents to the next population
9. If(next population < population size) repeat from 4
10. Mutate all of the nest population except the promoted elite

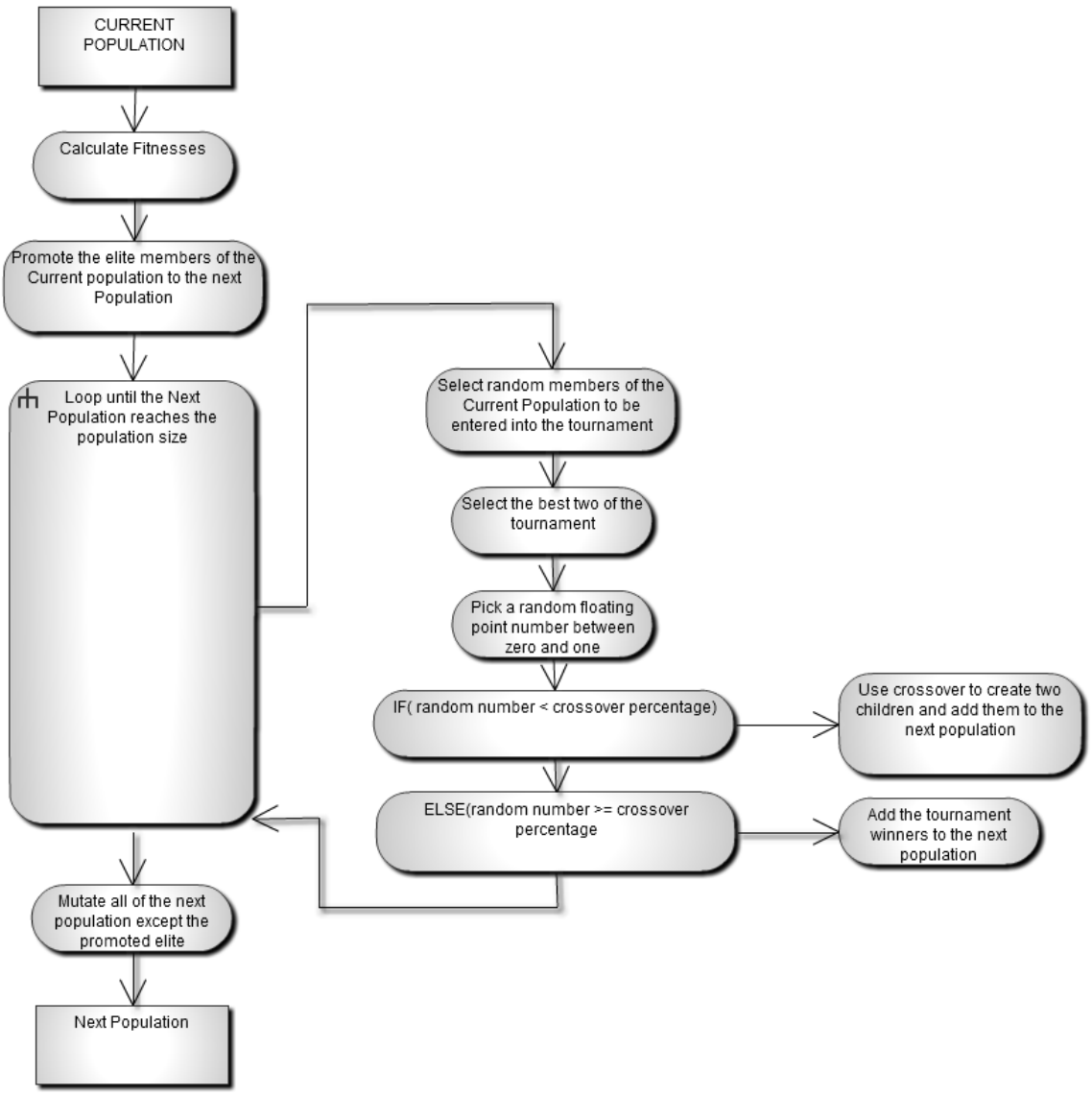

Figure 18 Flow chart showing the operation of BLX-0 crossover technique – as used for this project

For later reference, the process shown in Figure 18 is implemented by the Reproduction Class in the author's code.

It is seen here, that the crossover percentage variable controls how often crossover is used to generate the children. In situations where the crossover is not used, the children are clones of their parents. If crossover is not used at all (a more traditional implementation of Cartesian Genetic Programming) then the children of the next generation are created by asexual reproduction of the elite members whilst employing the mutation operator.

10 Specification

This chapter describes the specification for the Cartesian Genetic Program to be used throughout this project. Each of the specification items is split into mandatory or optional and then discussed in further detail.

10.1 Mandatory Specification

These are specifications which are required by the project; they were all to be completed.

1. The Cartesian Genetic Program should be easily adapted to different test scenarios with no (or little) change to the majority of the program.
2. All parameter variables which govern the evolutionary process are to be stored in one accessible location and easily edited.
3. The parameters are to be parsed within the code to check for errors, unrealistic values and discontinuities.
4. The parameters must include:
    a. Number of runs[10] (integer value)
    b. Number of generations (integer value)
    c. Mu (integer value)
    d. Lambda (integer value)
    e. Floating point representation (Boolean flag)
    f. Crossover (Boolean flag)
    g. Percent crossover (percentage)
    h. Percent mutation (percentage)
    i. Chromosome structure (number of inputs/function nodes/outputs)
5. The program must terminate when defined termination conditions are met. These termination conditions must include:
    a. Maximum number of generations has been reached
    b. The process has reached a solution which is considered acceptable
6. All experiments must be repeatable

[10] The number of runs parameter refers to the number of times the experiment is repeated.

7. Details of each experiment are to be automatically generated and stored for later inspection. These details must include:
    a. A full breakdown of the parameters used
    b. The final state of each run:
        i. The final best fitness
        ii. The generation on which this was achieved
        iii. The standard deviation of the chromosome fitnesses on the final run
        iv. The number of active nodes in the best chromosome
    c. A full breakdown of each run:
        i. Best fitness at each generation
        ii. Standard deviation of the fitnesses at each generation
        iii. Number of active nodes in the best chromosome at each generation
        iv. Average number of active nodes at each generation
    d. The best fitness at each generation averaged across all runs
    e. The structure of the best chromosome found by each run

10.2 Optional Specification

These were specification which would be implemented if time were available.

8. The ability to easily change the crossover being employed.
    a. Such as being able to change the alpha parameter in BLX-α crossover
9. The ability to use variable crossover (Such as used by Janet Clegg)
10. The ability to include extra termination conditions
    a. Such as if the best solution has not changed for a given number of generations

10.3 Specification Breakdown

This section explains and justifies each item given in the specification.

Specification 1 is to ensure that unnecessary time is not spent re-writing large sections of the Cartesian Genetic Program for each new test case. It should be possible to create a layer of abstraction between the fitness function and the Cartesian Genetic Program; so new test cases can be easily implemented.

Specification 2 is to ensure changing the parameters of the Cartesian Genetic Program is as simple as possible. Having all the parameters defined in a single place removes the task of locating every instance of their use within the code. This practice is common for medium scale programs and is particularly useful in this case as the parameters are to be changed regularly.

Specification 3 is to ensure that the parameters chosen for a particular experiment are valid both individually and with respect to the other parameters. This ensures that time is not wasted on invalid experiments which never correctly operate and reduces the debugging process.

Specification 4 lists the minimum number of parameters which are required for the proposed experiments to be undertaken.

Specification 5 describes constraints that should always be placed upon Evolutionary Computation; ensuring the search process does not continue indefinitely. The given termination conditions (a and b) are standard termination conditions and are therefore used for this project.

Specification 6 describes a common requirement of all scientific experiments. It could be important that a previous experiment is re-investigated and as a result this feature must be included.

Specification 7 is to ensure that the results of each execution of the program are stored in a human readable form after the program has terminated. This enables results to be stored and later compared to those of other experiments.

Specification 8 is included as it may be interesting to evaluate how effectively Cartesian Genetic Programming operates with other forms of crossover. Therefore the code should be structured in such a way that a modification to another crossover method is easily implemented.

Specification 9 is included as variable crossover is a slight modification on the BLX-0 crossover; as used by Janet Clegg and described in her paper[1]. It is possible that this

crossover method would be investigated at a later stage; as a result the design should allow this modification to be easily implemented.

Specification 10 is included as there may have been a requirement for different termination conditions to be implemented for different test cases. This modification should be easily implemented.

11 Cartesian Genetic Program Production

This Chapter discusses the design process used to plan, construct and test the code used throughout this project. This chapter begins by outlining an Initial Design which was created in order for coding to begin as early as possible. The second section, Meeting the Specification, describes how the initial design chosen meets each criteria within the specification given previously. The Code Production section describes the order in which the Cartesian Genetic Program is coded, tested, pieced together with the other sections of code and re-tested. The chapter closes with a section briefly covering the Final Design.

11.1 Initial Design

It was decided that many aspects of the design would only be appreciated once the coding stage had begun; as a result, the coding was started as early as possible. Of course one cannot begin coding blindly, and so this section describes the initial quick design which was completed so coding could begin.

The Simplest way to describe the design is to first show the chosen internal structure of the code via a Class Diagram. For those not familiar with Object Orientated Programming, Classes are similar to structures in the procedural programming language C, only in Object Orientated Programs everything must be described as a Class. See Figure 19 for the initial Class Diagram of the author's Cartesian Genetic Program. The arrows in this simplified Class Diagram show which Classes are dependent on other Classes e.g. the Fitness Class shown is using the functionality provided by the FunctionSet Class.

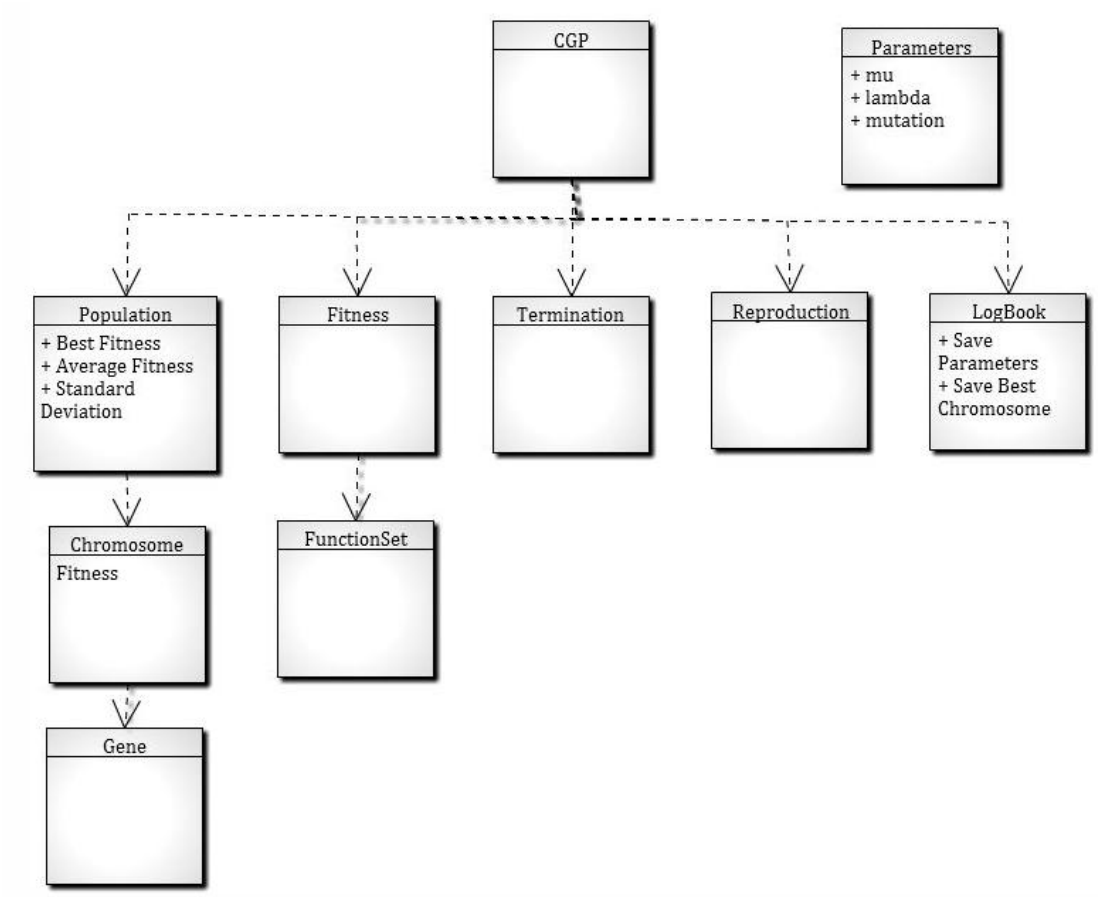

Figure 19 Initial Class Diagram of the author's Cartesian Genetic Program

As with all non-trivial programs, this Cartesian Genetic Program is implemented using sub modules, or sub Classes, to break up the design and implementation task; as seen in Figure 19. These sub Classes are now described in detail.

11.1.1 CGP

The CGP[11] class is treated as the main function and is used to implement the high level functionality of the Cartesian Genetic Program. This class employs the functionality of other sub classes to carry out its operations. This involves producing an initial population of chromosomes, assessing their fitness, checking if the termination conditions have been reached and also generating the next population. It is also responsible for managing large numbers of runs so statistics can be generated.

11.1.2 Parameters

The Parameters Class is used to store all of the parameters which are to control the Cartesian Genetic Program. All of these parameters are stored in a global manor, so they

[11] CGP is the acronym for Cartesian Genetic Program or Cartesian Genetic Programming.

can be accessed from anywhere within the code. The Parameters Class is also responsible for parsing the parameters, ensuring they are valid, and producing useful error messages if they are not.

11.1.3 Population

The Population Class provides access to, and stores, all of the chromosomes within the current population. It is also used to access statistics surrounding the overall population, such as the best or average fitness and the standard deviation of these fitnesses. The Population Class is also used to create the initial population of random chromosomes.

11.1.4 Chromosome

The Chromosome Class is used to provide access to all of the genes which make up each individual chromosome; these chromosomes then comprise the population. It also provides access to specific statistics for each chromosome such as its fitness and the number of active nodes. The Chromosome Class is also used to create the initial structure of the genes within each chromosome and ensure they represent valid representations.

11.1.5 Gene

The Gene Class is used to provide access to the parameters which make up each gene of a given chromosome. These parameters include: the gene type (function or output), input locations, output locations and the nodes functionality. It is also responsible for creating valid random values for each parameter when a new chromosome is generated.

11.1.6 Fitness

The Fitness Class is responsible for taking a given population of chromosomes and calculating the fitness to be assigned to each. This requires the use of the FunctionSet Class to implement the functionality of the function genes. The Fitness Class is also used for calculating and assigning the number of active nodes to each chromosome. Knowing which nodes are active helps reduce the computational time required to analyse each chromosome; as only active genes need to be evaluated. This Class is one of two which had to be altered when applying the Cartesian Genetic Program to different test cases.

11.1.7 FunctionSet

The FunctionSet Class is responsible for implementing the functionality of the function genes within each chromosome. It is also responsible for storing the range of functions which can be selected by the function genes. This is the second of the two classes which have to be altered when applying the Cartesian Genetic Program to different test cases.

11.1.8 Termination

The Termination Class is responsible for checking if any of the termination conditions have been reached. This is achieved by the Termination Class having access to the current population and generation. If any of the termination conditions are reached the current run of the Cartesian Genetic Program is terminated. If the current run is terminated, the Termination class is also responsible for calling the Log Book Class to create a record of each run.

11.1.9 Reproduction

The Reproduction Class is used to generate the next population from the current. This is achieved using a range of methods; depending upon which strategy is currently under investigation e.g. crossover or no crossover. Its internal operations also vary depending upon the evolutionary parameters described in the Parameters Class, for example: mutation percentage, crossover percentage, floating point representation.

11.1.10 LogBook

The LogBook Class is used to save all of the results in human readable .txt documents. The detail and depth of the results saved depends upon parameters set in the Parameters Class.

11.2 Meeting the Specification

This section describes how the initial design chosen meets each item described in the Specification chapter.

Specification 1 is achieved by ensuring that the code is structured such that very few areas in the code need to be altered in order to implement new test cases. The specificFitness method within the Fitness Class and the FunctionSet Class are the only areas within the code which need to be altered when implement different test cases. Although implementing

different test cases still may not be trivial for some of the given examples, it requires no major change to the Cartesian Genetic Program.

Specification 2 is achieved by storing all of the evolutionary parameters in one accessible Parameters Class, which can then be edited to change their values. It was considered that it may have been more suitable to store the parameters in a .txt document and read them in to the program; but as only the author uses the code this extra user-end simplicity seemed unnecessary.

Specification 3 is also achieved within the Parameters Class by employing a parser which checks the given parameter values before starting the Cartesian Genetic Programming section of the code.

Specification 4 is achieved by storing all of the given parameters as editable variables within the Parameters Class.

Specification 5 is achieved by employing a dedicated Termination Class, called each generation, to inspect if any of the termination conditions have been reached. If one of the given conditions has been reached, then the class terminates the current run.

Specification 6 is achieved by ensuring that the pseudo random number generator (used within the code) can be given a “seed” which ensures that it produces the same random numbers each time. This ensures that if the same experiment is repeated, all of the random variables are the same; hence the same results will be generated.

Specification 7 is achieved by employing a dedicated LogBook Class which contains many methods responsible for storing all of the given details of each experiment in a human readable "Log Book".

Specification 8 is achieved by designing the code such that changing the type of crossover been employed is a simple process. Other types of crossover were not implemented however, as time was not available to undertake further investigations into their effect.

Specification 9 follows the same process as Specification 8, the code was designed to make this a simple process, but not implemented until needed.

Specification 10 is similar again to Specification 8 and 9, except in this case it was achieved by adding and/or removing clauses from the Termination class.

11.3 Code Production

The code production strategy followed was to build and test each class individually where possible. If a class relied on other classes for its operation, then all of its dependencies were built and tested first; allowing then for the production and testing of the given class. To ensure the merging of all the classes was as simple as possible, a sideways approach was taken; where sections are brought together which can operate in isolation, so simplified testing can be undertaken. These sections are then brought together with other sections and continually tested. This is repeated until the entire program is constructed.

The production sequence which was followed is given in Table 1, this sequence was designed to reduce the number of test stubs[12] required to test each class; which reduced and simplified the testing stages. It can also be seen in Table 1 that the Cartesian Genetic Program was first written in the more standard form; which uses no crossover and an integer representation of the chromosomes. This was to simplify the coding and testing stage; the presence of crossover (which requires a floating point chromosome representation) was then later implemented.

[12] Test subs are additional code which is written purely for the testing of other sections of code.

Table 1 Code Production Breakdown

Production Stage	Activity	Associated Classes[13]	Details
1a	Build	Parameters	Additional parameters were added to the Parameters class when needed by other classes.
1b	Test	Parameters	As the Parameters class is so heavily used it was intrinsically test throughout.
2a	Build	Gene	Originally implemented to only work with an integer representation.
2b	Test	Gene	-
2c	Build	Chromosome	-
2d	Test	Chromosome Gene	-
2e	Build	Population	-
2f	Test	Population Chromosome Gene	-
3a	Build	FunctionSet	A simple arithmetic function set was used during production which contained '+', '-' , '*' and '/' operations.
3b	Test	FunctionSet	-
3c	Build	Fitness	This was implemented with a simple test case with the objective to produce a small integer number i.e. 12. The fitness value was then the absolute of the difference between the produced value and the target value.

[13] Almost all of the classes rely on the Parameters Class and so it is not included each time as an associated class.

3d	Test	Fitness FunctionSet Population[14]	-
4a	Build	Reproduction	Originally implemented to generate the next population through mutation only (no crossover).
4b	Test	Reproduction Fitness FunctionSet Population	-
5a	Build	LogBook	-
5b	Test	Logbook Population	-
6a	Build	Termination	
6b	Test	Termination Population LogBook	
7a	Build	CGP	Eventually treated as the main entry point to the whole program.
7b	Test	All Classes	
8a	Adapt	All Classes	Implement the possibility for floating point representation and crossover.
8b	Test	Population	Now with floating point representation and crossover.
8c	Test	Fitness FunctionSet Population	Now with floating point representation and crossover.
8d	Test	All Classes	Now with floating point representation and crossover

[14] And all of the dependencies; Chromosome and Gene classes.

11.3.1 Evaluation of the Coding

During the initial coding phase the author was far too hasty with the coding and not enough time was spent on maintaining readable structured code. As a direct result, furthering the development of the code became increasingly difficult. This led to the author having to spend multiple days on "neatening" the code, including the addition of meaningful comments and structuring the code in a more standard format. Once this had been completed, navigating and extending the code became much simpler. It is thought however, that the initial "messy" stage of getting ideas down in code may be very beneficial to the design and learning process.

The difficulties encountered during the coding stage were both language specific and issues with the author's logic. This included confusion between the assignment of the address of an object (variable type) and the value(s) of that object itself. Overall however the coding was undertaken quickly and effectively with no major bugs or design errors.

11.4 Final Design

The final design followed the same structure outlined in the Initial Design section. It was the original intention to provide a detailed breakdown of how each Class operated and interfaced with the other Classes. It was decided however, that this process would be hugely time consuming, significantly lengthen the final report and would not contribute to the overall project. For these reasons a detailed breakdown has not been provided. A full Class diagram showing the structure of the final code is given in Figure 20 (and in Appendix D). This combined with the commented code also provided in Appendix D, should enable the reader to read and navigate the code with relative ease if this is desired.

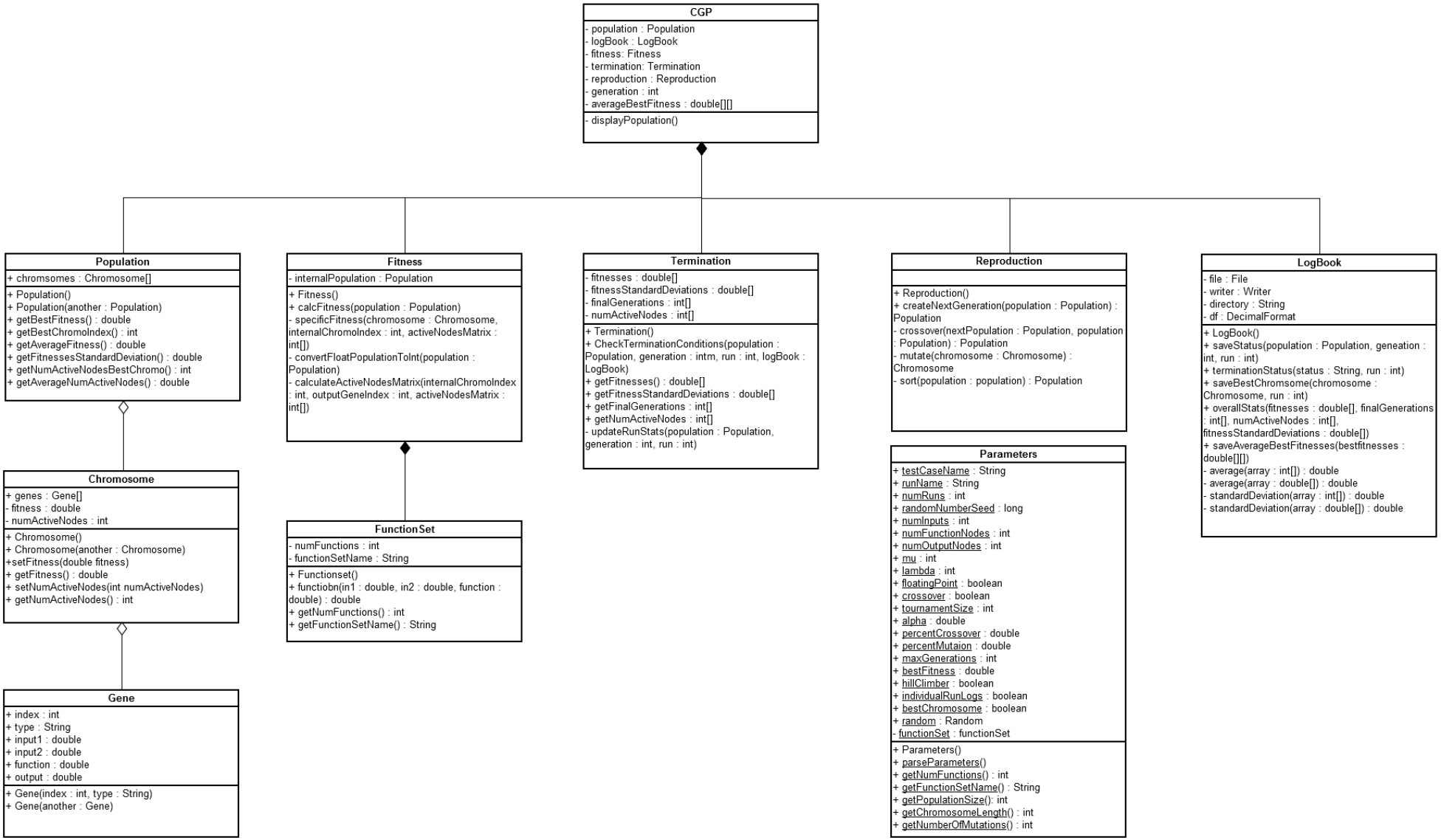

Figure 20 Final Class Diagram of the author's Cartesian Genetic Program

12 Testing

During the testing stages, the Cartesian Genetic Program was written to generate an integer number (twelve), from the inputs zero and one. The operations available to the function nodes were: addition, subtraction, multiplication and protected division. This ensured that sections of the code were operational as early as possible and aided the debugging process.

The majority of the testing was undertaken using print statements to the consol describing the current state of the program and variable values. As the code became more complete, analysis was then achieved by inspecting the generated "log book" files. It is understood that more rigorous formal test strategies are available; JUnit testing is a common technique used for java projects [57]. It was thought however that these formal methods of testing would take far longer to implement (and presumably test themselves) than the actual code itself. It was also thought that an adequately rigorous testing process could be achieved through simple printouts/inspection of results and that this would help simplify the testing procedures. The individual testing strategies used for each class are now described in detail.

12.1 Parameters

As the majority of the Parameters class role is to simply store values, testing this class was very simple. Printouts of all of the variables stored by the Parameters class were generated and compared to the values which were expected to be found. The testing of the Parameters parser was slightly more complex. This was achieved by first entering a range of correct parameters and ensuring that the parser passed through the values. Values were then assigned to the parameters which were either not allowed or conflicted with each other e.g. a mu value less than one or trying to implement crossover without using the floating point chromosome representation. The parser was systematically checked to ensure it "flagged" nonsensical values and produced helpful error messages where appropriate. Another operation of the Parameters Class is to generate further values from the given parameters e.g. calculating the number of genes to mutate from the given number of function/output nodes and the mutation percentage. These types of operation were again ensured to be correct via printouts to the consol; for instance the number of genes to be

mutated can be displayed and compared to manually calculated versions to check for discontinuities.

12.2 Gene

When testing the Gene class, a range of values were assigned to the different gene parameters (input, function and output) and then printed out for observation. The Gene class is also responsible for generating its own random valid values for use by the initial chromosomes. These valid random values are determined by values set in the Parameters Class e.g. the number of functions available or whether the floating point representation is been employed. All these possible scenarios were again tested via printouts and by changing values in the Parameter class.

12.3 Chromosome

The same style of testing was also carried out for the Chromosome class as used by the Gene class. New random chromosomes were generated and then their gene sequence printed out for observation. The structure of the chromosome could then be confirmed to be correct e.g. ensuring each node only indexed previous nodes and checking that the correct number of each node type was present (function and output). The values of the genes within the chromosomes were then altered to ensure this functionality was operational; as required by the mutation operator. Finally other parameters associated with each chromosome (such as fitness and number of active nodes) were set and retrieved; although only dummy values were used.

12.4 Population

For the testing of the Population class, a simple test stub was made which prints each member of the population as a gene sequence, along with its corresponding fitness; see Figure 21. An initial population was then created and displayed in the console. It was again possible to check the structure of each chromosome and that the correct number of chromosomes were present in the population. Other methods (functions) associated with the Population class, such as generating the average fitness of the population, could then be tested. All of these methods were tested by assigning values to the members of the population and then observing the analysis completed by the code via printouts. These

values were then compared to the same calculation carried out by the author manually; to confirm the results.

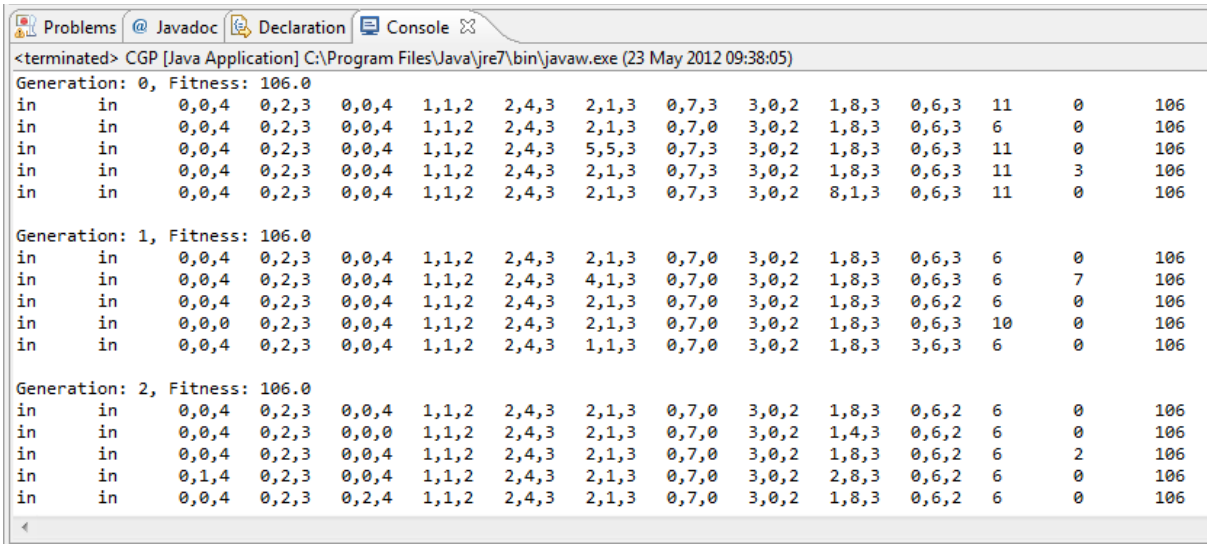

Figure 21 Printout of the chromosomes in each population followed by the corresponding fitness (106 in all cases)

12.5 FunctionSet

Once again the functionality of the FunctionSet class was tested via consol printouts. A range of input values were tested for each function type (addition etc) and the results compared to expected values. The FunctionSet class also stores parameters describing its operation, including function set identity and the number of functions it contained; these were also observed via printouts.

12.6 Fitness

The inner functionality of the fitness class changes depending upon the test case currently been evaluated; as a result it is necessary to re-test this class for each new application. There are however aspects of this class which remain constant independent of the test case being investigated. These include calculating the number of active nodes present in each chromosome. The testing of this functionality is achieved by printing out the calculated number of active nodes along with the corresponding chromosome. The number of active nodes can then be manually determined and compared to the calculated value. Another internal function used by the Fitness class, is to convert chromosomes represented using floating point numbers into an integer form. Again this operation can be confirmed to be correct by printing out chromosomes in their floating point and integer forms. The floating

point form of the chromosomes can then be manually converted into their integer forms and compared to those generated by the fitness class.

When each specific test case was evaluated, the operation of the Fitness class was confirmed using similar methods. The assigned fitness of each chromosome was displayed to the consol along with the corresponding chromosome. The chromosomes fitness was then manually calculated and compared to that generated by the fitness class.

12.7 Reproduction

The Reproduction class is one of the more complex classes to test, but once again is tested in the same format of printing various values to the consol and inspecting the results.

The first role of the Reproduction class is to determine which members of the population are to be automatically promoted to the next population as elite members. This is achieved by sorting the population into fitness order and then selecting the top μ elite members. The sorting of the population also depends upon whether high or low fitness values represent fitter or weaker members; this is determined by the hillClimber boolean flag set in the Parameters class (if true high values represent the fitter members and vise-versa). The testing was achieved by creating a dummy population and then passing it through the sorting process. The order of the chromosomes within the population can then be inspected along with their corresponding fitnesses; to ensure they are now in fitness order. This was undertaken for the hillClimber flag set to true and false.

The second role of the Reproduction class is to conduct crossover on the current population; to create the children which become the members of the next population (along with the elite members). Once again the testing was achieved via printouts to the consol. The current population was printed to the consol[15], with each chromosome on a new line as a series of numbers, as seen in Figure 21. Crossover was then conducted on this population and then the newly generated population printed in the same format. This was initially undertaken for small population sizes; ensuring there was less information to be viewed at once. The differences between the two populations could then be inspected to identify if the

[15] This is achieved via a method (function) called displayPopulation, which has been left in the code for future use.

crossover was operating correctly. Parameter class variables, including tournament size and crossover percentage, were also varied and their effects monitored.

The final role of the Reproduction class is to conduct the mutation operator; this is conducted regardless of whether crossover is being employed. When crossover is not being employed, the next generation is comprised of the elite members of the previous generation and mutated versions of those elite members. When crossover is being employed, the mutation operator is used on all members of the population generated by the crossover operator. The mutation method (function) was tested by printing to the consol, a series of chromosomes before and after mutation had been undertaken, and then inspecting the differences over a range of mutation percentages. This process was carried out for the integer and floating point form of the chromosomes.

The Reproduction class was then tested as a whole, by passing it a dummy population of chromosomes and printing the resulting new population. This was undertaken for a range of parameters including: population size, number of input/function/output nodes, integer/floating point representation, hillClimber true/false, mu values, lambda values, tournament size, crossover true/false, mutation percentage and crossover percentage.

12.8 LogBook

There are four separate "Log Book" files which can be generated: Average_Best_Fitnesses, Overall_Stats, individual run logs and the best chromosome. The Average_Best_Fitnesses stores the average fitness across all runs at each generation. The Overall_Stats saves all of the details of the experiments and then produces statistics including the average number of evaluations along with the corresponding standard deviation. The individual run logs stores statistics surrounding each run, along with the best fitness at each generation. Finally the best chromosome saves the structure of the best chromosome after each run, along with the number of active nodes and its assigned fitness. Figure 22 shows a selection of these generated "Log Book" files.

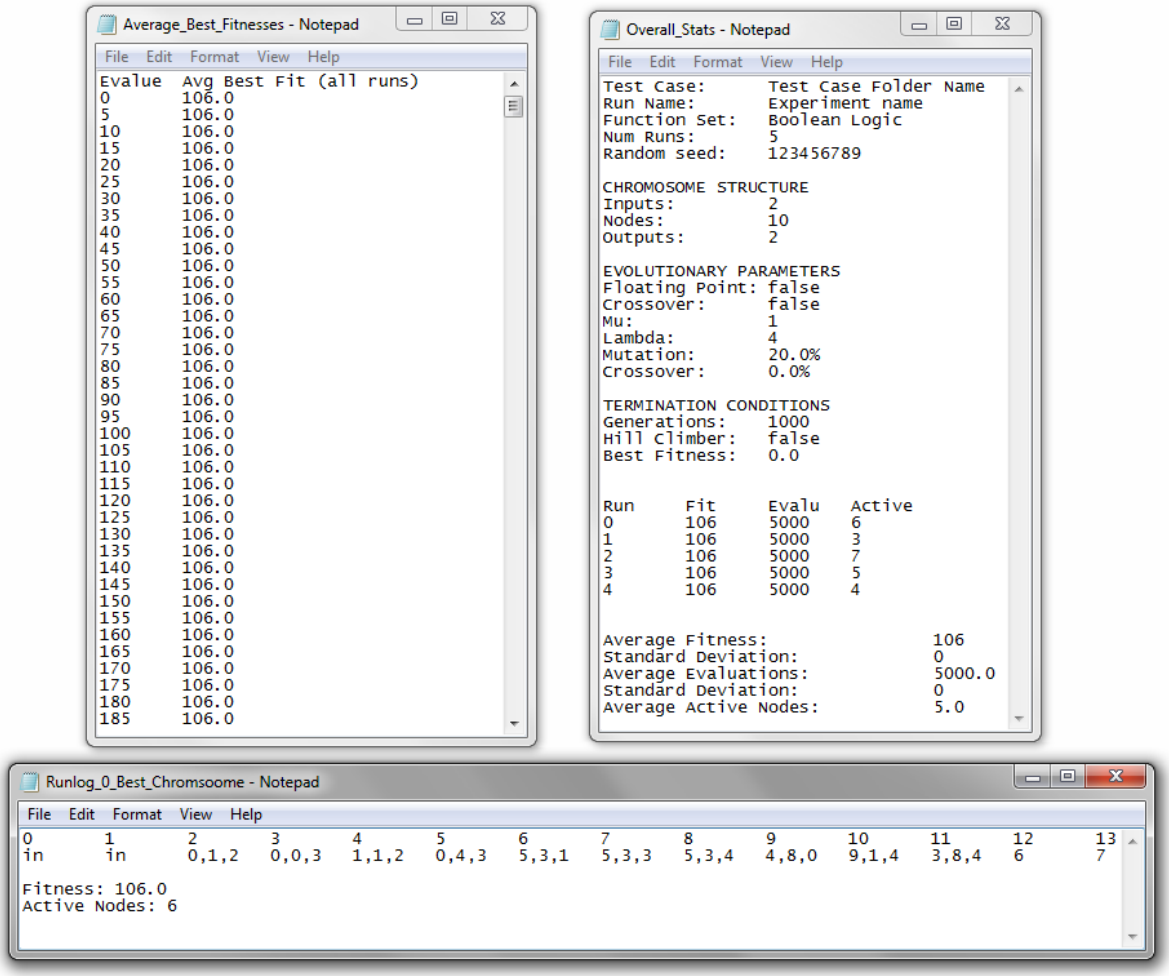

Figure 22 Sample selection of generated "Log Book" files

All of these log book files were tested by printing out all of the raw data been passed into the LogBook class and comparing the generated results with those calculated manually. This was only undertaken for: small population sizes, small numbers of generations and small numbers of runs; to keep the manual calculations reasonable.

The LogBook class also contains the functionality to calculate averages and standard deviations for given arrays. To test the operation of these methods (functions), a small test stub was made which contained various arrays and then printed the averages and the standard deviations to the consol. The same calculations could then be conducted manually to confirm correct operation.

12.9 Termination

Testing the main functionality of the Termination class was relatively simple as there were only two termination criteria: the maximum number of generations has been reached or an acceptable solution has been found. Printouts of the current generation were made to the consol and multiple tests carried out to ensure the maximum number of generations limit was never exceeded. The best fitness was also printed alongside the current generation, to test if the program terminated when an acceptable solution had been found.

The Termination class is also used to pass information to the LogBook class and so this functionality was tested in parallel with the LogBook class.

12.10 CGP

This was effectively a test of the entire Cartesian Genetic Program. As previously mentioned, the simple test of creating small integer values from the inputs zero and one was used as the test case during this testing stage. The testing process included changing different evolutionary parameters and observing printouts of each population and the generated log book files. Tests were also conducted with incompatible parameter values, to ensure the programs parser rejected them and gave useful error messages. The first major test case investigated, Repeating Janet Clegg's Experiments (given in the following chapter), was also partly undertaken as a testing procedure for the Cartesian Genetic Program.

12.11 Evaluation of Testing

The testing of the author’s code was a long and tedious process, the modular structure helped simplify the process, but the overall complexity and number of variables was quite challenging. In hindsight it is felt by the author, that a more regimented approach could have been followed; where a predetermined range of inputs and scenarios was decided and then each test case undertaken. This would have provided more substantial documentation for the testing stage and made the author feel more confident in the correct operation of the code in the early stages. Overall however, the author is confident that the code is operating correctly and feels that the testing strategy used was suitable for a project of this scale.

13 Repeating Janet Clegg's Experiments

The first experiment undertaken during this project was to reproduce the results presented by Janet Clegg in her original paper [1]. This is to both ensure the author's implementation of the Cartesian Genetic Program is functioning correctly and confirm the results obtained by Janet Clegg. This chapter also presents some additional experiments to confirm that the new crossover technique offers and advantage over regular Cartesian Genetic Programming without crossover.

13.1 The Experiments

Although several experiments are described in Janet Clegg's original paper, only two are repeated in this chapter. These experiments are the application of Cartesian Genetic Programming implemented with 0%, 25%, 50% and 75% crossover on two symbolic regression problems. The section entitled Symbolic Regression (Curve Fitting), in the Possible Test Cases chapter, describes Symbolic regression problems and the two functions used in Janet Clegg's paper.

Table 2 shows the evolutionary parameters used by Janet Clegg during her experiments. These parameters are unusual as implementations of Cartesian Genetic Programs commonly use small population sizes and much lower mutation rates. As a result a further experiment was investigated which aimed to optimise these parameters for the first symbolic regression problem; identifying if the parameters used by Janet Clegg were the most suitable. The optimisation process used to find suitable parameters is given in Appendix B.

Table 2 Parameters used by Janet Clegg in her paper

Parameter Name	Value
Population Size	50
Mu	2
Lambda	48
Function Nodes	10
Mutation Rate	20%
Max Generations	1000
Tournament Size[16]	4
Runs	1000

When implementing BLX-0 crossover (as used by Janet Clegg), it is a requirement that a selection scheme is employed during the evolutionary process; to be able to select the parents. Previous implementations of Cartesian Genetic Programs do not however use any selection scheme. Another further experiment is therefore an investigation into how the presents of a selection scheme affects the search process. This experiment is a comparison between Cartesian Genetic Programming without crossover and using no selection scheme, with Cartesian Genetic Programming implemented without crossover, but with a tournament selection scheme. In both cases the parameters were optimised to ensure a fair comparison. To keep the number of parameters which were optimised manageable, the tournament size was fixed at four when using the selection scheme. It is worth noting that implementing the new crossover technique with 0% crossover is in fact the same as implementing a Cartesian Genetic Program without crossover, but with a tournament selection scheme.

The use of a floating point representation for the chromosomes is also a requirement for the new crossover technique. A final experiment therefore investigates if this floating point representation changes the behaviour of the search process. This experiment is a comparison between a Cartesian Genetic Program (without a selection scheme or crossover), represented in the integer form, with the same Cartesian Genetic Program represented in the floating point form.

[16] This tournament size was not actually specified in Janet Clegg's paper and so four was used as a starting point. It was later realised during discussions with Janet Clegg that a tournament size of twenty was used.

In Janet Clegg's work each experiment was repeated one thousand times and then the best fitness at each generation, averaged over all runs, plotted graphically against generation. This approach was continued to be used, as it enables various search methods to be easily and fairly compared. One tweak to this approach is to replace the generation axis with evaluations (the product of generation and population size). This ensures that a fair comparison can be made when using different population sizes.

13.2 Design

As described in the Cartesian Genetic Program Production chapter, the code has been designed such that only two classes have to be altered when implement each test case: the Fitness and FunctionSet Classes. The specificFitness method is implemented by assigning a fitness to each chromosome which is the sum of the differences between the real symbolic function outputs and the outputs generated by each chromosome over a range of inputs. This results in a fitness value of zero representing a perfect solution. The range of inputs used to test the chromosomes, are fifty evenly spaced values between negative and positive one. The function set used contained the following operations: addition, subtraction, multiplication and protected division. Protected division was implemented by returning a value of one when the given denominator was less than 0.00000000001. All of the details described are in line with Janet Clegg's original experiments [1].

13.3 Results

The first experiment was to use the author's code, with the parameters used by Janet Clegg, to repeat the results obtained by Janet Clegg on the two regression problems. Figure 23 and Figure 24 give the author's results obtained for the first and second regression problem respectively. These plots give a comparison between the relative crossover percentages; no comparison is made with normal Cartesian Genetic Programming in these figures. The vertical axis shows the average best fitness over all 1000 runs at each evaluation; with zero representing a perfect solution.

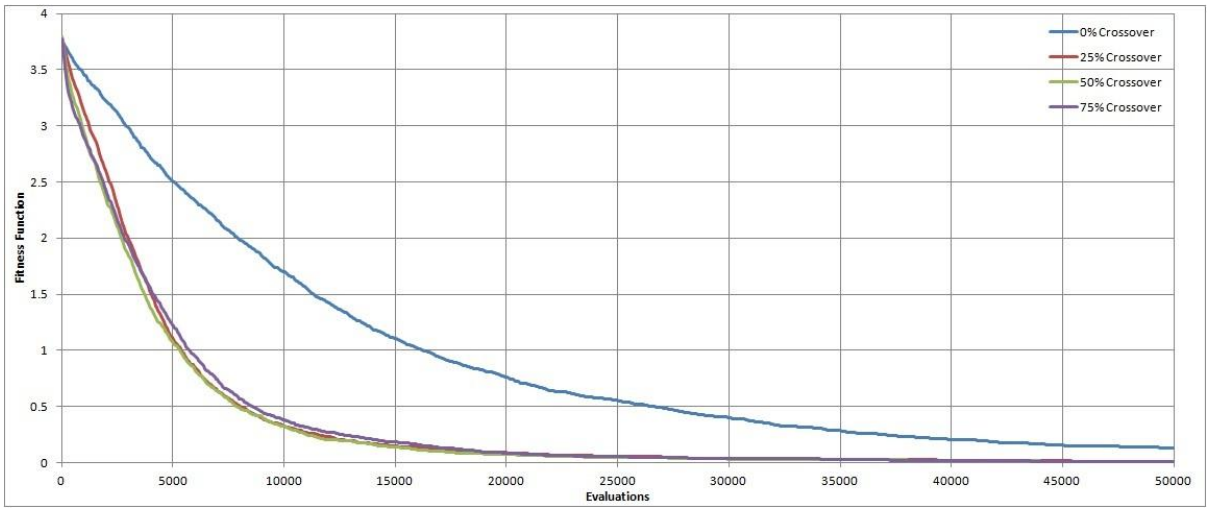

Figure 23 The author's Code applied to the symbolic regression problem $x^6 - 2x^4 + x^2$ using Janet Clegg's parameters given in Table 2

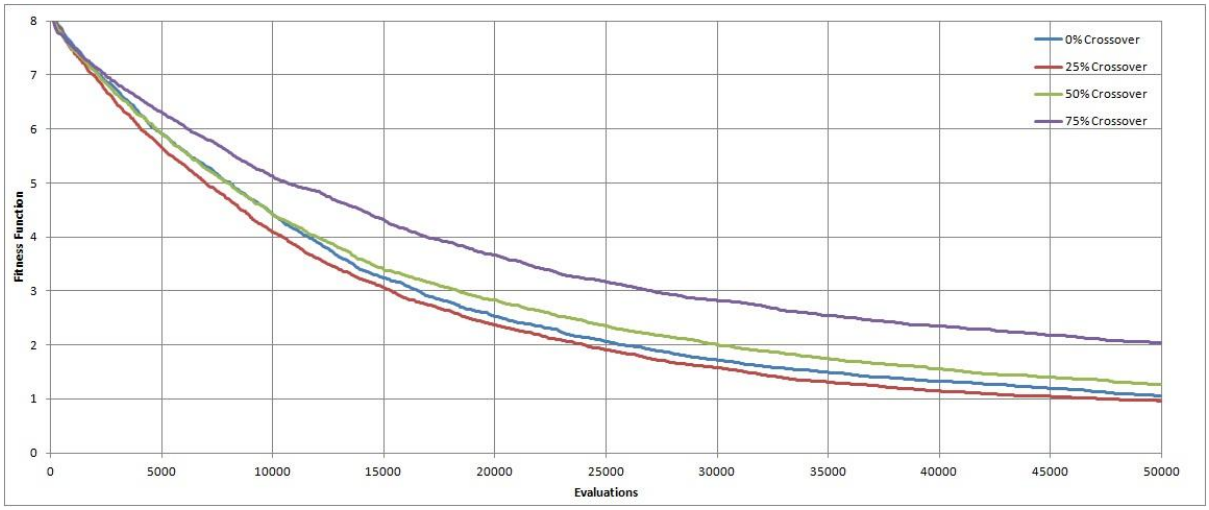

Figure 24 The author's Code applied to the symbolic regression problem $x^5 - 2x^3 + x$ using Janet Clegg's parameters given in Table 2

Figure 23 shows quite clearly that crossover is strongly aiding the search process, compared to regular Cartesian Genetic Programming implemented with a tournament selection scheme and using the floating point chromosome representation; shown as 0% crossover on the graph. It can also be seen from Figure 23, that higher levels of crossover percentage is more beneficial to the search process in the early stages of the search, and conversely lower crossover percentages are more beneficial towards the end; although this effect is subtle. Although Figure 23 shares characteristics with the results obtained by Janet Clegg, such as 0% crossover converging on a solution far more slowly than higher percentages, it is not exactly the same; which is strange as it should have been the same experiment. After

discussions with Janet Clegg, it was discovered that there were a number of differences between her and the author's implementation.

The first difference surrounds how mutation percentage is converted into the number of actual mutations carried out on each chromosome. To calculate the number of actual changes made to the chromosomes during mutation, the author's code takes the product of the mutation percentage and the number of nodes. This was due to the author considering each node to represent a single gene, and the mutation percentage referring to the number of genes mutated; although it is now understood that this is an unusual interpretation. Janet Clegg calculated the number of actual changes cause to each chromosome during mutation, to be the product of the mutation percentage and the number of parameters which describe the chromosomes; this is a more standard implementation of mutation. It was decided however, that although this may account for some of the differences between the author's and Janet Clegg's results, it would not affect future comparisons over the effectiveness of the crossover technique; as all experiments employ the author's implementation of mutation. In future test cases this difference becomes even less of an issue as all of the parameters are optimised for each experiment. An approximate conversion from the mutation percentages used by the author to a more standard mutation percentage is to third the values quoted in this project.

The second difference between the implementations is that Janet Clegg's code did not mutate the children generated by crossover; whereas the author's code did. It was decided that it is likely to be more beneficial to the search process if the children were mutated[17], and so this was continued to be carried out. Again, as for the first difference, this second difference may account for some of the discontinuities seen between the author's and Janet Clegg's results, but should not affect future experiments as long as the process used is consistent.

The final difference was surrounding the tournament sizes used. The author assumed that the tournament size used by Janet Clegg was four; although in hindsight there was no basis for this assumption. After discussions with Janet Clegg, it was decided that a different

[17] Although this was never proven.

tournament size was likely to have been used; possibly twenty. Reinspection of the original paper [1] discovered that no tournament size was specified.

All of these differences may count towards explaining the variation between the results shown in Figure 23, and those achieved by Janet Clegg. Despite these differences, Figure 23 still shows that the use of crossover is very effective when compared to Cartesian Genetic Programming, implemented with a tournament selection scheme, using a floating point chromosome representation, but without BLX-0 crossover.

Figure 24 however does not show the same promising results present in Figure 23 and also does not show the same results as reported by Janet Clegg. There is no indication that crossover is aiding the search process to any structured extent. These results are in complete contrast with those achieved by Janet Clegg, which showed higher crossover rates improving the search process. It is believed that these differences were not due to errors within the author's code, which appeared to be operating correctly in the previous example. It is therefore thought that the differences are due to the differences in how the Cartesian Genetic Programs were implemented; as described in the previous paragraphs.

Figure 25 shows the results of the second experiment, which was to repeat the first symbolic regression problem using parameters found by the author to produce the best results for each of the crossover percentages. The best parameters found for each level of crossover are given in Table 3. These results show again, that for this symbolic regression problem, crossover is very beneficial to the search process and increasing the levels of the crossover increases its effectiveness.

It can be seen by comparing Table 2 and Table 3, that the parameters found by the author to produce the best results are significantly different to those used by Janet Clegg. This may be an indication that the different implementations are influencing the search process or that the parameters used by Janet Clegg were not the most suitable. An interesting result shown in Table 3 is that different parameters were found to produce the best results for different levels of crossover percentage. This backs up the assumption that it is necessary to optimise the parameters for each individual experiment, as it cannot be guaranteed that the same parameters work effectively across a range of different investigations.

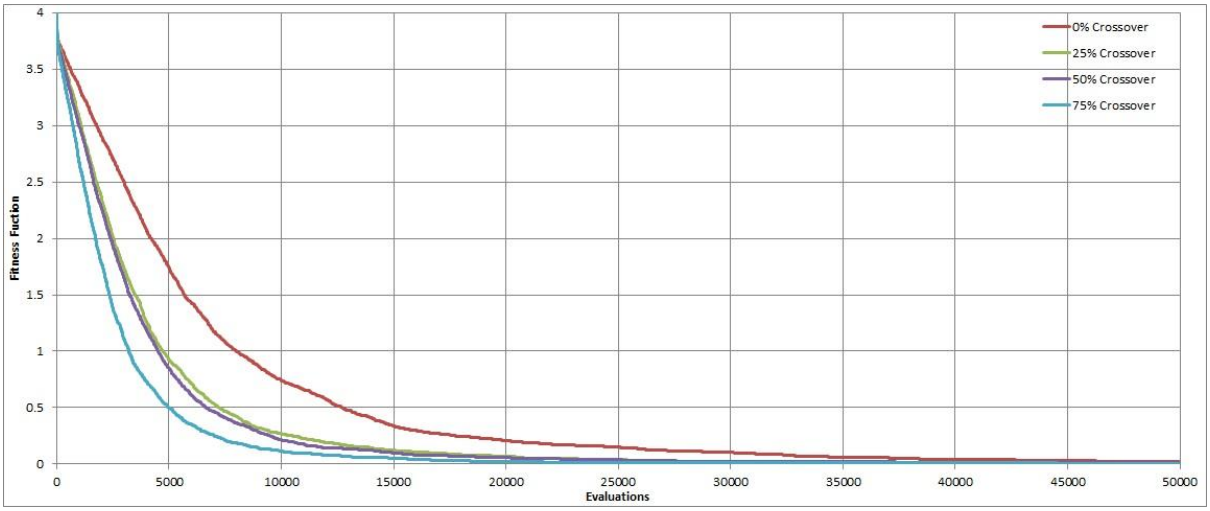

Figure 25 The author's Code applied to the symbolic regression problem $x^6 - 2x^4 + x^2$ using the author's parameters given in Table 3

Table 3 Best parameters found for the symbolic regression problem $x^6 - 2x^4 + x^2$

Crossover	Mu	Lambda	Mutation
0%	2	17	50%
25%	2	18	30%
50%	1	12	20%
75%	1	7	30%

The results of the investigation into whether the presence of a tournament selection scheme helps or hinders the normal[18] Cartesian Genetic Programs search process is shown in Figure 26. The parameters used were also found using the methods described in Appendix B and are given in Table 4. It should be noted that "No Crossover" is similar to "0% Crossover", except "0% crossover" uses tournament selection and "No crossover" does not.

Figure 26 shows how the presence of tournament selection appears to offer a slight advantage over Cartesian Genetic Programming implemented without a selection scheme; at least for this specific example. Interestingly it appears that early in the search process the presence of the selection scheme does not offer any advantage and it is only later in the search where it has a positive effect. It was indicated by Julian Miller (via emails) that this result does not tie in with his past experiences.

[18] Where "normal" refers to Cartesian Genetic Programming implemented without crossover.

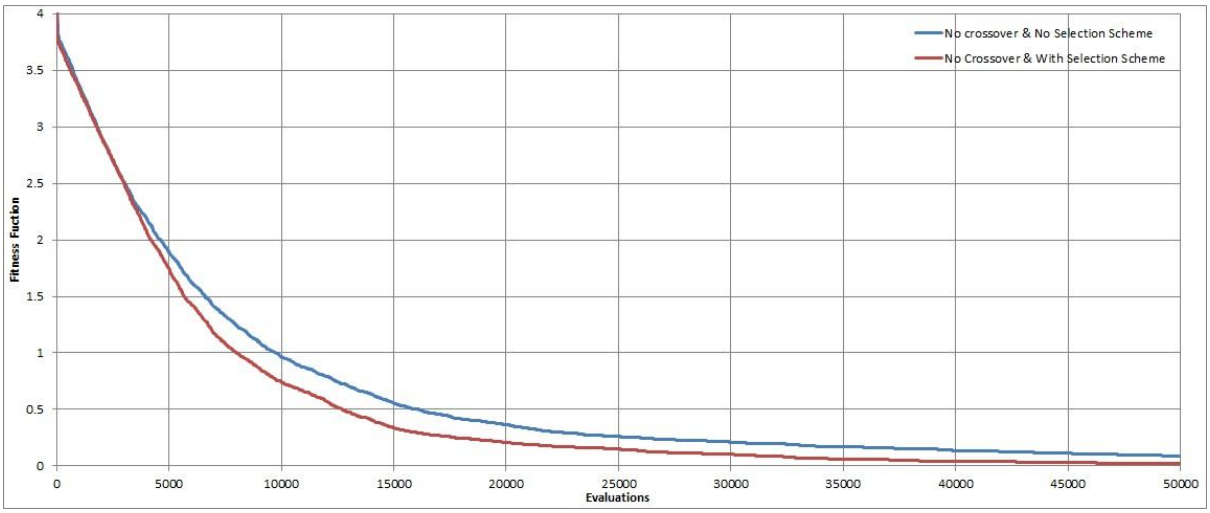

Figure 26 The author's Code applied to the symbolic regression problem $x^6 - 2x^4 + x^2$ to investigate the effect of a tournament selection scheme on a Cartesian Genetic Program

Table 4 The parameters used when evaluating the presence of a tournament selection scheme

Tournament Selection	Mu	Lambda	Mutation
yes	2	17	50%
no	2	14	40%

The final proposed experiment was to investigate if the floating point chromosome representation, required for the BLX-0 crossover technique, affected the search process. Figure 27 quite clearly shows that the floating point representation has little to no effect on the search process. The parameters used for this experiment were the same as used when investigating the effect of no tournament selection given in Table 4.

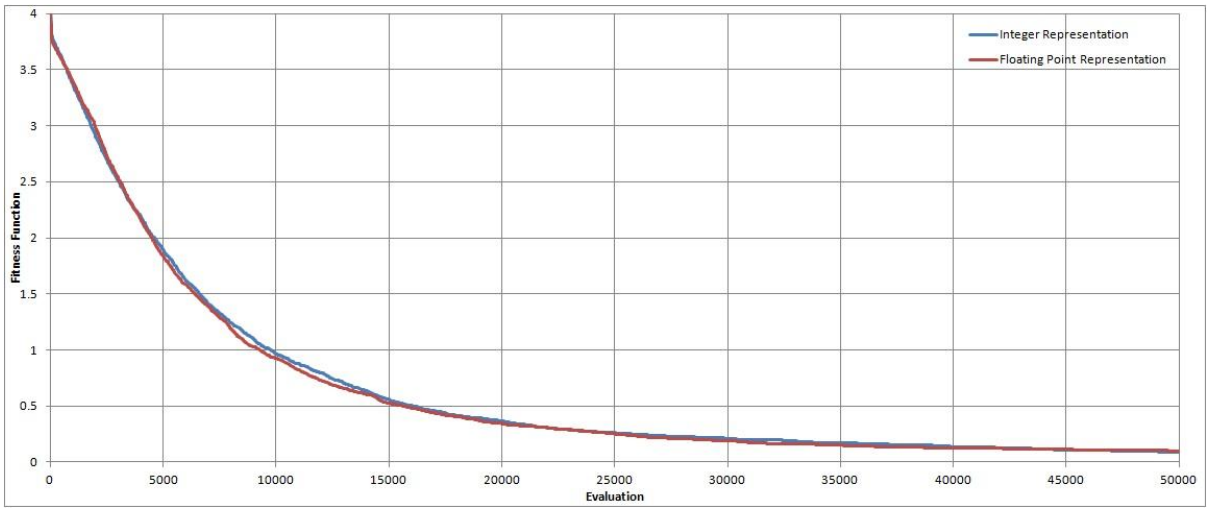

Figure 27 The author's Code applied to the symbolic regression problem $x^6 - 2x^4 + x^2$ to investigate the effect of the floating point representation

All of the experiments described in this chapter were then analysed to produce statistics which are often used to compare different Evolutionary Strategies. These statistics included, the average number of evaluations taken by each run to find the perfect solution; if a perfect solution was not found then the average was taken as if the best solution was found on the last evaluation. The average number of evaluations was used in place of the average number of generations as used by Janet Clegg, as the average number of evaluations enables comparisons with experiments which used different population sizes. Another statistic often used for the comparison of Evolutionary Strategies is Computational Effort; described in Appendix C. Lower Computational Effort values indicate a better search process. The statistics described in this paragraph are given in Table 5 for all the experiments investigated in this chapter.

Table 5 Average Evaluations and Computational Effort for all the experiments described in this chapter

Experiment Description	Symbolic Problem	Average Evaluations	Computational Effort
0% Crossover - Janet Clegg's Parameters	$x^6 - 2x^4 + x^2$	16,800	93,407
25% crossover - Janet Clegg's Parameters	$x^6 - 2x^4 + x^2$	6,800	45,008
50% crossover - Janet Clegg's Parameters	$x^6 - 2x^4 + x^2$	6,450	47,689
75% crossover - Janet Clegg's Parameters	$x^6 - 2x^4 + x^2$	7,000	45,008
0% Crossover - Janet Clegg's Parameters	$x^5 - 2x^3 + x$	31,650	278,150
25% crossover - Janet Clegg's Parameters	$x^5 - 2x^3 + x$	28,050	204,869
50% crossover - Janet Clegg's Parameters	$x^5 - 2x^3 + x$	30,700	224,156
75% crossover - Janet Clegg's Parameters	$x^5 - 2x^3 + x$	36,800	344,967
0% Crossover – Author's Parameters	$x^6 - 2x^4 + x^2$	9,984	60,329
25% crossover - Author's Parameters	$x^6 - 2x^4 + x^2$	6,037	46,406
50% crossover - Author's Parameters	$x^6 - 2x^4 + x^2$	5,358	46,406
75% crossover - Author's Parameters	$x^6 - 2x^4 + x^2$	4,036	39,637
No Tournament without Crossover - integer	$x^6 - 2x^4 + x^2$	13,066	88,894
No Tournament without Crossover - float	$x^6 - 2x^4 + x^2$	12,367	90,260
Tournament without Crossover - float	$x^6 - 2x^4 + x^2$	9,984	60,329

The average number of evaluations and computational effort should described the same information but from different viewpoints. A simple plot of the average number of evaluations against computational effort shows this linear relationship between the two statistics, see Figure 28. These values are all taken from this chapter.

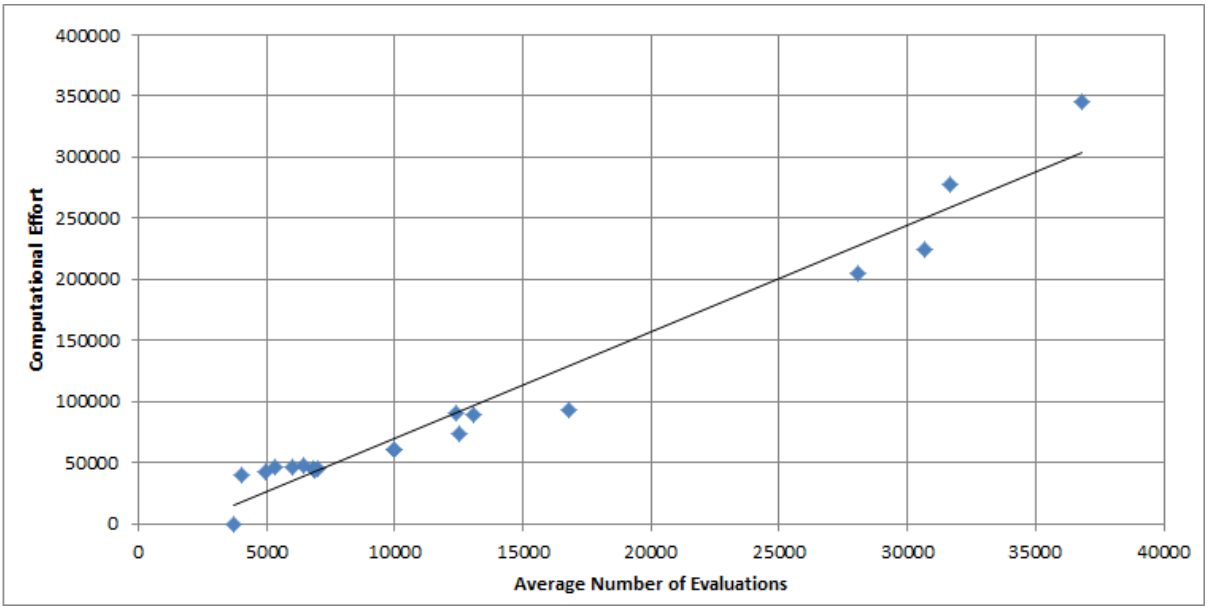

Figure 28 Depiction that the two shown analytical methods used portray the same information

As can be seen from Table 5, for the first symbolic regression problem ($x^6 - 2x^4 + x^2$), all of the experiments show that increasing the level of crossover decreases the average number of generations needed to find a solution and the computational effort required. Unfortunately this trend was not seen in the second symbolic regression problem ($x^5 - 2x^3 + x$), which does not show any trend of crossover aiding or hindering the search process. Table 5 also shows that using the integer form of the chromosomes has similar results to the floating point form. This indicates that representing the chromosomes in the floating point form has no effect on the overall search process and therefore does not contribute to the effectiveness of the crossover technique. Finally Table 5 also shows that tournament selection appears to aid the search process for the example shown in this chapter. It should be noted that all of the results obtained through observing the statistics shown in Table 5, can also be seen in the graphs also provided throughout this section.

13.4 Conclusion

The first conclusion to be made is that the trend described by Janet Clegg, BLX-0 offering a strong advantage when solving symbolic regression problems, has only been indentified for one of the two symbolic regression problems investigated. At first this was thought to be because of the differences in implementation, but after some thought and discussion, it was decided that this would change the raw values of the data e.g. the average fitness at each generation, but not the trends within the data. This was indeed true for the first symbolic

regression problem ($x^6 - 2x^4 + x^2$) and therefore does not explain why the author's results were so different from Janet Clegg's for the second ($x^5 - 2x^3 + x$).

It has been shown for one of the symbolic regression problems, that the encoding of the chromosomes in a floating point form (as opposed to an integer form), has no noticeable effect on the search process. It can therefore be concluded that the floating point form does not influence the effectiveness of the crossover technique. The requirement of BLX-0 crossover to use the floating point form therefore does not affect the search process. However, the extra level of decoding required to convert the floating point chromosome into an integer form, before the fitness can be calculated, will always have the penalty of incurring a larger time debt.

It appears from the results described in this chapter, that the use of a tournament selection scheme is offering a slight advantage to the search process. This result however is in contrast with Julian Millers previous experience.

13.5 Thoughts

One oddity in these results is the unusually high mutation rates found to be most suitable for these experiments. A possible reason for this may be the small number of nodes used for each chromosome (ten function nodes and one output node). This requires that a minimum mutation rate of ~10% has to be used otherwise no mutation is carried out[19]. As it is possible that multiple mutations have to be carried out at once to progress from local minima, it follows that the minimum mutation rate may have to be ~20% (in order to change two parameters in the chromosome when implementing mutation). Also, to change the actual number of alterations to the genotype the mutation rate must be varied in ~10% increments i.e. 11% and 12% mutation are likely to cause the same number of actual mutations.

One of the key issues which arose during these experiments was the realisation that the author's implementation differed from that of Janet Clegg's; as described in the Results section. Most of these differences relate to the implementation of mutation, it is thought however, that as long as the method used is consistent during the experiments this should

[19] Due to the author's usual way of interoperating mutation percentage.

not affect the analysis of the effectiveness of the crossover technique. The other difference in implementation related to the tournament size used, it is thought that this could have an effect on the analysis of the crossover technique. Future experiments therefore vary the tournament size to identify if this has an effect on the evolutionary process. These differences also count towards explaining the differences between the author's and Janet Clegg's results for the average number of evaluations used and the computational effort required to find a solution. Again it is thought that these differences do not affect the analysis as long as the author is consistent with his own methods.

14 Test Case 1: Symbolic Regression

Although two symbolic regression problems have already been investigated in the previous chapter, Repeating Janet Clegg's Experiments, it was decided that further investigation was needed to reach a fair conclusion over the effectiveness of BLX-0 crossover. This was due to one of the two symbolic regression problems showing results which indicated BLX-0 crossover not providing an advantage and because only one tournament size was investigated; four.

The first problem case, taken from John Koza's Book [13], is to re-discover the relationship $\cos(2x) = 1 - 2\sin(x)$. This is achieved by calculating the fitness as the difference between $\cos(2x)$ and the evolved solution over a range of inputs, using the possible function nodes: addition, subtraction, multiplication, protected division and sin(x). This investigation is undertaken using only a tournament size of four and is studied to increase the number of symbolic regression problems investigated; thus leading to stronger conclusions.

The second problem case is to repeat the symbolic regression problems investigated in the previous chapter, using tournament sizes of ten and twenty. This investigates if tournament size has any influence on the effectiveness of BLX-0 crossover. As previously mentioned, the paper published by Janet Clegg [1] does not indicate the tournament size used, but through discussions Janet Clegg has indicated that a tournament size of twenty was most likely.

14.1 The Experiments

The first experiment (for both of the described problem cases), is a comparison between normal Cartesian Genetic Programming implemented without crossover, with that which uses 0%, 25%, 50% and 75% crossover. This is to assess the relative effectiveness of BLX-0 crossover, not only to various crossover percentages, but also to a normal Cartesian Genetic Program[20]. Using these experiments it is also possible to assess the effect of employing a tournament selection scheme on Cartesian Genetic Programming. As the only difference

[20] That which used an integer chromosome form, implements no crossover, and employed no selection scheme.

between 0% crossover and normal Cartesian Genetic Programming is the presence of a tournament selection scheme; and the use of a floating point chromosome representation.

The second experiment (again applied to both problem cases), is a comparison between normal Cartesian Genetic Programming using the integer and floating point chromosome representation. This analyses if the floating point representation is effecting the search process.

All of these experiments are undertaken using evolutionary parameters which are found to be the most suitable; following the method described in Appendix B. As previously mentioned, this process of optimising parameters is long and tedious; it is however considered necessary if a fair comparison is to be made between the different methods. This is due to the complex nature of Evolutionary Computation, meaning small changes in the parameters can have a huge influence on the effectiveness of the search process.

14.2 Design

The basic design is exactly the same as used in the chapter Repeating Janet Clegg's Experiments. The fitness function again returns the sum of the absolute differences between the correct output, and the output of the evolved solution over a range of inputs. The range of inputs used for the $\cos(2x) = 1 - 2\sin(x)$ problem case are twenty values evenly spaced between zero and 2π. A slight difference between this symbolic regression problem and those used previously, is that this particular problem case requires two inputs; the variable 'x' and the integer value one. The range of inputs for the remaining symbolic regression problems, as used by Janet Clegg, are again fifty values evenly spaced between pulse and minus one.

The symbolic regression problem $\cos(2x) = 1 - 2\sin(x)$, and the first of the symbolic regression problems used by Janet Clegg ($x^6 - 2x^4 + x^2$), were implemented using ten function nodes. The second symbolic regression problem used by Janet Clegg, ($x^5 - 2x^3 + x$), initially also used ten nodes. However, after inspecting the results from using a tournament size of ten, it was indentified that $x^5 - 2x^3 + x$ represented a much more complex search space. The remaining experiment, that using a tournament size of twenty, was therefore carried out using twenty nodes; to attempt to quicken the search process.

This was undertaken as redundancy in the chromosomes is thought to aid the search process, Julian Miller et al [18].

14.3 Results

As described, the first experiment was an evaluation of how effectively the relationship $\cos(2x) = 1 - 2\sin(x)$ could be "re-discovered"; using only a tournament size of four.

Figure 29[21] appears to indicate that BLX-0 crossover is providing no advantage over normal Cartesian Genetic Programming (implemented without crossover or tournament selection). To clarify, the lower blue line shown beneath the red line (0% crossover), is the "Normal" plot, and the upper lighter blue line represents 50% crossover. The plot shows that "Normal" and "0% Crossover" are similar, indicating that the presence of tournament selection is having little effect on the search process; as the presence of tournament selection is the only meaningful difference between the two searches. The three remaining plots (25%, 50% and 75% crossover), are grouped together and located above the other two plots; indicating that the presence of crossover is hindering the search process.

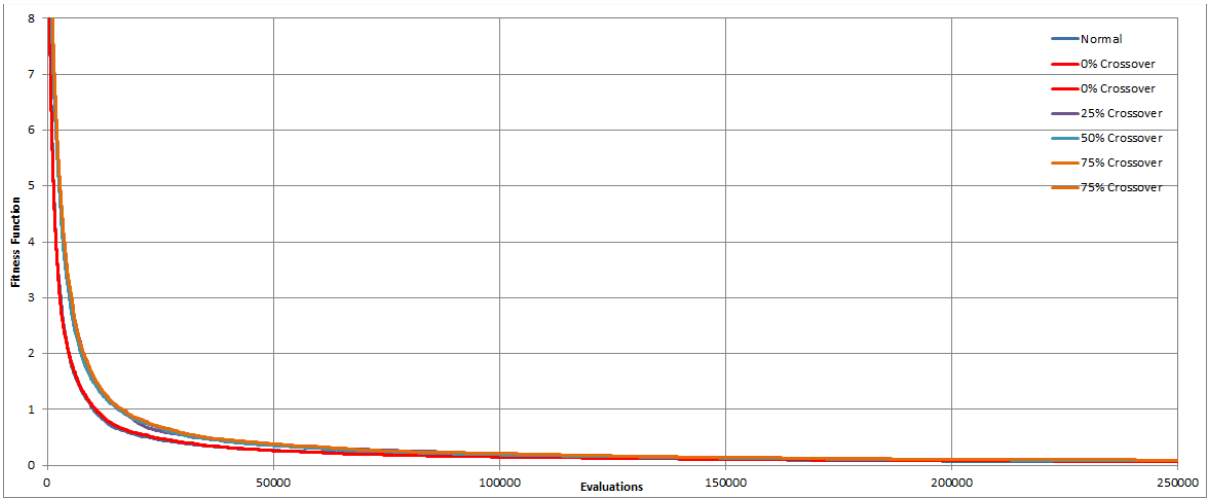

Figure 29 Various levels of crossover applied to the Cos(2x) Symbolic Regression Problem - tournament size four

Table 6 shows the best parameters which were found for the $\cos(2x) = 1 - 2\sin(x)$ problem case; as previously described these were found using the method described in Appendix B. It can be seen in Table 6, that for all of the experiments undertaken for this particular problem case, a mu value of one was found to give the best results. There is

[21] The reason some of the plots shown have two data labels is due to Microsoft excel limiting the number of data points allowed per plot to 64,000. Therefore some of the data had to be displayed using two plots; both coloured the same for ease of reading.

however a wide range of lambda and mutation values. As previously mentioned, the mutation method used by the author takes the mutation percentage as the percentage of genes which are changed, where a gene refers to one node; function or output. For example, if there were eight function nodes and two output nodes, a mutation percentage of 10% would cause one of the parameters of one of the nodes to be changed. This explains the high mutation rates seen throughout this project. An approximate conversion to a more conventional mutation rate (where the percentage refers to the percentage of parameters changed) is to third the figures quoted by the author. For example, a mutation rate of 30% quoted during this paper is equivalent to approximately 10% mutation using a more conventional mutation strategy.

Table 6 Best Parameters found for the Cos(2x) Symbolic Regression Problem - tournament size four

Experiment Description	Mu	Lambda	Mutation
Normal CGP - integer representation	1	7	35%
Normal CGP - floating point representation	1	7	35%
0% Crossover	1	5	35%
25% Crossover	1	11	15%
50% Crossover	1	8	20%
75% Crossover	1	3	20%

Table 7 shows the average evaluations and the computational effort required to find a solution to the $\cos(2x) = 1 - 2\sin(x)$ problem case. It can be seen that for both average evaluations and computational effort, increasing the level of crossover increases the effectiveness of the search (when comparing different levels of crossover). It also shows that for both statistics, 75% crossover is more effective than normal Cartesian Genetic Programming; indicating that BLX-0 crossover is beneficial to the search process. This is in contrast to the results seen in Figure 29.

Table 7 Statistics used to analyse the Cos(2x) symbolic regression experiments - tournament size four

Experiment Description	Average Evaluations	Computational Effort
No Crossover - Integer representation	213,339	3,164,253
No Crossover - Floating Point representation	212,603	3,580,072
0% Crossover	220,211	3,690,339
25% crossover	208,560	3,253,876
50% crossover	202,990	3,189,442
75% crossover	200,439	3,114,839

Figure 30 shows a comparison between the integer and floating point chromosome representation[22], when applied to the $\cos(2x)$ symbolic regression problem. This is to assess whether the floating point representation is affecting the search process. Figure 30 clearly shows that the floating point representation is not affecting the search process; in line with all previous results. Table 7 also shows this trend, as the averages evaluations required to find a solution is approximately the same for both representations. Oddly however, the Computational Effort statistic does not show this trend and is the first instance of any results not indicating that the floating point representation has no effect on the search process.

[22] When using normal Cartesian Genetic Programming i.e. no crossover.

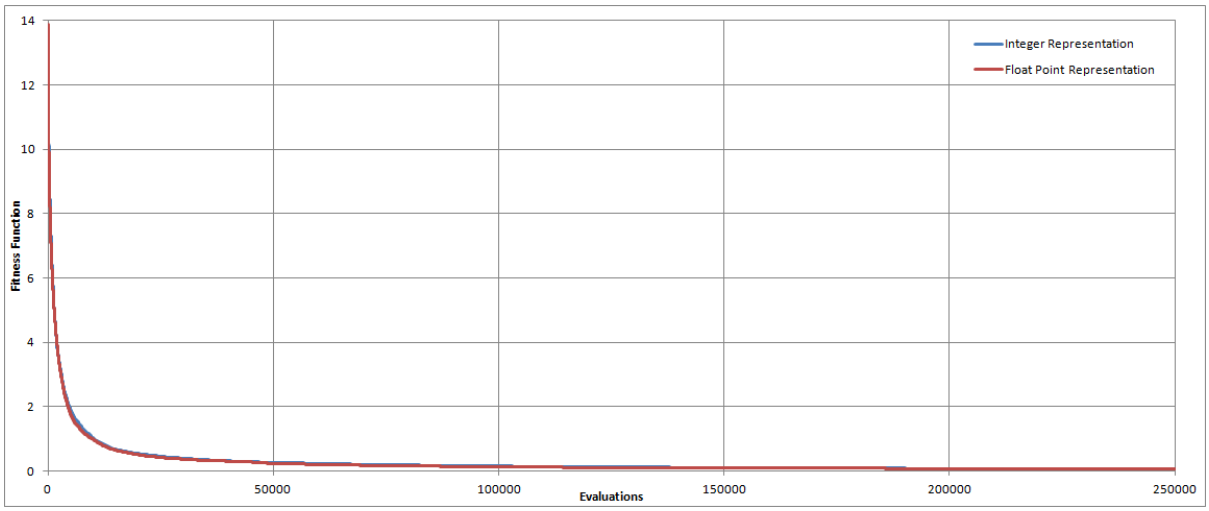

Figure 30 Comparison between the integer and floating point representation applied to the Cos(2x) symbolic regression problem when using Cartesian Generic Programming with no crossover or tournament selection

Figure 31 gives the results of applying normal Cartesian Genetic Programming and that which uses 0%, 25%, 50% and 75% crossover to the $x^6 - 2x^4 + x^2$ problem case when using a tournament size of ten. Figure 31 clearly shows that increasing the levels of crossover increases the effectiveness of the search process; as seen previously when using a tournament size of four. It also shows that all levels of crossover investigated outperformed normal Cartesian Genetic Programming. This result is not seen however for the $x^5 - 2x^3 + x$ symbolic regression problem, Figure 32, which shows different levels of crossover being most effective at different stages of the search; this is indicated by the intercepting of the plots. This intercepting was also noted by Janet Clegg [1] and led to the implementation of variable crossover, where a high level of crossover percentage is used initially, which is then reduced as the search progresses. The results given in Figure 32, show that for this symbolic regression problem, higher levels of crossover percentages were most effective at the start of the search, and lower levels towards the end. Figure 32 also shows that after the maximum allowed evaluations, normal Cartesian Genetic Programming produces the worst results, followed by 0% crossover; which represents normal Cartesian Genetic Programming implemented with a tournament selection scheme and using the floating point chromosome representation. 50% crossover appears to produce the best results overall.

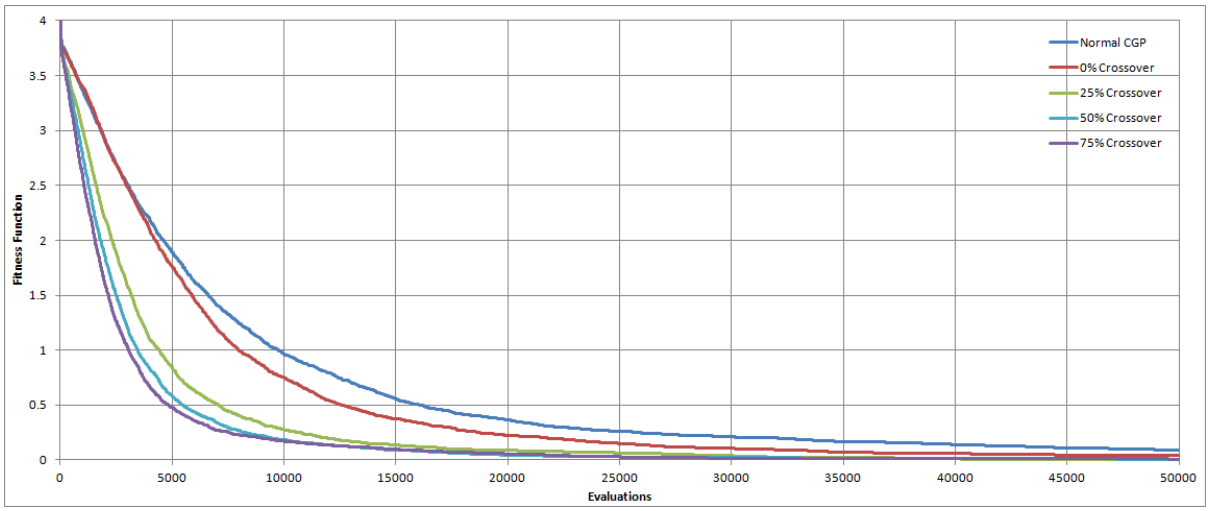

Figure 31 Various levels of crossover applied to the $x^6 - 2x^4 + x^2$ Symbolic Regression Problem - tournament size ten

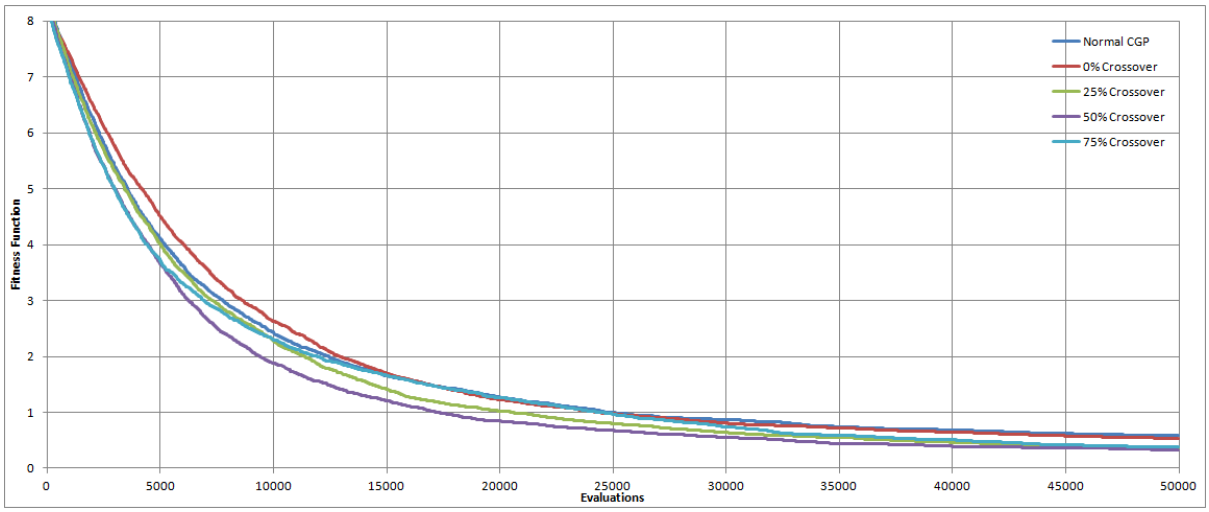

Figure 32 Various levels of crossover applied to the $x^5 - 2x^3 + x$ Symbolic Regression Problem - tournament size ten

Table 8 and Table 9 give the average evaluations and the computational effort statistics for the $x^6 - 2x^4 + x^2$ and the $x^5 - 2x^3 + x$ symbolic regression problems respectively; when using a tournament size of ten. Table 8 and Table 9 both show that there is a level of crossover, for both symbolic regression problems, which out performs normal Cartesian Genetic Programming; indicated by both Average Evaluation and Computational Effort. They also show a trend of, increasing the level of crossover, increases the effectiveness of the search. It can also be seen from comparing the values in Table 8 and Table 9 that the symbolic regression problem $x^5 - 2x^3 + x$, is much more challenging than $x^6 - 2x^4 + x^2$; this is a result also noted by John Koza [13] when he was studying the two symbolic regression problems.

Table 8 Statistics used to analyse the $x^6 - 2x^4 + x^2$ symbolic regression experiments - tournament size ten

Experiment Description	Average Evaluations	Computational Effort
No Crossover - Integer representation	13,066	88,894
No Crossover - Floating Point representation	12,367	90,260
0% Crossover	9,606	65,036
25% crossover	5,884	39,637
50% crossover	4,661	43,459
75% crossover	4,381	43,459

Table 9 Statistics used to analyse the $x^5 - 2x^3 + x$ symbolic regression experiments - tournament size ten

Experiment Description	Average Evaluations	Computational Effort
No Crossover - Integer representation	25,769	207,690
No Crossover - Floating Point representation	24,710	203,750
0% Crossover	24,600	184,977
25% crossover	20,663	142,623
50% crossover	18,177	132,540
75% crossover	20,416	130,809

The best parameters found for these two experiments are given in Table 10 and Table 11 for the respective symbolic regression problems. Once again very low mu values were found to produce the best results (values of one or two). When not using crossover the lambda values found to produce the best results for the two symbolic regression problems were quite different; fourteen for what is considered the easier problem and five for the other. When using crossover the lambda values appeared to decrease as the crossover percentage increased.

An interesting point to note is that when population size (mu + lambda) is the same as the tournament size, the children are only produced from the elite members of the population. This has the implication that as population size approaches the tournament size, the effect of the tournament selection scheme is reduced. This is interesting because for all of the

symbolic regression problems discussed so far in this chapter, the effect of the tournament selection scheme is reduced or completely removed for the higher orders of crossover. As the population size found to be most suitable reduced with crossover percentage.

The mutation rates given in Table 10 and Table 11 were fairly consistent for the two symbolic regression problems (40%, 50%), except when using 75% crossover on the harder problem, where a mutation percentage of 80% was found to produce the best results.

Table 10 Best Parameters found for the $x^6 - 2x^4 + x^2$ Symbolic Regression Problem - tournament size ten

Experiment Description	Mu	Lambda	Mutation
Normal CGP - integer representation	2	14	40%
Normal CGP - floating point representation	2	14	40%
0% Crossover	2	22	50%
25% Crossover	1	22	40%
50% Crossover	1	18	40%
75% Crossover	1	11	50%

Table 11 Best Parameters found for the $x^5 - 2x^3 + x$ Symbolic Regression Problem - tournament size ten

Experiment Description	Mu	Lambda	Mutation
Normal CGP - integer representation	1	5	40%
Normal CGP - floating point representation	1	5	40%
0% Crossover	2	21	50%
25% Crossover	1	11	50%
50% Crossover	1	10	50%
75% Crossover	1	10	80%

Figure 33 shows that crossover is offering an advantage to the search process for the symbolic regression problem $x^6 - 2x^4 + x^2$ when using a tournament size of twenty. Figure 34 also indicates this result for the second symbolic regression problem $x^5 - 2x^3 + x$, which shows the final average fitness values in order of crossover percentage, with the higher percentages producing the best result. In both of the figures, "Normal" Cartesian Genetic Programming, with no crossover, produces the worst results when compared to those which employed crossover. Once again Figure 34 shows, that for the $x^5 - 2x^3 + x$ symbolic regression problem, the plots intercept each other at various points. It is therefore

speculated again that this is an indication that variable crossover might produce even more promising results.

It should be noted that for the symbolic regression problem $x^5 - 2x^3 + x$, shown in Figure 34, the number of function nodes was increased to thirty; this was to accommodate the discovery that this was a much more complex search space.

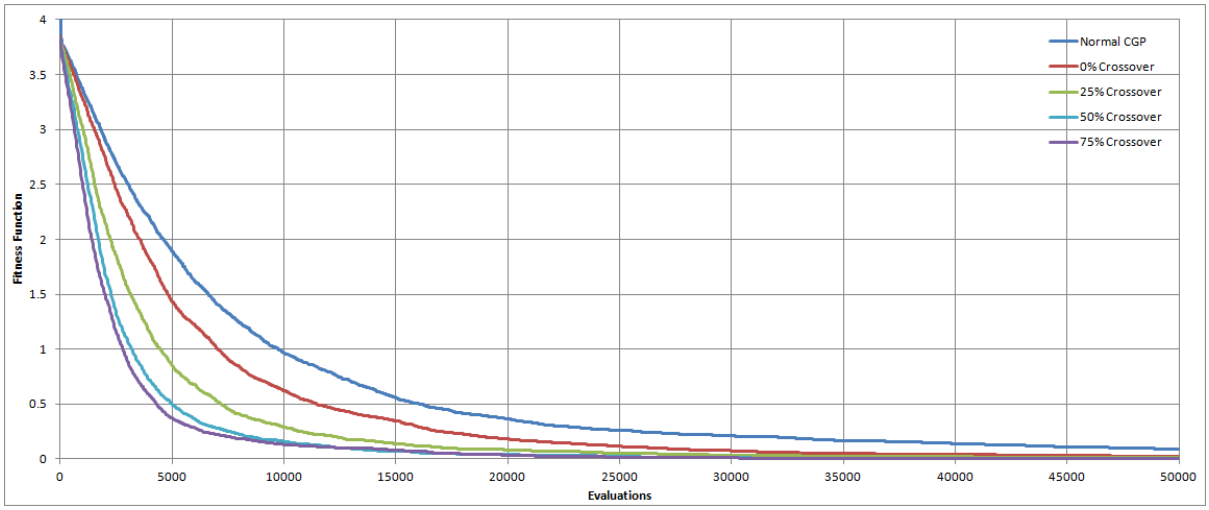

Figure 33 Various levels of crossover applied to the $x^6 - 2x^4 + x^2$ Symbolic Regression Problem - tournament size twenty

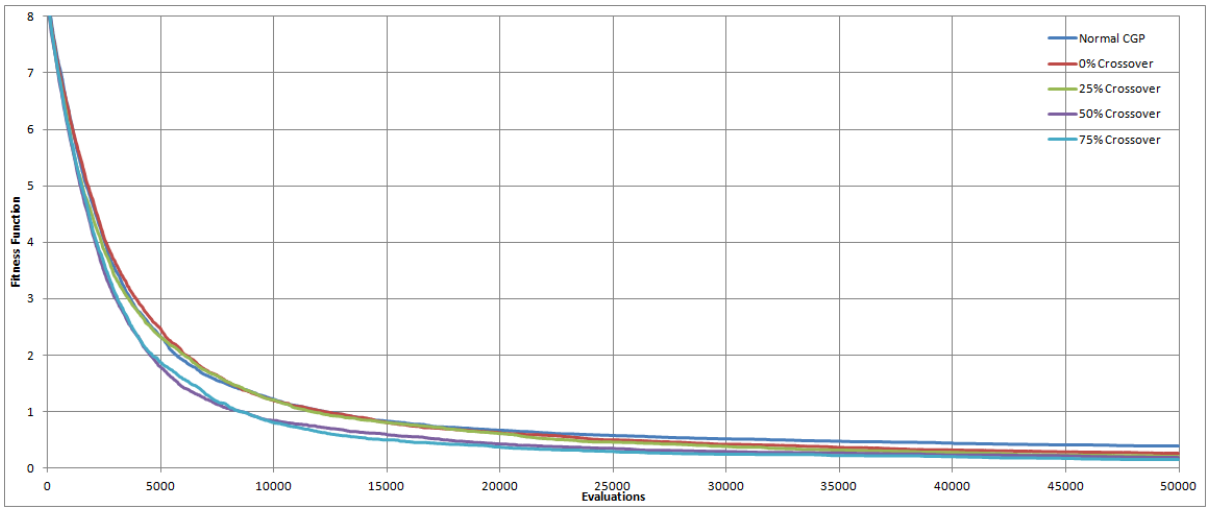

Figure 34 Various levels of crossover applied to the $x^5 - 2x^3 + x$ Symbolic Regression Problem - tournament size twenty

The same results shown in Figure 33 and Figure 34 are also present in Table 12 and Table 13; increasing the levels of crossover offers an increase to the effectiveness of the search process. In both cases the presence of crossover out performs "normal" Cartesian Genetic Programming implemented without crossover; when using a tournament size of twenty. The

results in these tables back up the assumption that the second symbolic regression problem ($x^5 - 2x^3 + x$) is much more challenging than the first ($x^6 - 2x^4 + x^2$).

Table 12 Statistics used to analyse the $x^6 - 2x^4 + x^2$ symbolic regression experiments - tournament size twenty

Experiment Description	Average Evaluations	Computational Effort
No Crossover - Integer representation	13,066	88,894
No Crossover - Floating Point representation	12,367	90,260
0% Crossover	8,620	58,097
25% crossover	5,938	50,000
50% crossover	4,186	33,333
75% crossover	3,783	33,333

Table 13 Statistics used to analyse the $x^5 - 2x^3 + x$ symbolic regression experiments - tournament size twenty

Experiment Description	Average Evaluations	Computational Effort
No Crossover - Integer representation	25,769	207,690
No Crossover - Floating Point representation	24,710	203,750
0% Crossover	19,121	142,180
25% crossover	16,422	118,390
50% crossover	13,360	112,434
75% crossover	11,846	100,434

Table 14 and Table 15 show the best parameters found for the two symbolic regression problems under investigation; when using a tournament size of twenty. As for when the tournament size was ten, the mu values which produce the best results were very low (one or two). This appears to be a trend, as both the author and Janet Clegg have found low values of mu to work well for these types of problems. Again, as for when the tournament size was ten, when not implementing crossover higher lambda values were found to work better for the easier problem, than for the harder. The population sizes were above the tournament size in most cases, except for the harder symbolic regression problem ($x^5 - 2x^3 + x$) when implementing higher crossover percentages. The mutation

percentages found to produce the best results were broadly spread across the range of 40% to 70%.

Table 14 Best Parameters found for the $x^6 - 2x^4 + x^2$ Symbolic Regression Problem - tournament size twenty

Experiment Description	Mu	Lambda	Mutation
Normal CGP - integer representation	2	14	40%
Normal CGP - floating point representation	2	14	40%
0% Crossover	2	28	70%
25% Crossover	1	28	50%
50% Crossover	1	33	50%
75% Crossover	1	30	60%

Table 15 Best Parameters found for the $x^5 - 2x^3 + x$ Symbolic Regression Problem - tournament size twenty

Experiment Description	Mu	Lambda	Mutation
Normal CGP - integer representation	1	5	40%
Normal CGP - floating point representation	1	5	40%
0% Crossover	1	26	50%
25% Crossover	1	33	60%
50% Crossover	1	20	40%
75% Crossover	1	20	40%

Table 12 and Table 13 also show that the use of the floating point representation is producing no significant deviations from the results obtained using the integer form; indicating that it is not affecting the search process. This result is also present in Figure 35 and Figure 36. One oddity is that Figure 36 shows the floating point chromosome representation outperforming the integer representation between 5000 and 30000 evaluations. The author assumes this deviation is due to the random nature of Evolutionary Strategies; although it was thought that averaging over 1000 runs would eliminate this.

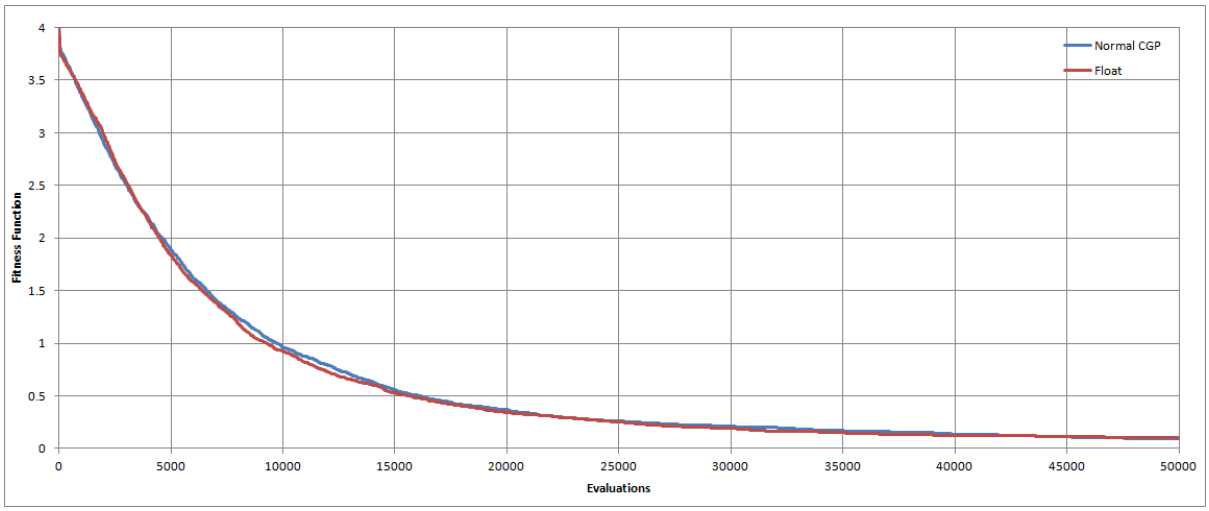

Figure 35 Comparison between the integer and floating point representation applied to the $x^6 - 2x^4 + x^2$ symbolic regression problem when using Cartesian Generic Programming with no crossover or tournament selection

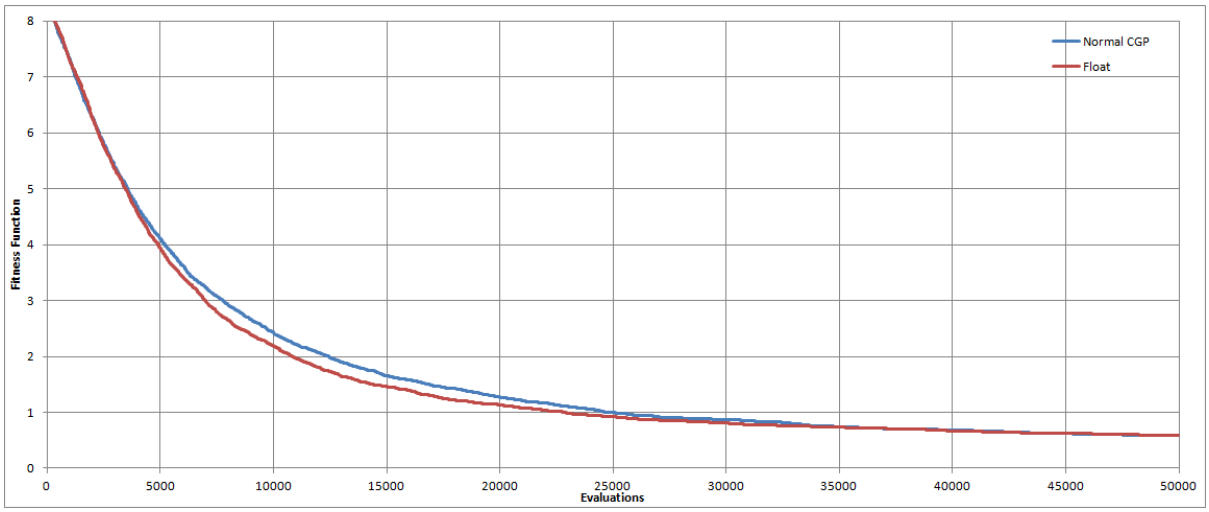

Figure 36 Comparison between the integer and floating point representation applied to the $x^5 - 2x^3 + x$ symbolic regression problem when using Cartesian Generic Programming with no crossover or tournament selection

To aid comparison, some of the results obtained in this and the previous chapter are given in Table 16, which shows the computational effort required by the symbolic regression problems used by Janet Clegg for all of the experiments where the parameters have been optimised. This table shows promising results for the application of crossover to symbolic regression type problems; as in almost every case the effectiveness of the search increases with crossover percentage. The effectiveness of the search is also increasing with tournament size; although it is highly unlikely that this result would carry on indefinitely.

Table 16 All of the Computational Effort results for all of the experiments undertaken on the two Symbolic Regression problems taken from Janet Clegg's Paper

Experiment Description	$x^6 - 2x^4 + x^2$			$x^5 - 2x^3 + x$	
Tournament Size	4	10	20	10	20
No Crossover - Integer representation	888,894	88,894	88,894	207,690	207,690
0% Crossover	60,329	65,036	58,097	184,977	142,180
25% crossover	46,406	39,637	50,000	142,623	118,390
50% crossover	46,406	43,459	3,333	132,540	112,434
75% crossover	39,637	43,459	3,333	130,809	100,434

The method by which Janet Clegg implemented BLX-0 crossover uses a tournament selection scheme to select the parents from the population. Whether this selection scheme is beneficial to the search process can be evaluated by comparing the results of "No Crossover - Integer representation" to those generated by "0% crossover". This is due to the fact that 0% crossover carries out no crossover (hence 0%) but does implement tournament selection; instead of only elitism. Therefore the only two differences between "Normal" Cartesian Genetic Programming and "0% crossover", is that the latter uses a floating point representation and tournament selection. It has been shown that the floating point representation poses no significant effect to the search process and therefore the effect of tournament selection can be seen in isolation. Based on these assumptions, Table 16 shows that the tournament selection scheme offers an advantage for the two symbolic regression problems used by Janet Clegg. The same result is not however found for the cos(2x) symbolic regression problem, Table 7, where the computational effort of "0% crossover" (tournament selection) is much higher than "normal" (no tournament selection).

14.4 Conclusion

It appears from the results provided in this, and the previous chapter, that Cartesian Genetic Programming implemented with BLX-0 crossover (as used by Janet Clegg in her paper [1]) is more effective at finding solutions to symbolic regression type problems than Cartesian Genetic Programs implemented without. In all cases this has been shown both graphically and via commonly quoted statistics (average evaluations and computational effort).

It has been decided that there is not enough evidence to conclude how the presence of a tournament selection scheme is affecting the search process e.g. offers and

advantage/disadvantage or does not affect the search process. Of the three symbolic regression problems investigated, two indicated that it offered an advantage, and the third (cos(2x)) showed a disadvantage.

It has been concluded that the floating point representation required by BLX-0 crossover does not alter the search process to any significant extent; this was shown in all cases except for the cos(2x) problem case, where the computational effort was higher for the floating point representation, Table 7. It was however also shown in Table 7 that the floating point representation did not significantly change the average evaluations, only the computational effort, which is an unexpected result[23].

Throughout this chapter low mu values were always found to produce the best results; values of one or two. There were however a wide range of lambda values between three and thirty three; with normal Cartesian Genetic Programming usually preferring lower lambda values. This may indicate that the crossover operator prefers larger population sizes, possibly to ensure more diverse populations; a situation in which crossover has the largest influence.

The mutation percentages also covered a wide range between 15% and 80%, although it appeared that different symbolic regression problems preferred different levels of mutation percentage.

14.5 Further Work

If more time were available, the optimal parameters would have been found for the $x^5 - 2x^3 + x$ symbolic regression problem using a tournament size of four; the obvious absence from Table 16. Again if more time were available, further symbolic regression problems would have been studied over a wider range of tournament sizes; to find the point where increasing the tournament size is no longer beneficial to the search process.

Another interesting investigation could be to evaluate how effective BLX-0 crossover is when implemented without a selection scheme. This would be possible by always selecting the parents to be the two elite members of the population and generating all the children from these; using BLX-0 crossover and mutation. Although this would cause the mu

[23] This experiment was repeated at a later date to confirm this strange result.

parameter to be fixed at two[24], it is not thought that this would be an issue, as the majority of the time two was found to be the best value (or one, which is close).

Another investigation which would have been undertaken if time had permitted was to implement the variable crossover as used by Janet Clegg [1]. There has been evidence that this may have offered an advantage, Figure 32 and Figure 34, which both showed the plots intersecting each other at various stages. It was however decided that it would be more beneficial to investigate different test cases than try every variation on the crossover technique.

14.6 Thoughts

It is thought by the author, that the type of crossover (BLX-0) being employed might be acting as a restricted type of mutation. This is thought for two reasons, the first is that when Janet Clegg implemented variable crossover, it was implemented so that the crossover level reduced as the search progressed, this is a technique often employed when implementing variable mutation rates. Secondly, BLX-0 crossover selects a restricted random value for each gene limited by the parent's genes. This causes massive alterations[25] at the start of the search (when the population is very diverse) and much less when converging on a solution; again very much like variable mutation. This second point also causes slightly higher alterations to take place after regular mutation makes a large beneficial change, which is carried through to the next population. When this occurs the population becomes slightly more diverse, this allows crossover to make larger changes to the chromosomes initially after a jump in the search space, but causing less changes again during convergence.

[24] Or three, and conducting crossover using all three parents etc.
[25] Where alterations can be thought of as mutations under a different name.

15 Test Case 2: Synthesis of Boolean Logic

Synthesis of Boolean logic was chosen for the next problem case because, like symbolic regression, it showcases the ability for Cartesian Genetic Programming to create programs; rather than simply optimise a number of parameters like most other Evolutionary Strategies. A secondary reason for selecting this problem case, is that Cartesian Genetic Programming was originally developed for the synthesis of Boolean logic and so holds some historic value. This problem case is discussed in further detail in the Possible Test Cases chapter.

15.1 The Experiments

Two circuits were selected to test the effectiveness of BLX-0 crossover on this test case, as it was felt a single example would not be sufficient to draw strong conclusions; due to time restraints more could not be undertaken. The two chosen circuits selected for this test case were the full adder and the four bit even parity generator. A truth table showing the operation of a full adder is given in Table 17 and the conventional logic configuration is given in Figure 37, taken from [58]. As can be seen, this circuit comprises of three inputs and two outputs; this is the first instance of a multiple output problem case. The truth table showing the operation of the four bit even parity generator is also provided in Table 18.

Table 17 Truth Table of Full Adder

A	B	Cin	Q	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

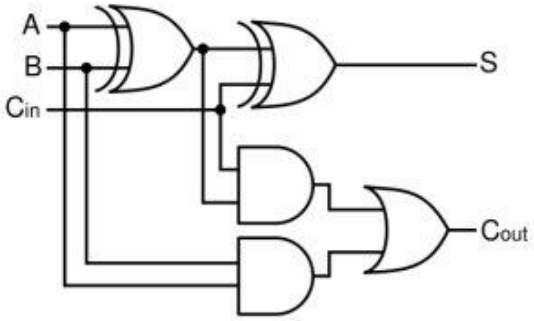

Figure 37 Conventional Full Adder circuit configuration

Table 18 Truth Table for Four Bit Even Parity Generator

A	B	C	D	Q
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

When evolving circuit configurations for the full adder, the following logic gates were made available: AND, OR, NAND, NOR and XOR. When evolving circuit configurations for the four bit even parity generator, the following logic gates were be made available: AND, OR, NAND and NOR. The absence of the XOR gate for the four bit even parity generator case produces a much more challenging search space, as described in Julian Millers book [15].

For both circuits, experiments are undertaken for tournament sizes four and twenty, this investigates the effect of tournament size on the search process[26]. For all experiments

[26] Unfortunately time does not permit a more rigorous sweep of tournament sizes.

undertaken in this chapter, the number of function nodes is set to thirty. It is understood that these circuits can be implemented with far fewer logic gates, however it has been shown that redundancy in the function nodes can aid the search process [18].

For both circuits, using both tournament sizes, experiments were carried out without crossover (normal), and with 0%, 25%, 50% and 75% crossover. This investigated the effect of BLX-0 crossover on the search process. Using these experiments it was also possible to analyse the effect of employing a tournament selection scheme on Cartesian Genetic Programming; by comparing the “normal” and the "0% Crossover"[27] results, as previously mentioned. For both of these circuits, experiments were also undertaken without BLX-0 crossover, and without tournament selection, but using the floating point representation of the chromosomes. This investigated the effect of the floating point form on the search process. As previously discussed, by isolating the effect of the floating point form and the effect of the tournament selection scheme, the effectiveness of BLX-0 crossover can be fairly analysed.

As for all the previous problem cases, the experiments were undertaken using what were found to be suitable evolutionary parameters. This process involved finding suitable mu, lambda and mutation percentage values. As mentioned in previous chapters, the process of finding suitable values is very time consuming but is necessary if fair comparisons are to be made between the techniques.

As with the previous problem cases, all the experiments were averaged over 1000 runs with the results provided graphically and via commonly quoted statistics: average evaluations and computational effort.

15.2 Design

The design and integration of the fitness classes was relatively simple, due to the modular approach which had been followed when designing and implementing the code. The fitness assigned to each chromosome is the number of incorrect outputs generated by the evolved circuit, when all possible inputs were swept. The inputs for each line of the truth table are used as the inputs to the current chromosome under evaluation. The outputs are taken as

[27] 0% Crossover actually implements no crossover but use the floating point representation for chromosomes and employs a tournament selection scheme to select the members of the next generation.

the outputs of the current chromosome; and then compared to the corresponding outputs of the truth table. For the case where there are multiple outputs (full adder), a fitness value is assigned for each output i.e. one row of the full adder truth table can increment the fitness by two; in the case where both of the outputs are incorrect.

The design of the Boolean logic function set, used by this problem case, was also very simple due to the modular structure of the code and because JAVA contains many built in bit wise operands.

The testing followed a similar strategy to that used by the symbolic regression problem case. The chromosomes generated by the Cartesian Genetic Program had their fitnesses calculated manually to ensure the assigned fitnesses were as expected. The manual calculation of the fitnesses was achieved using an excel spreadsheet, which natively contain the Boolean logic expressions; AND, OR and NOT. The logic expressions NAND and NOR were generated using a combination of these gates. The final expression, XOR, was implemented via a excel macro written in the Visual Basic programming language, see Figure 38. Using these logic gates, the circuits generated by the Cartesian Genetic Program could be implemented within excel and there fitnesses calculated.

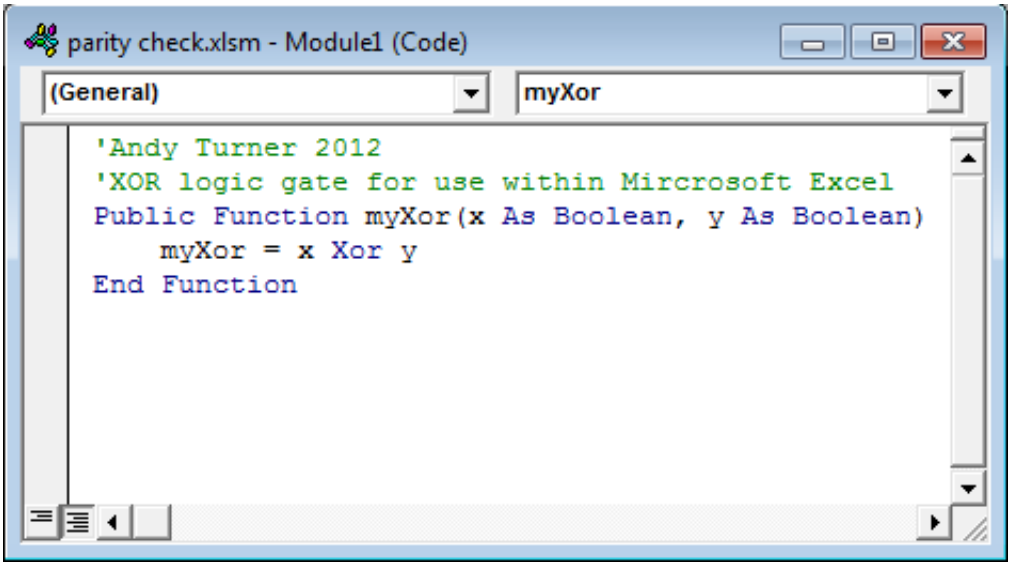

Figure 38 Visual Basic Macro to implement XOR logic gate in Microsoft's Excel

15.3 Results

The first experiment was the application of normal Cartesian Genetic Programming and Cartesian Genetic Programming implemented with 0%, 25%, 50% and 75% crossover, to the full adder problem case using a tournament size of four. Figure 39 shows that BLX-0 crossover is offering no advantage to the search process, with the Cartesian Genetic Program implemented without crossover outperforming all strengths of crossover. It also shows that tournament selection is degrading the search process, indicated by 0% crossover performing worse than "Normal" Cartesian Genetic Programming[28].

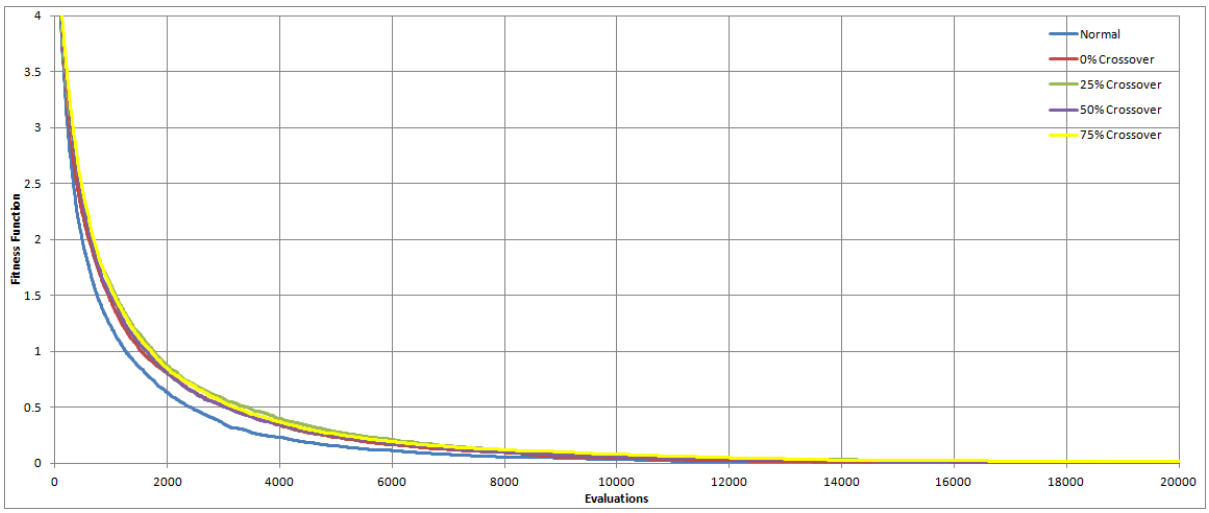

Figure 39 Various levels of crossover applied to the evolution of a Full Adder using optimised parameters with a tournament size of four

The second results shown in Figure 40 are for the same experiment previously described; now with the tournament size set to twenty. Figure 40 shows the same result present in the previous experiment, this time however the plots are slightly more spread out and it is clear that the search process is becoming worse as the crossover percentage is increased. This indicates that increasing the tournament size is not beneficial to the crossover technique for the full adder example.

[28] As mentioned previously, the only difference between "Normal" and "0% crossover", is the presence of tournament selection and the use of the floating point representation.

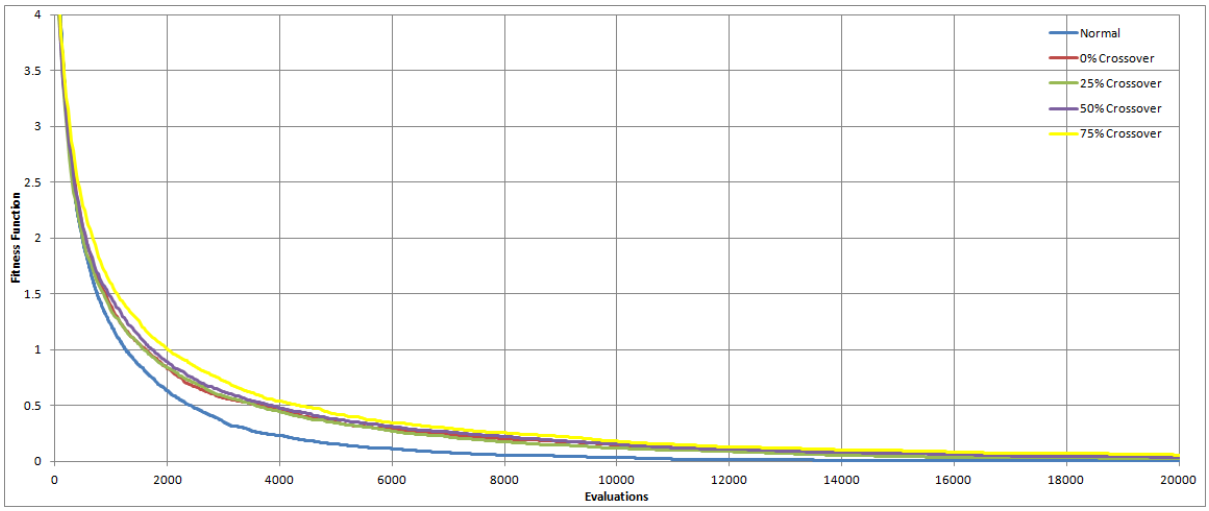

Figure 40 Various levels of crossover applied to the evolution of a Full Adder using optimised parameters with a tournament size of twenty

Table 19 shows the average evaluations and computational effort statistics for the full adder experiments implemented with a tournament size of four. The first thing to note is that for some of the experiments no computational effort figure is given. This is because for all of the 1000 runs used to generate the statistics, no single run failed to find a solution and so the computational effort equation was not valid. Experiments with fewer evaluations (generations) could have been conducted to increase the likelihood of not finding a solution on every run; but this was not considered necessary as the graphical figures and the average evaluations were considered sufficient to analyse the results. The average evaluations confirm that the presence of BLX-0 crossover is not beneficial to the search process in this case; as seen in Figure 39. The average evaluations also show that there is little difference between using the integer/floating point form for the chromosomes; indicating that this is not influencing the search process.

Table 19 Statistics used to analyse the Full Adder Problem - tournament size four

Experiment Description	Average Evaluations	Computational Effort
Normal CGP - integer representation	2,953	-
Normal CGP - floating point representation	2,994	-
0% Crossover	3,674	33,333
25% Crossover	4,137	-
50% Crossover	3,808	-
75% Crossover	4,111	33,333

Table 20 shows the statistics for the same experiment, this time using a tournament size of twenty. The same result as seen in the previous table is present; normal Cartesian Genetic Programming out performing that which uses BLX-0 crossover. The average evaluations statistic seen in Table 20, also shows the trend seen in Figure 40; increasing the crossover percentage decreases the effectiveness of the search process.

Table 20 Statistics used to analyse the Full Adder Problem - tournament size twenty

Experiment Description	Average Evaluations	Computational Effort
Normal CGP - integer representation	2,953	-
Normal CGP - floating point representation	2,994	-
0% Crossover	5,414	41,702
25% Crossover	4,944	37,051
50% Crossover	5,613	39,637
75% Crossover	6,310	43,459

Table 21 shows the parameters which were found to be the most suitable for the full adder problem case experiments; when using a tournament size of four. For all cases, the best mu value was found to be one. Low lambda values were also found to produce the best results, all in the range of three to five. In many cases, where crossover was been employed, the best population size (mu + lambda) was found to be the same as the tournament size. This has the effect of functioning as if there were no tournament selection scheme been employed (the two parents used to generate the children are always the best two chromosomes in the population). This could indicate that tournament selection is not benefiting the search process. The mutation percentages were all in the range of 14% - 20% and it appears that mutation percentage is inversely proportional to crossover percentage in this case.

Table 21 Best Parameters found for the Full Adder problem - tournament size four

Experiment Description	Mu	Lambda	Mutation
Normal CGP - integer representation	1	4	20%
Normal CGP - floating point representation	1	4	20%
0% Crossover	1	3	20%
25% Crossover	1	5	17%
50% Crossover	1	3	14%
75% Crossover	1	3	14%

Table 22 shows the best parameters found for the same experiment when using a tournament size of twenty. As when the tournament size was four, mu parameters of one were found to produce the best results. Very low lambda values were also again found to produce good results, indicating that the presence of a tournament selection scheme may be hindering the search process. The range of mutation percentages was this time between 14% - 26% and the pattern of mutation percentage being inversely proportional to crossover percentage was not present in these parameters.

Interestingly, when Julian Miller synthesises Boolean logic circuits using "normal" Cartesian Genetic Programming, he often uses the parameters mu=1 and lambda=4[29], which were found to be the most suitable during the extensive parameter optimisation process.

Table 22 Parameters found for the Full Adder problem - tournament size twenty

Experiment Description	Mu	Lambda	Mutation
Normal CGP - integer representation	1	4	20%
Normal CGP - floating point representation	1	4	20%
0% Crossover	1	22	14%
25% Crossover	1	19	23%
50% Crossover	1	21	23%
75% Crossover	1	22	26%

[29] As explained during a four year taught lecture course “Biologically Inspired Computation”, University of York, 2011.

Finally for the full adder experiments, Figure 41 confirms the result seen in Table 19 (and Table 20), that the floating point chromosome representation is not affecting the search process.

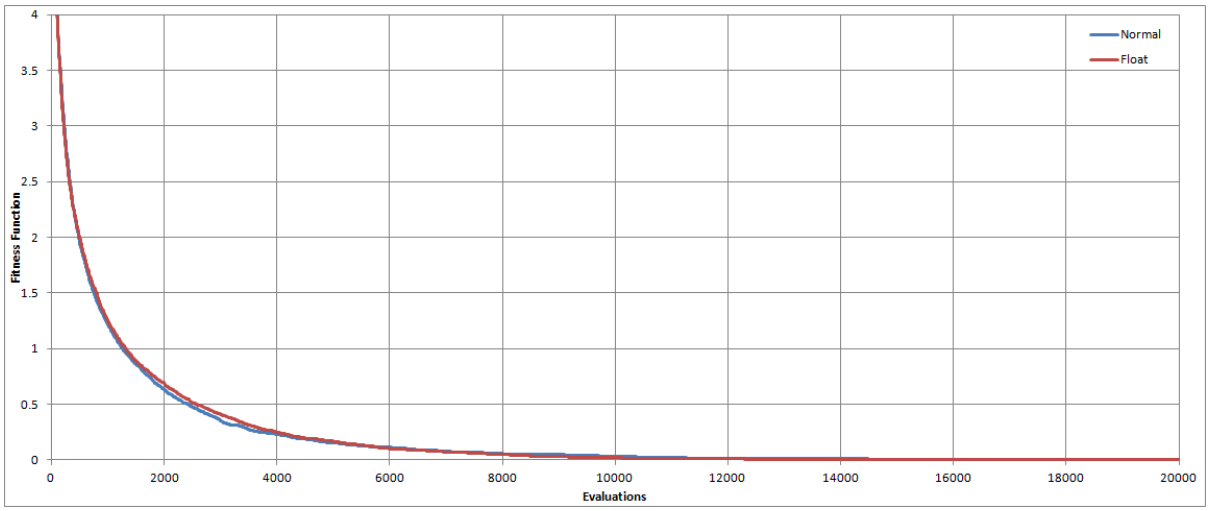

Figure 41 Normal Cartesian Genetic Programming applied to evolving Full Adder using conventional integer chromosome representation and floating point representation

Figure 42 shows the results of the first experiment into the evolution of a four bit even parity generator; using a tournament size of four. The first thing to note is the "Evaluations" scale is now much larger than before; this is because this experiment is significantly harder to solve than the full adder example. As a result, the parameter optimisation process and conducting the final experiments took significantly longer to complete than for the full adder[30].

As with the full adder example, Figure 42 shows that "Normal" Cartesian Genetic Programming is out performing that which uses BLX-0 crossover across all crossover percentages. Figure 42 also shows that "Normal" Cartesian Genetic Programming is out performing "0% Crossover", indicating that the presence of the tournament selection scheme is not beneficial to the search process. The plots 0%, 25%, 50% and 75% crossover are all very close, so no real conclusions can be drawn about their relative effectiveness; it does appear however that 75% crossover is the least effective at this search.

[30] The full adder circuit took approximately 20 minutes to conduct 1000 runs, whereas the four bit even parity circuit took approximately six hours.

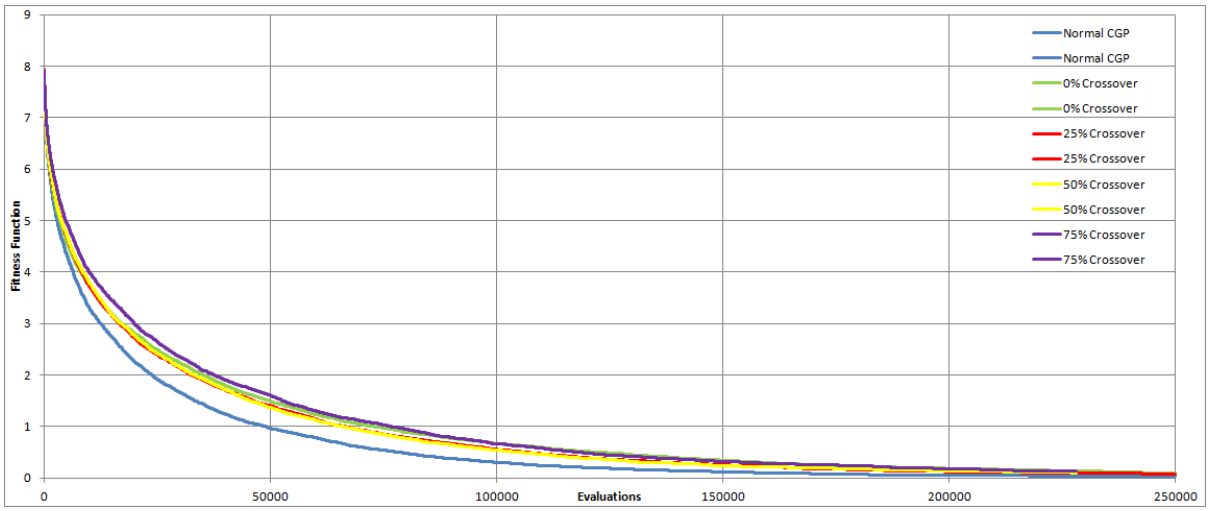

Figure 42 Various levels of crossover applied to the evolution of a four bit even parity generator using optimised parameters with a tournament size of four

When the tournament size is increased to twenty, the same results as seen in Figure 42 are even more apparent in Figure 43; showing that for this example BLX-0 crossover is operating more effectively with a lower tournament size. For this example however it appears that 75% crossover is outperforming the other strengths of crossover.

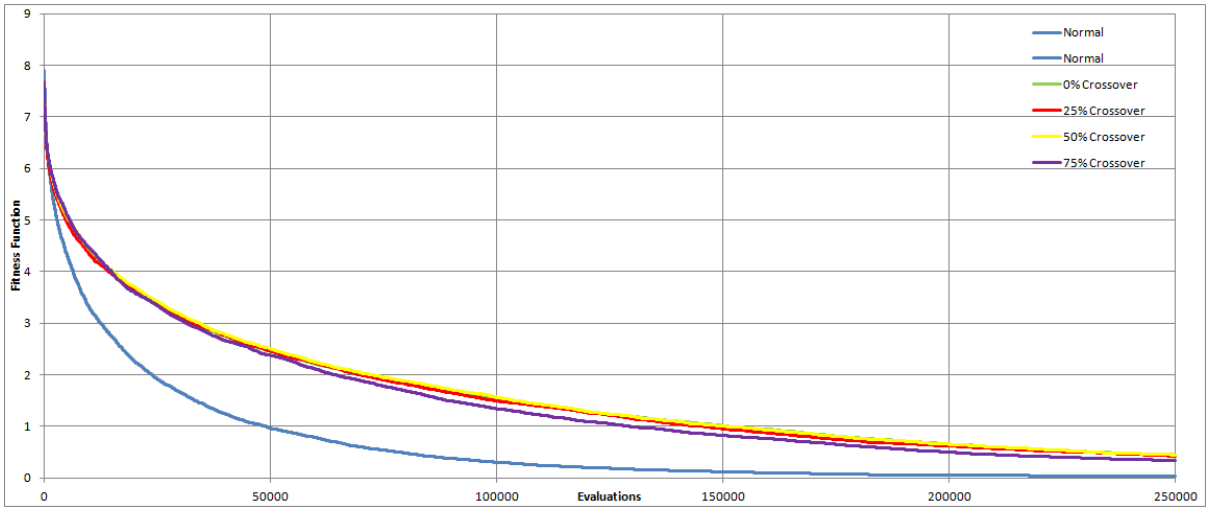

Figure 43 Various levels of crossover applied to the evolution of a four bit even parity generator using optimised parameters with a tournament size of twenty

The statistics used to analyse the effectiveness of the different search techniques for the four bit even parity generator, using a tournament size of four, are given in Table 23. Both the average evaluations and the computational efforts produced for this experiment show that “Normal” Cartesian Genetic Programming outperformed all levels of BLX-0 crossover. It

can also be seen that there appears to be no correlation between crossover percentage and the effectiveness of the search for this example.

Table 23 Statistics used to analyse the four bit parity generator Problem - tournament size four

Experiment Description	Average Evaluations	Computational Effort
Normal CGP - integer representation	71,297	315,452
Normal CGP - floating point representation	77,049	357,669
0% Crossover	105,920	486,916
25% Crossover	93,168	428,269
50% Crossover	93,331	435,259
75% Crossover	102,284	462,568

Table 24 show the same statistics for the four bit even parity generator using a tournament size of twenty. Once again the same result of "Normal" Cartesian Genetic Programming outperforming all levels of BLX-0 crossover is present. There also again appears to be no correlation between crossover percentages and the effectiveness of the search.

Just out of interest, it can be seen how much harder the four bit even parity problem is to solve than the full adder, by comparing the average number of evaluations needed by "Normal" Cartesian Genetic Programming in Table 19 and Table 23. It took approximately 25 times more evaluations to implement the four bit even parity generator than for the full adder. It is likely the difference in complexity is due to the presence of XOR logic gate in the full adder case (which was not available to the four bit even parity generator).

Table 24 6 Statistics used to analyse the four bit parity generator Problem - tournament size twenty

Experiment Description	Average Evaluations	Computational Effort
Normal CGP - integer representation	71,297	315,452
Normal CGP - floating point representation	77,049	357,669
0% Crossover	158,046	937,853
25% Crossover	151,747	879,297
50% Crossover	162,094	991,167
75% Crossover	146,178	820,931

The parameters found to be the most suitable for the four bit even parity generator, when using a tournament size of four, are given in Table 25. As for the full adder example, the best parameters found for "Normal" Cartesian Genetic Programming, were when mu=1 and lambda=4; the parameters found by Julian Miller to be the most suitable when evolving Boolean circuits. Again as for the full adder example, when using BLX-0 crossover the population size was always found to approach the tournament size. As previously mentioned this has the effect of making the tournament selection process redundant; always producing the children from the two fittest members of the population. The mutation percentages were in the range 8% - 17% and appeared to be inversely proportional to crossover percentage.

Table 25 Best Parameters found for the four bit parity generator Problem - tournament size four

Experiment Description	Mu	Lambda	Mutation
Normal CGP - integer representation	1	4	14%
Normal CGP - floating point representation	1	4	14%
0% Crossover	1	3	17%
25% Crossover	1	3	11%
50% Crossover	1	3	11%
75% Crossover	1	3	8%

Table 26 gives the parameters which were found to be most suitable for the four bit even parity generator when using a tournament size of twenty. Once again the population sizes were always very close to the tournament size (when crossover was been employed), indicating that the tournament selection process was not beneficial to the search. The range of mutation percentages was 14% - 23% and showed little correlation with crossover percentage.

Table 26 Best Parameters found for the four bit parity generator Problem - tournament size twenty

Experiment Description	Mu	Lambda	Mutation
Normal CGP - integer representation	1	4	14%
Normal CGP - floating point representation	1	4	14%
0% Crossover	1	20	20%
25% Crossover	1	19	17%
50% Crossover	1	22	23%
75% Crossover	1	19	23%

Once again Figure 44 shows that the use of the floating point chromosome representation has no effect on the search process for the evolution of a four bit even parity circuit. The values in Table 23 (and Table 24) also indicate this result, although there is a slightly larger difference between the average evaluations and computational effort than seen in previous examples.

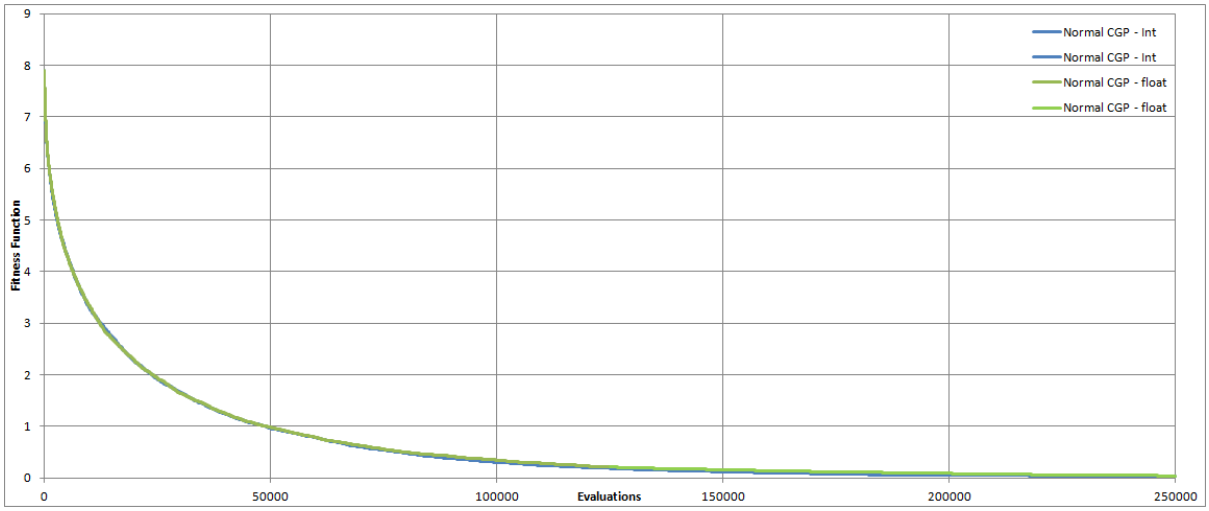

Figure 44 Normal Cartesian Genetic Programming applied to evolving a four bit parity generator using conventional integer chromosome representation and floating point representation

15.4 Conclusion

The main conclusion to be drawn is that BLX-0 crossover appears to be significantly degrading the effectiveness of the Cartesian Genetic Program when applied to the synthesis of Boolean logic. This result was shown by all three analytical methods (graphically, average evaluation and computational effort) for both of the test circuits investigated within this chapter. There was no notable correlation of the relative levels of crossover percentage offering an advantage or disadvantage for either of the two test circuits used in this chapter.

It appears that the implementation of a tournament selection scheme significantly reduces the effectiveness of the Cartesian Genetic Program. This has been shown by comparing the "Normal" Cartesian Genetic Programming results with those obtained using 0% crossover. The fact that many of the experiments found population sizes similar to the tournament size to be most effective, also indicates that the presence of the tournament selection scheme may be detrimental.

The affect of using different tournament sizes when implementing BLX-0 crossover can also be assessed from the results obtained in this chapter. The results show that for both circuits, using a tournament size of four resulted in a better search process than using a tournament size of twenty. This result was shown in the graphical plots of average fitness at each generation against evaluation, the average evaluations and in the computational effort required by each search. This result indicates that BLX-0 crossover would not perform better if larger tournament sizes were used.

All of the experiments described in this chapter found mu values of one to be most suitable in all cases. Additionally both experiments found lambda values of four to be the most suitable when not implementing crossover; in line with results obtained by Julian Miller[31]. This strongly indicates that a (1+4)-ES may be most suitable for the evolution of Boolean logic expressions when using Cartesian Genetic Programming (without crossover).

The effect of the floating point chromosome representation, required by BLX-0 crossover, has once again been shown not to affect the search process. The highest indication that this was not the case can be seen for the four bit even parity generator, Table 23 (or Table 24), which showed an 11.8% increase in computational effort required when using the floating point representation. However this increase was not seen in any of the graphical plots analysing the effect of the floating point form (Figure 41 and Figure 44) or in the average evaluations[32] recorded the full adder example (Table 19 and Table 20); which showed a difference of 1.4%.

[31] Information obtained during a four year lecture course "Biologically inspired Computation" taught by Julian Miller, 2011, University of York.
[32] Computational effort was not available.

16 Test Case 3: Function Optimisation

Function optimisation was chosen as the next test case for the main reason given in the Possible Test Cases chapter; nearly all problems can be reduced to the process of optimising parameters. The results of this test case are therefore highly important because if crossover can be shown to offer an advantage to generic function optimisation problems, it would make it beneficial to many real world applications of Cartesian Genetic Programming. For details of the specific functions to be optimised throughout this test case see the Possible Test Cases chapter.

16.1 The Experiments

All three multi-dimensional graphs described in the Possible Test Cases chapter (The Shekel Function, The Griewank Function and the Rosenbrock Function) were investigated so strong conclusions could be drawn over the effectiveness of BLX-0 crossover. An overview of the three functions is provided in Table 27 for reference. These graphs contain a range of different complexities/design spaces, including: many variables (Griewank), many local minima (Griewank & Shekel) and very flat landscapes (Rosenbrock).

Table 27 Overview of the three Graphical Functions used in this chapter

Function Name	Variables	Variable Range	Minimum	Optimal Co-coordinates
Shekel	4	$0 \leq Xi \leq 10$	-10.5364098167	4.00075, 4.00059, 3.99966, 3.99951
Griewank	10	$-600 \leq Xi \leq 600$	0	0,0,0,0,0,0,0,0,0,0
Rosenbrock	2	$-2 \leq Xi \leq 2$	0	1,1

Each graph is investigated using: Normal Cartesian Genetic Programming with and without the floating point chromosome representation and using the BLX-0 crossover at strengths of 0%, 25%, 50% and 75%. The crossover experiments were undertaken using tournament sizes of four and twenty. These experiments evaluate the effectiveness of the BLX-0 crossover in comparison to normal Cartesian Genetic Programming implemented without crossover. They also investigate the effect of varying the tournament size and crossover percentage when using BLX-0 crossover. Finally the floating point chromosome representation, required by the BLX-0 crossover, is also evaluated.

In all cases the parameters used for each experiment were found using the method described in Appendix B. As previously discussed, the process of finding suitable parameters requires substantial time investigating different evolutionary parameter values. The process is considered necessary as selecting semi-random values for the parameters would not be a fair comparison of the different search techniques; which may require different parameters to operate effectively.

Before the experiments were undertaken, the author felt concerned that the Cartesian Genetic Program would be able to prematurely solve the Griewank function due to its minimum been located at 0,0,0,0,0,0,0,0,0,0. This was because one of the inputs made available to the Cartesian Genetic Program was zero; and could simply be mapped to all of the outputs to solve the Griewank problem. As a result the Griewank function was slightly altered so the minimum was shifted to the arbitrary position of 0.27583, 0.27583, 0.27583, 0.27583, 0.27583, 0.27583, 0.27583, 0.27583, 0.27583, and 0.27583. The Shekel Function was also raised, so as the minimum produced a fitness of zero instead of -10.5364098167[33].

All of the experiments used throughout this chapter were averaged over 1000 runs to provided statistically reliable data. The results are presented in the usual formats: a graphical plot of fitness against evaluation, average evaluations and computational effort.

16.2 Design

To make the Cartesian Genetic Program generate co-ordinates for the given functions, the number of inputs was set to five and fixed arbitrarily as: 0, 0.1, 0.2, 0.3, and 0.4. The co-ordinates for the function under consideration were then the corresponding chromosome outputs to these inputs. To accommodate the different ranges of each functions parameter, the Cartesian Genetic Programs function nodes were chosen to only produce values between minus and positive one. This ensured that the outputs generated by each chromosome were also between minus and positive one and could then be scaled (and if necessary shifted) to a suitable range for each function.

[33] This was undertaken so the graphs of average fitness against evaluation would be consistent with all the other graphs produced during this project; with zero representing a perfect solution.

The functions provided for the function nodes were as follows[34]: absolute(input 1), square root of the absolute(input 1), sin(input 1), cos(input 1), tanh(input 1), sin(input 1 + input 2), cos(input 1 + input 2), tanh(input 1 + input 2), hypotenuse(input 1 , input 2)/√2, (input 1 + input 2)/2, (input 1 - input 2)/2, input 1 * input 2, bounded division (where the largest input value is always the denominator).

The fitness assigned to each chromosome is the value returned by the function under inspection; at the co-ordinates produced by the scaled outputs of each chromosome. For all the functions investigated, lower fitness values represented a fitter chromosome; with zero always representing a perfect solution. A solution was considered suitable, and hence the search terminated, when the assigned fitness was < 0.001.

To ensure correct fitnesses were being assigned to each chromosome, an excel spread sheet was constructed which decoded a given chromosome and produced its outputs; for the given fixed inputs. These outputs were then scaled and/or shifted appropriately and input into the corresponding function. The output of this function was then used to ensure the fitness assigned to the chromosome under inspection was correct; confirming the correct operation of the fitness function within the code.

16.3 Results

The first set of results surround the Rosenbrock Function, implemented with a tournament size of four and twenty (when BLX-0 crossover is being employed). Figure 45 shows the fitness at each evaluation, averaged over 1000 experiments, for normal Cartesian Genetic Programming and that implemented with 0%, 25%, 50% and 75% crossover; using a tournament size of four. The results shown in Figure 45 differ to those seen previously in this project as it shows a wide range of fitnesses in the initial population. This is an interesting result as the randomly generated initial populations should produce random fitness values, which when averaged over the 1000 runs should be the same regardless of the presence or strength of crossover. The likely explanation for this range of initial fitnesses is that larger random populations are more likely to contain a fitter chromosome than those of a smaller size. This appears to be the case in Figure 45, as the order of initial fitnesses are

[34] These functions were taken from code provided by Julian Miller for a fourth year lecture course "Biologically Inspired Computation", Electronic Engineering, University of York, 2011.

directly ordered with the population sizes used; as can be seen from Table 30, which shows the population sizes[35] used for this experiment.

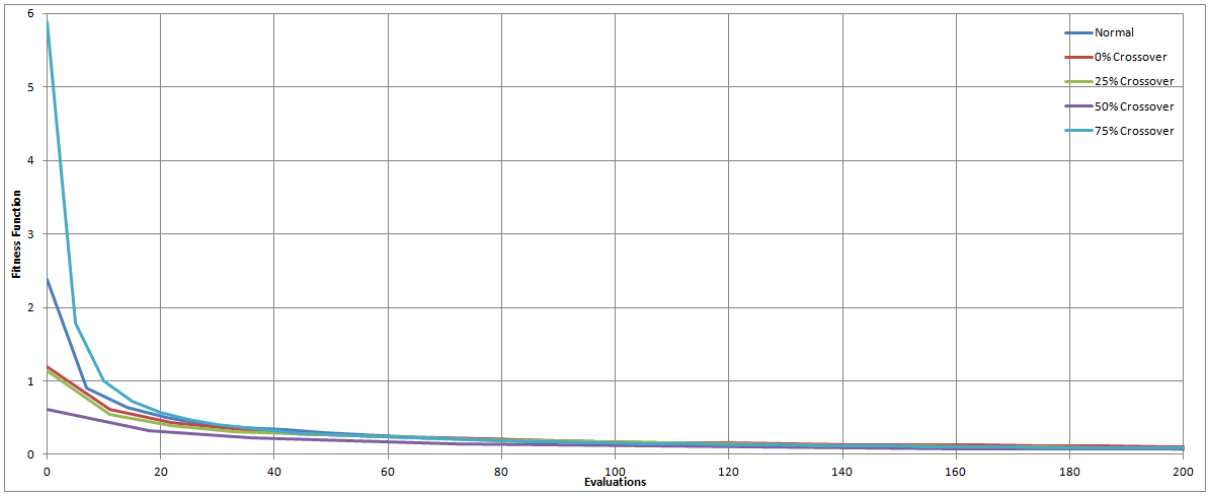

Figure 45 Various levels of crossover applied to finding the minimum value for the Rosenbrock Function with a tournament size of four

The true effectiveness of different levels of crossover cannot be assessed by inspecting Figure 45, as it is not clear whether the different plots are performing better than others due to the initial population size, or because they represent a more effective search process.

Figure 46 shows the results of the same experiment as seen in Figure 45, now implemented with a tournament size of twenty. Again it is not possible to determine the relative effectiveness of the different search methods, as the difference in population size appears to affect the results more dominantly than the differences in the search strategies.

It may be the case, that if this were a more challenging problem to solve, the plots would not converge on a solution so quickly and the relative effectiveness of the search techniques could have been identified.

[35] Population size is the sum of mu and lambda.

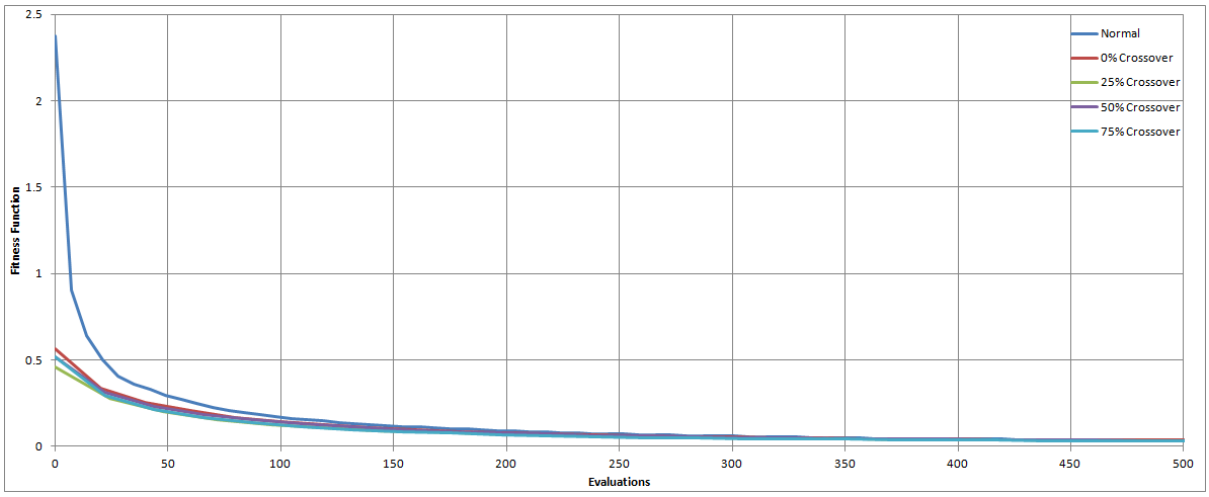

Figure 46 Various levels of crossover applied to finding the minimum value for the Rosenbrock Function with a tournament size of twenty

Table 28 and Table 29 show the average evaluations and the computational effort required to solve the Rosenbrock function; using tournament sizes of four and twenty respectively. It can be seen from these results that normal Cartesian Genetic Programming, implemented without crossover, produced the most effective search when compared to BLX-0 crossover using a tournament size of four; but only by a small margin. When BLX-0 crossover was implemented using a tournament size of twenty, the most effective search was produced using a crossover percentage of 0%; this result was not mirrored by the computational effort. 0% crossover does not implement the BLX-0 crossover, but is distinct from normal Cartesian Genetic Programming as it uses the floating point form for the chromosomes and a tournament selection scheme. In both cases (for tournament sizes four and twenty) there appears to be no strong correlation between crossover percentage and the average evaluations required to find a solution (or computational effort).

When using a tournament size of four, Table 28 shows little difference between "Normal" and "0% Crossover", indicating that the presence of a tournament selection scheme is not influencing the search process of the Cartesian Genetic Program. This result was also mirrored in Table 29, now using a tournament size of twenty, as the average evaluations showed tournament selection to be beneficial and computational effort showed the opposite.

Table 28 Statistics used to analyse the Rosenbrock Function Problem - tournament size four

Experiment Description	Average Evaluations	Computational Effort
Normal CGP - integer representation	2,610	20,000
Normal CGP - floating point representation	2,782	23,239
0% Crossover	2,679	20,423
25% Crossover	3,471	22,926
50% Crossover	2,827	20,000
75% Crossover	3,703	24,968

Table 29 Statistics used to analyse the Rosenbrock Function Problem - tournament size twenty

Experiment Description	Average Evaluations	Computational Effort
Normal CGP - integer representation	2,610	20,000
Normal CGP - floating point representation	2,782	23,239
0% Crossover	2,399	20,824
25% Crossover	2,492	22,926
50% Crossover	2,958	22,605
75% Crossover	2,770	22,605

The parameters found to be most suitable for the Rosenbrock function are given in Table 30 and Table 31; for crossover implemented with tournament sizes of four and twenty respectively. For the majority of the experiments, mu values of one were found to produce the best results. Values for the lambda parameters varied in a seemingly random manor; with no clear pattern or correlation to crossover percentage. The population sizes found to be most suitable, when using a tournament size of four, were consistently above the tournament size, except for the highest crossover percentage of 75%. This result was not seen when the tournament size was increased to twenty; with the population sizes now much closer to the tournament size. As previously mentioned, population sizes at, or near the tournament size, have the affect of removing the effect of the tournament selection process.

The mutation rates found to produce the best results for this optimisation problem were the highest used throughout this project. As previously mentioned, the authors

implementation of mutation is slightly different from that usually employed, the authors percentage mutation refers to the number of nodes which are changed, not the number of parameters in the chromosome. Even when taking this difference into account, the mutation rates found to be most suitable for the Rosenbrock problem are still very high.

Table 30 Best Parameters found for the Rosenbrock Function Problem - tournament size four

Experiment Description	Mu	Lambda	Mutation
Normal CGP - integer representation	1	6	130%
Normal CGP - floating point representation	1	6	130%
0% Crossover	1	10	70%
25% Crossover	1	10	50%
50% Crossover	1	17	160%
75% Crossover	1	4	60%

Table 31 Best Parameters found for the Rosenbrock Function Problem - tournament size twenty

Experiment Description	Mu	Lambda	Mutation
Normal CGP - integer representation	1	6	130%
Normal CGP - floating point representation	1	6	130%
0% Crossover	1	19	110%
25% Crossover	1	23	140%
50% Crossover	3	19	130%
75% Crossover	2	20	170%

Finally for the analysis of the Rosenbrock function, Figure 47 shows graphically a comparison between the integer and the floating point form for the chromosomes. It can be seen that there appears to be almost no difference between the two representations. By inspecting Table 28 (or Table 29) it can be seen however, there is a slight difference and that the floating point form appears to be performing slightly worse than the integer counterpart.

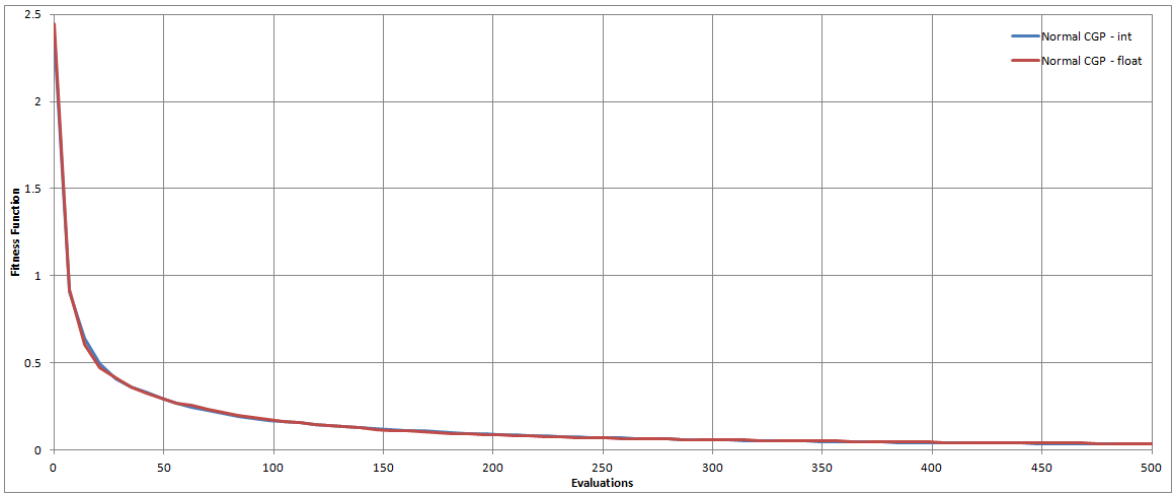

Figure 47 Normal Cartesian Genetic Programming applied to finding the minimum value for the Rosenbrock function using conventional integer chromosome representation and floating point representation

Figure 48 and Figure 49 show graphically the results of applying normal Cartesian Genetic Programming, and that implemented with 0%, 25%, 50% and 75% crossover, to the Griewank function; using tournament sizes of four and twenty respectively. In both cases normal Cartesian Genetic Programming outperformed all levels of crossover percentages; for both tournament sizes. As with the results from the Rosenbrock function, the differences in the fitnesses on the first evaluation appear to be directly related to population size; see Table 34 and Table 35. In this instance however, it is thought that this is not the only reason why normal Cartesian Genetic Programming is outperforming crossover implementations. As is shown in Figure 49, Cartesian Genetic Programming starts at a disadvantage and still manages to outperform the other strategies. The plots of different levels of crossover also show that increasing the crossover percentage decreases the effectiveness of the search for both tournament sizes investigated; four and twenty. It is clear from these two plots that BLX-0 crossover is operating more effectively when implemented with the larger tournament size; twenty.

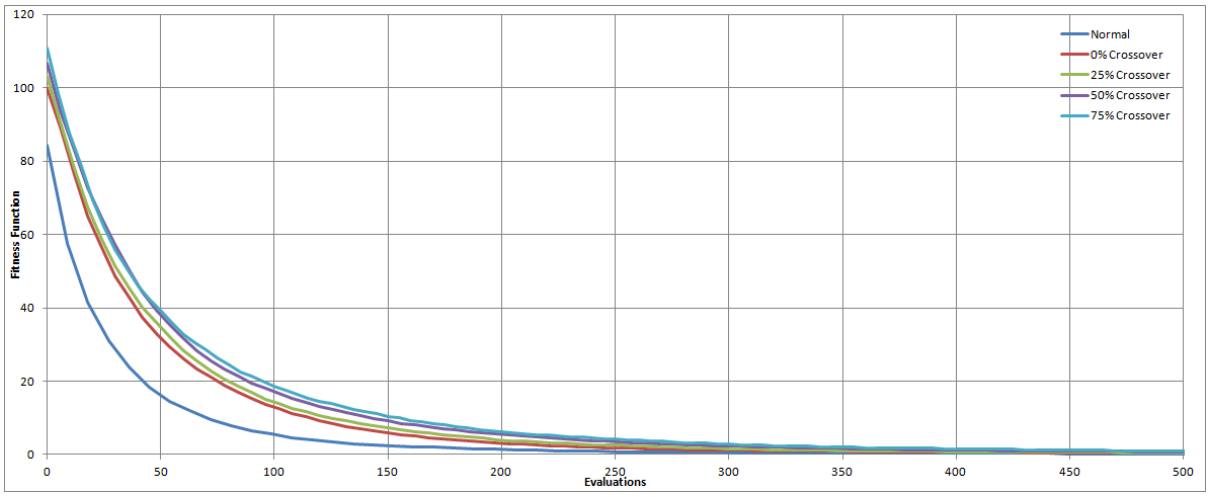

Figure 48 Various levels of crossover applied to finding the minimum value for the Griewank Function with a tournament size of four

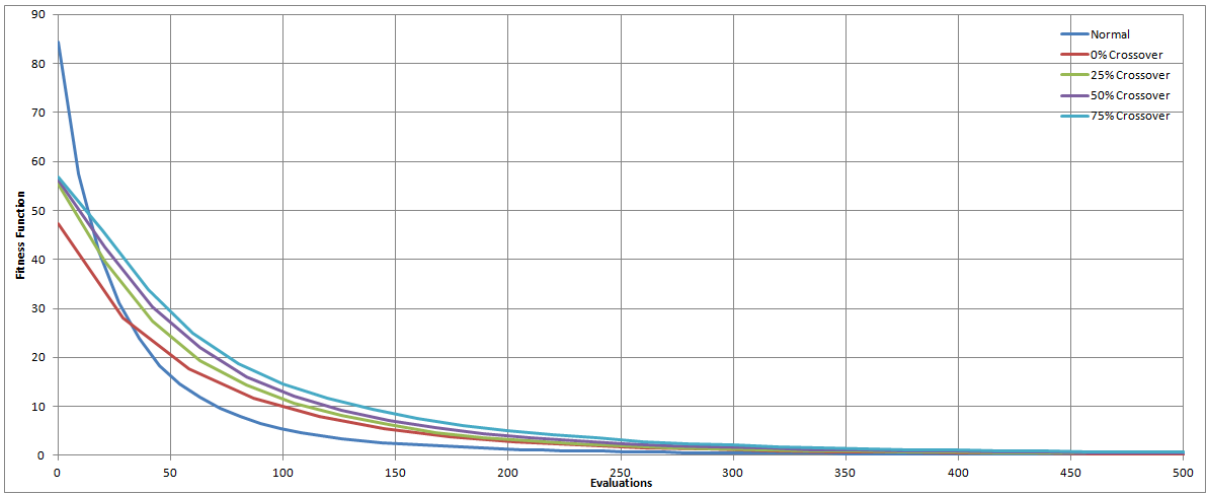

Figure 49 Various levels of crossover applied to finding the minimum value for the Griewank Function with a tournament size of twenty

The same result seen graphically in Figure 48 and Figure 49 is also present in Table 32 and Table 33; normal Cartesian Genetic Programming outperforming all levels of crossover using both tournament sizes investigated. This result is shown by both the average number of evaluations needed to find a solution and the computational effort. It can also be seen in Table 32 and Table 33 that increasing the crossover percentage decreases the effectiveness of the search for both tournament sizes investigated; again confirmed by both average evaluations and computational effort.

Table 32 and Table 33 also show, for both tournament sizes, that the presence of a tournament selection scheme is detrimental to the search process; as can be seen by comparing the "Normal" statistics to those calculated for "0% Crossover".

Table 32 Statistics used to analyse the Griewank Function Problem - tournament size four

Experiment Description	Average Evaluations	Computational Effort
Normal CGP - integer representation	10,320	64,350
Normal CGP - floating point representation	10,526	65,105
0% Crossover	11,849	89,176
25% Crossover	12,391	94,159
50% Crossover	13,040	101,899
75% Crossover	14,242	128,012

Table 33 Statistics used to analyse the Griewank Function Problem - tournament size twenty

Experiment Description	Average Evaluations	Computational Effort
Normal CGP - integer representation	10,320	64,350
Normal CGP - floating point representation	10,526	65,105
0% Crossover	11,249	79,293
25% Crossover	11,695	94,159
50% Crossover	12,467	109,734
75% Crossover	13,282	129,114

As with the Rosenbrock function, Table 34 and Table 35 show that for the Griewank function, mu values of one were found to be most suitable in nearly all cases. The lambda values found to produce the best results for the Griewank function were also similar to previous results; except for 0% crossover, tournament size twenty, which had a high value of 27. The population sizes also appeared to decrease with crossover percentage, and were never significantly greater than the tournament size; except for 0% crossover, tournament size twenty, previously mentioned. The mutation rates, in complete contrast to the Rosenbrock results, were all very low; 5% or 10%.

Table 34 Best Parameters found for the Griewank Function Problem - tournament size four

Experiment Description	Mu	Lambda	Mutation
Normal CGP - integer representation	1	8	10%
Normal CGP - floating point representation	1	8	10%
0% Crossover	1	5	5%
25% Crossover	1	5	5%
50% Crossover	1	5	5%
75% Crossover	1	4	5%

Table 35 Best Parameters found for the Griewank Function Problem - tournament size twenty

Experiment Description	Mu	Lambda	Mutation
Normal CGP - integer representation	1	8	10%
Normal CGP - floating point representation	1	8	10%
0% Crossover	2	27	10%
25% Crossover	1	20	5%
50% Crossover	1	20	5%
75% Crossover	1	19	5%

Finally for the Griewank function, Figure 50 shows graphically a comparison between Cartesian Genetic Programming implemented with integer and floating point chromosome representation. As can be seen from Figure 50, and from Table 32 (or Table 33), the floating point representation is not effecting the search process to any significant amount.

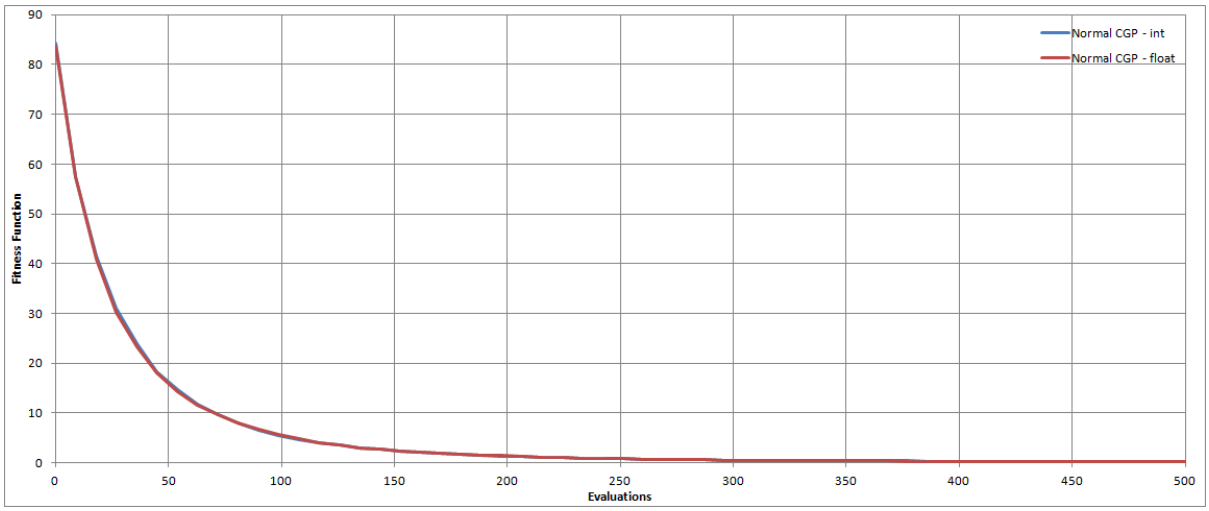

Figure 50 Normal Cartesian Genetic Programming applied to finding the minimum value for the Griewank function using conventional integer chromosome representation and floating point representation

The final function optimisation problem investigated in this chapter is the shekel function. The same graphical plot of average fitness against evaluations is shown in Figure 51 and Figure 52; for BLX-0 crossover implemented with tournament sizes of four and twenty respectively. From the two graphs described, it appears that a tournament size of twenty, rather than four, is more effective for this particular problem when using BLX-0 crossover. Figure 51 shows normal Cartesian Genetic Programming out performing all levels of crossover when using a tournament size of four. When using a tournament size of twenty however, the results appear similar, with different plots intercepting each other at various points; indicating that different levels of crossover are more effective at different stages of the search.

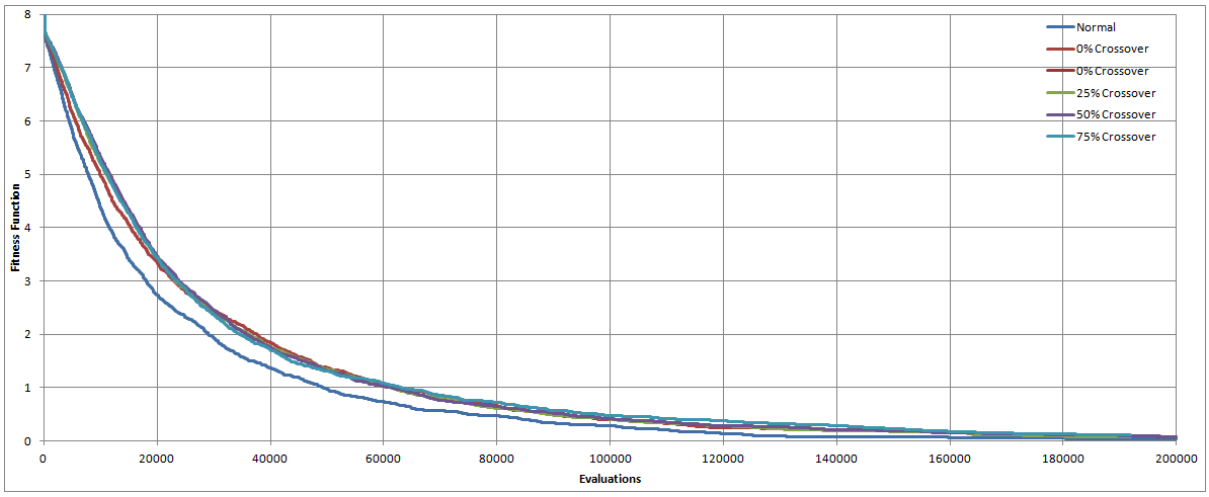

Figure 51 Various levels of crossover applied to finding the minimum value for the Shekel Function with a tournament size of four

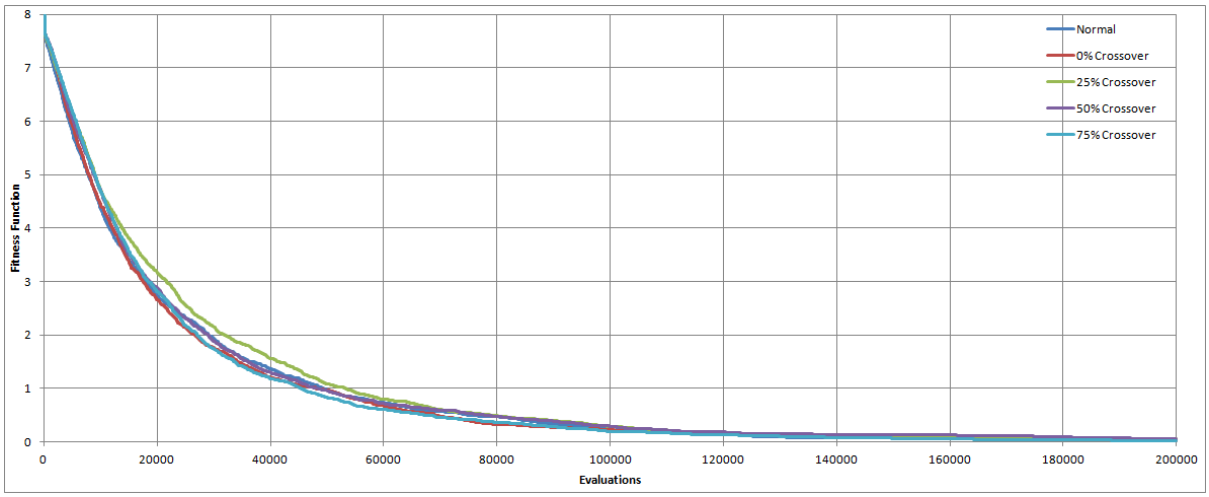

Figure 52 Various levels of crossover applied to finding the minimum value for the Shekel Function with a tournament size of twenty

The average number of evaluations required to find a solution and the computational effort are given for the Shekel problem case in Table 36 and Table 37; for tournament sizes four and twenty respectively. Strangely, for both tournament sizes four and twenty, the average evaluations and the computational effort statistics indicate that different strategies produced the best search results. When using a tournament size of four, Table 36 shows that normal Cartesian Genetic Programming produced the best search results according to the average evaluations, but the computational effort indicates that 0% crossover produced the best results. When using a tournament size of twenty, the average evaluations shows that 0% crossover produced the best search results whereas computational effort indicates 75% crossover. This confusion in the results is likely to be due to the fact that the

effectiveness of the different levels of crossover are similar; especially for when a tournament size of twenty is been employed by the crossover.

Table 36 Statistics used to analyse the Shekel Function Problem - tournament size four

Experiment Description	Average Evaluations	Computational Effort
Normal CGP - integer representation	34,428	252,362
Normal CGP - floating point representation	34,310	244,160
0% Crossover	41,456	262,660
25% Crossover	44,951	244,160
50% Crossover	43,363	279,369
75% Crossover	45,864	267,585

Table 37 Statistics used to analyse the Shekel Function Problem - tournament size twenty

Experiment Description	Average Evaluations	Computational Effort
Normal CGP - integer representation	34,428	252,362
Normal CGP - floating point representation	34,310	244,160
0% Crossover	33,463	249,679
25% Crossover	37,460	241,316
50% Crossover	36,801	257,592
75% Crossover	36,256	190,757

Table 36 and Table 37 can also be used to identify if the presence of a tournament selection scheme affects the search process of a normal Cartesian Genetic Program; by comparing the "Normal" statistics with those calculated for "0% Crossover". It can be seen that when using a tournament size of four, the presence of a tournament selection scheme appears to be detrimental to the search process. When using a tournament size of twenty however, the presence of a tournament selection scheme appears to be aiding the search process.

As with the previous two function optimisation problems, mu values of one were found to be the most suitable for the Shekel function for both tournament sizes four and twenty; see Table 38 and Table 39 respectively. A large range of lambda values were found to produce the best results, from the lower limit of the tournament size, to reasonably high values in

both cases. The mutation percentages found to produce the best search results were all in the range of 20% to 40%.

Table 38 Best Parameters found for the Shekel Function Problem - tournament size four

Experiment Description	Mu	Lambda	Mutation
Normal CGP - integer representation	1	8	40%
Normal CGP - floating point representation	1	8	40%
0% Crossover	1	3	30%
25% Crossover	1	5	30%
50% Crossover	1	6	20%
75% Crossover	1	5	20%

Table 39 Best Parameters found for the Shekel Function Problem - tournament size twenty

Experiment Description	Mu	Lambda	Mutation
Normal CGP - integer representation	1	8	40%
Normal CGP - floating point representation	1	8	40%
0% Crossover	1	19	40%
25% Crossover	1	25	30%
50% Crossover	1	26	30%
75% Crossover	1	20	40%

Finally for the Shekel function, Figure 53 shows graphically a comparison between the integer and floating point form of the chromosomes. Figure 53 clearly shows that the floating point representation is not affecting the search process; a result confirmed by the average evaluations and computational effort seen in Table 36 (or Table 37).

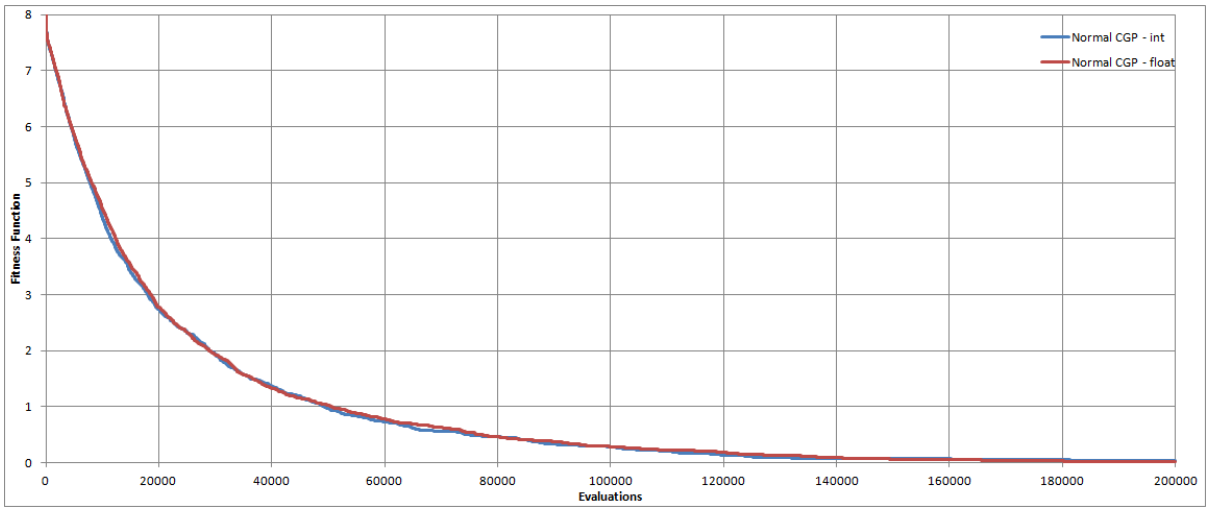

Figure 53 Normal Cartesian Genetic Programming applied to finding the minimum value for the Shekel function using conventional integer chromosome representation and floating point representation

16.4 Conclusion

It can be concluded that BLX-0 crossover is not offering a significant advantage to the search process. The only instances of normal Cartesian Genetic Programming not producing the best results was for the Shekel function; which was a tie between 0% and 75% crossover[36].

The presence of crossover produced similar results to Cartesian Genetic Programming implemented without crossover (the Rosenbrock and Shekel functions) or resulted in a significant disadvantage (the Griewank function). None of the function optimisation problems investigated showed BLX-0 crossover to produce significantly better results than normal Cartesian Genetic Programming; at best the results were similar. There also appeared to be no positive correlation between crossover percentage and search effectiveness, in fact for the Griewank function there appeared to be a negative correlation; whereby increasing the crossover percentage actually made the search worse. It is therefore concluded that BLX-0 crossover is not a beneficial addition to Cartesian Genetic Programming when applied to function optimisation type problems.

It is understood that only a specific form of function optimisation problem has been investigated in this chapter. For this reason it is not possible to speculate that BLX-0 crossover would be detrimental to all function optimisation type problems. It seems clear however for the simplistic smooth search spaces provided, that BLX-0 crossover is not offering an advantage.

[36] Where 0% crossover does not actually implement BLX-0 crossover; only a tournament selection scheme.

Interestingly, better results were always found with the higher tournament size of twenty, when using BLX-0 crossover, than the lower size of four. If time had permitted it would have been beneficial to see if this trend continued with increasing tournament size; possibly to a point which produced better results than normal Cartesian Genetic Programming. However due to time restraints this investigation is left for further work.

It is not possible to reach a conclusion over how the presence of a tournament selection scheme affects the search process of a Cartesian Genetic Program; for function optimisation type problems. This is due to tournament selection appearing to be ineffective for the Rosenbrock function, detrimental to the Griewank function and beneficial/detrimental to the Shekel function depending upon the tournament size used.

It can be concluded that the floating point chromosome representation, required by the BLX-0 crossover, is not affecting the search process to any significant amount. For the Rosenbrock function the floating point representation increased the average Evaluations by 6.18% (13.9% for the computational effort). The Griewank functions average evaluations was increased by 1.96% (1.16% for the computational effort. Finally the number of evaluations was reduced by 0.34% (3.25% for the computational effort) for the Shekel function when using the floating point representation.

The parameters found to produce the best search results were fairly erratic for the functions investigated; with the exception of mu values of one. There appears to be no obvious pattern with lambda values used or the mutation percentages; which ranged from 10% to 110%.

16.5 Thoughts

The author feels that although Cartesian Genetic Programming can be applied to function optimisation type problems, it is not as elegant as other applications investigated e.g. symbolic regression and synthesis of Boolean logic. It seems there would be no reason to use a Cartesian Genetic Program to generate the parameters under optimisation, rather than a simpler Genetic Algorithm. It does show however the adaptability and general purpose nature of Cartesian Genetic Programming. An interesting investigation would be a comparison between a Cartesian Genetic Program and a Genetic Algorithm over a range of function optimisation problems, to assess their relative effectiveness.

17 Test Case 4: Wall Avoider

The Wall Avoider is the final test case investigated during this project; it is also one of the most interesting, as described in the Possible Test Cases chapter. The Wall Avoider test case shares many characteristics with the synthesis of Boolean logic example; the distinction being that the Wall Avoider uses no specific truth table as the "target". Instead the fitness is determined by testing the suitability of the logic generated against a given task; navigating a unit across a "world" filled with obstructions. This has the affect of evolving a truth table which describes the logic to solve the given problem, whilst simultaneously evolving an implementation for the same truth table.

17.1 The Experiments

To evaluate the effectiveness of BLX-0 crossover the following strategies were compared: normal Cartesian Genetic Programming[37], that which uses the floating point chromosome representation and that which uses 0%, 25%, 50% and 75% BLX-0 crossover. When implementing BLX-0 crossover a tournament size of ten was used. A single tournament size was chosen due to time restraints not permitting this parameter to be varied; as seen in previous chapters. A value of ten was chosen, as opposed to four or twenty, as previous experiments had shown both high and low values being effective for different test cases, and ten was a value in the centre of this range. These experiments assess the effectiveness of the BLX-0 crossover using a range of crossover percentages, compared to normal Cartesian Genetic Programming. They also independently assess the effect of the floating point chromosome representation, and the presence of a tournament selection scheme.

As for all the previous test cases, the evolutionary parameters (mu, lambda and mutation percentage) used for each experiment are determined by the process described in Appendix B. This enables a fair comparison between the different strategies, as it cannot be assumed that the same parameters are the most suitable in each case.

Originally the "world" to be navigated was that shown in Figure 54; the blue and yellow squares represent the starting position and the finish line respectively. This layout was not

[37] Implemented without crossover, without tournament selection and using the usual integer chromosome representation.

selected however as it transpired during the initial testing phase, that the solution of following the wall to the left (initially moving downwards) represented a perfect fitness and was quickly converged upon[38]. Instead the "world" seen in Figure 55 was employed; with the starting position now represented by a green square. This is a much more complex "world" and contains the following challenging structures (from right to left): a continuous wall to avoid the solution of wall following, a vertical slalom to test simple wall avoiding, a horizontal slalom which represents a scenario whereby moving away from the finish line is overall beneficial, two narrow paths with the upper representing a shorter route and finally a structure where the direction of motion must be changed each move in order to avoid a collision. It is thought that this series of challenges creates a complex search space suitable for this final test case.

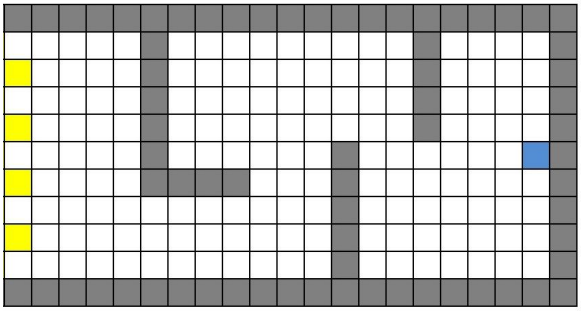

Figure 54 Old "World" for the wall avoider problem case

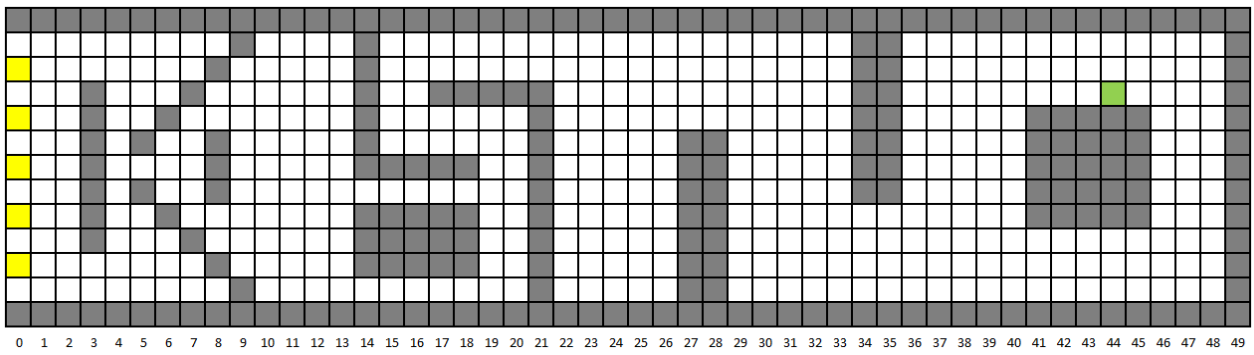

Figure 55 New "World" for the wall avoider problem case

17.2 Design

For this test case the inputs available to the Cartesian Genetic Program are ten binary values. The first eight values represent whether the blocks surrounding the unit are "free space", or "wall"; represented by zero and one respectively. These first eight inputs are

[38] The author intended for a more challenging search space to be used for this final test case.

intended to represent the ability to "see" the surrounding "world", as shown in Figure 56. The remaining two inputs are the outputs of the previous move, this provides a simple "memory" for the unit; the initial "memory" will be that of moving forwards.

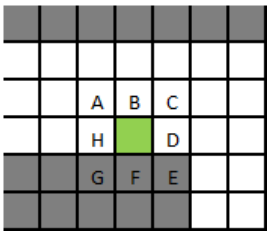

Figure 56 Depiction of "Sight"

There are two outputs from this Cartesian Genetic Program, which are again binary values. These outputs are decoded into movements with the following mapping: 00 - move up, 01 - move down, 10 - move left, 11 - move right.

The evolved solutions are then assigned fitnesses determined by how successfully they "navigated" across the "world". This fitness is calculated as the distance from the finish line (in horizontal squares) after: 100 sense-act loops have passed, a wall has been struck or the finish line has been reached. If a wall is struck, the fitness is taken from the last "alive" position; striking a wall is considered to cause the navigating unit to "die". The fitness is taken from the last "living" position, as it was noticed during testing that the units had "suicidal tendencies", favouring being closer to the finish line over "life"; which was not the intention. Using this system, lower fitness values represent fitter chromosomes with zero representing a perfect solution.

The function nodes made available to the chromosomes are the Boolean expressions: AND, OR, NAND, NOR and XOR; as used by Test Case 2: Synthesis of Boolean Logic. The number of available function nodes was set to twenty; this was an educated guess at a suitable number of function nodes which proved effective during initial testing.

To ensure that the correct fitnesses were assigned to each chromosome, an elaborate testing procedure was undertaken. First the chromosomes were decoded into a truth table using Microsoft's excel; as described in Test Case 2: Synthesis of Boolean Logic. These truth tables were then implemented in Java, along with a model of the "world". The units were then tested within this "world"; with the current position displayed graphically along with

the distance from the finish line (the fitness). These generated animations were interesting to observe and provided a good testing strategy. A selection of the generated animations is available to the reader in Appendix D.

17.3 Results

The first set of results presented in this chapter is a comparison between normal Cartesian Genetic programming and that which employs 0%, 25%, 50% and 75% BLX-0 crossover. The results of this first experiment are given graphically in Figure 57, where it can be seen that normal Cartesian Genetic Programming is out performing all levels of BLX-0 crossover. As seen in previous chapters, the plots for the different levels of crossover percentage intersect at various points, indicating that different crossover strengths may be more beneficial at various stages of the search.

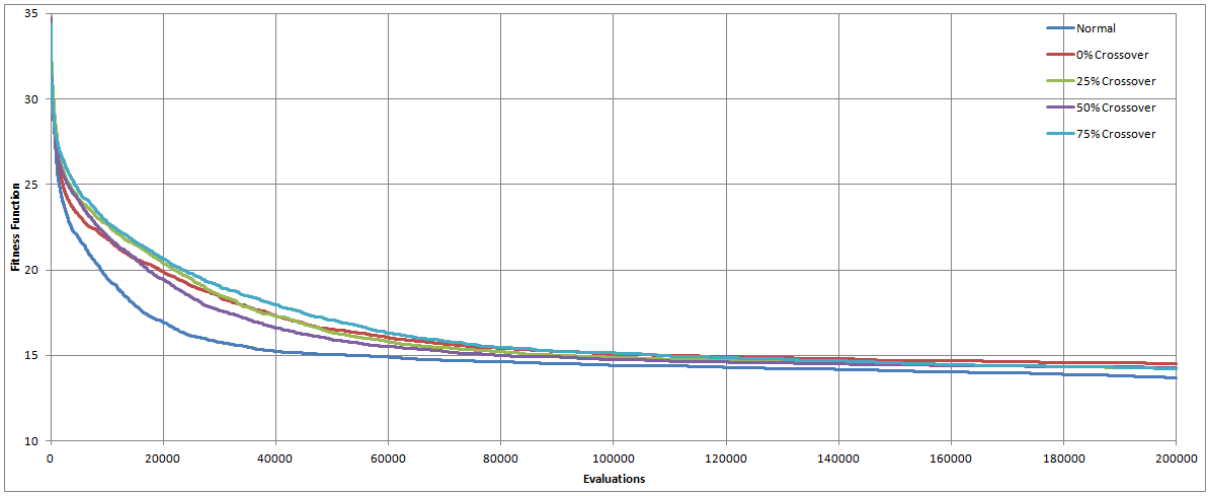

Figure 57 Various levels of crossover applied to the Wall Avoider problem case with a tournament size of ten

Table 40 gives the average evaluations, computational effort and the average final fitness for the same experiment previously described. The average final fitness was included as an additional statistic, due to the fact that the majority of the runs failed to reach a solution causing the average evaluations to become meaningless[39]. The experiments could have been ran for more evaluations, thus creating more meaningful average evaluation statistics; but as these experiments were already taking substantial time to undertake, this was not done.

[39] The average evaluations statistic assumes the solution was found on the final evaluation if no solution was found.

All three statistics given in Table 40 indicate that normal Cartesian Genetic Programming, implemented without BLX-0 crossover, produced a better search than all levels of crossover. It also appears that there is no correlation between the effectiveness of the search and crossover percentage.

Table 40 Statistics used to analyse the Wall Avoider problem case - tournament size ten

Experiment Description	Average Evaluations	Computational Effort	Average Final Fitness
Normal CGP - integer representation	197,106	22,562,201	13.705
Normal CGP - floating point representation	197,235	29,247,841	13.762
0% Crossover	199,506	153,044,694	14.52
25% Crossover	199,092	83,269,001	14.248
50% Crossover	199,274	153,044,694	14.284
75% Crossover	199,995	153,044,694	14.23

Table 41 shows the parameters which were found to be most suitable for the different search strategies investigated in this chapter. It can be seen that for all of the strategies, mu values of one or two were found to produce the best results and the lambda values were all in the range of 10 - 13. This resulted in all the population sizes only being slightly higher than the tournament size. The mutation rates were all in the range of 15% - 25%, with no apparent correlation with crossover percentage.

Table 41 Best Parameters found for the Wall Avoider problem case - tournament size ten

Experiment Description	Mu	Lambda	Mutation
Normal CGP - integer representation	1	11	25%
Normal CGP - floating point representation	1	11	25%
0% Crossover	2	12	20%
25% Crossover	1	10	15%
50% Crossover	2	10	25%
75% Crossover	2	13	20%

Finally Figure 58 gives graphically a comparison between the integer and floating point chromosome representation. It can be seen that the use of the floating point form is not affecting the search process; this result is also shown in the statistics given in Table 40.

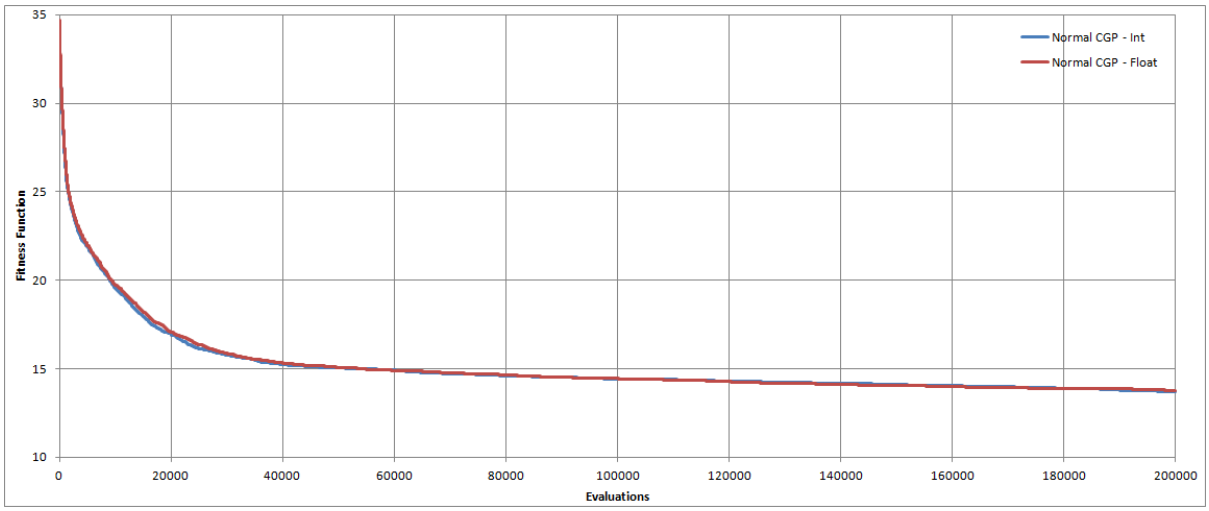

Figure 58 Normal Cartesian Genetic Programming applied to the Wall Avoider problem case using conventional integer chromosome representation and floating point representation

17.4 Conclusion

Although only one "world" layout was investigated, using only one tournament size, it seems that the presence of BLX-0 crossover is not offering an advantage over normal Cartesian Genetic Programming implemented without crossover. This result was seen both graphically in Figure 57 and in the statistics given in Table 40. There also appears to be no correlation between crossover percentage and the effectiveness of the search, indicating that crossover is not offering any advantage. It should be noted however, that only one tournament size was investigated and there may be a tournament size, which when used by BLX-0 crossover, outperforms normal Cartesian Genetic Programming; although from previous test cases this seems unlikely.

The floating point representation appears once again to produce no significant change to the search process; as can be seen from Table 40 and Figure 58. This is a common result seen across the majority of the test cases.

It appears once again that the presence of a tournament selection scheme is detrimental to the search process; when using Cartesian Genetic Programming. This can be seen by comparing the normal Cartesian Genetic Programming's results to those obtained for 0% crossover. It can also be seen that the population sizes were always close to the tournament size (when using tournament selection), which as previously mentioned reduces the effect of the tournament selection process.

The parameters were fairly consistent for all of the experiments investigated in this chapter. There appeared to be no correlation with crossover percentage or any noteworthy difference between when/when not using crossover.

18 Additional Investigations

This chapter does not continue the investigation into the effectiveness of BLX-0 crossover when used by Cartesian Genetic Programming. Instead it will demonstrate the power of Cartesian Genetic Programming[40] (and by extension other forms of Genetic Programming) applied to two previously seen test cases. This was carried out at the author's interest and recorded here for the reader's interest.

18.1 Optimised Full Adder

The full adder circuit has been seen previously in this project in Test Case 2: Synthesis of Boolean Logic. The aim of this test case was to implement a full adder circuit with no constraints on the number or type of logic gates used. One of the powerful properties of Genetic Programming is that complex design criteria can be added to the fitness function which would usually cause traditional design processes to become less effective. These criteria may include: overall propagation delay, overall cost, the number of logic gates or types of logic gates.

To demonstrate this power the fitness function was altered so it now favoured using fewer logic gates[41], therefore changing the search to not only produce a full adder but one which uses the least number of gates. This was achieved by calculating the fitness to be 100 plus the number of incorrect outputs, when the circuit did not correctly implement a full adder. If however the circuit did correctly implement a full adder, the fitness was calculated as the number of active function nodes. The large offset of 100 was to ensure it was never possible to achieve a better fitness than a functioning full adder by using as fewer gates as possible. The sudden reduction in fitness value should not affect the search process, as the elite promoted members of the population are the fittest, and not dependant on by how much they are fitter that the rest of the population.

For this investigation the parameters were as follows: mu = 1, lambda = 4, mutation = 20% and number of function nodes = 20.

[40] This is using "Normal" Cartesian Genetic Programming.
[41] This can be used as a criterion to promote cheap circuit designs.

The Cartesian Genetic Program was given 2,500,000 evaluations to find the best solution; although on average 12903.25 evaluations were required. One of the circuits found which implemented a full adder with the fewest number of logic gates is shown in Figure 59; adapted from [58]. This is in fact the same circuit as the conventional full adder circuit given in Figure 60, taken from [58]. The XOR gate at the output "Cout" in Figure 59, in replace of the OR gate seen in Figure 60, has no effect on the operation of the circuit. Other five logic gate solutions were also found, but are not shown in this report.

This example shows quite clearly how Genetic Programs can be used to solve real world problems with real world constraints.

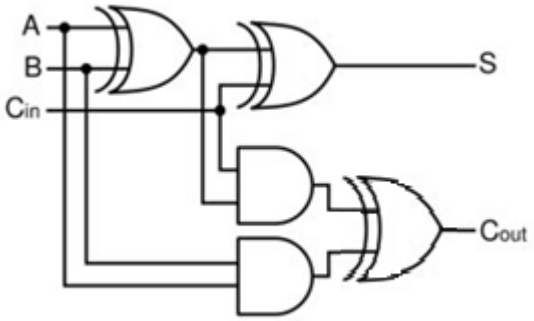

Figure 59 Best Full Adder circuit evolved by the authors Cartesian Genetic Program

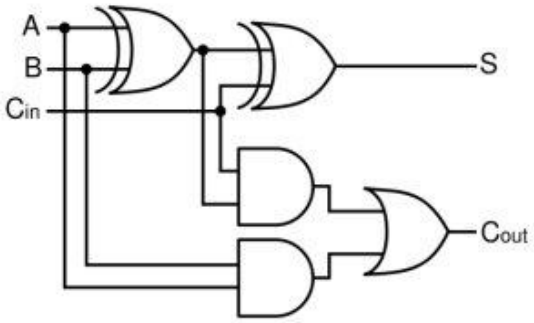

Figure 60 Conventional Full Adder circuit

18.2 Efficient Wall Avoider

The Wall Avoider has been seen previously in Test Case 4: Wall Avoider. In this previous example the fitness assigned to each chromosome was the horizontal distance in squares from the finish line after: the maximum number of moves had been allowed, a wall had

been struck or the finish line reached. This led to the evolution of a solution which navigated it's self across the "world".

In this chapter the fitness function used by the wall avoider was altered so as to try and promote not only solutions to crossing the "world", but solutions which achieved this in the least number of moves; hence efficient wall follower. The fitness function was altered to the following rules: if a wall is struck the assigned fitness is the distance from the finish line plus the maximum allowed moves (100), if the maximum number of moves elapses the fitness is also the distance from the finish line plus the maximum allowed moves and finally if the finish line was reached the fitness is the number of moves which was required in doing so. The objective fitness was then set to zero[42] and left to run for an extended period[43].

This change in fitness function had the desired effect of reducing the number of moves required to navigate the "world" from ~100 down to 80. Various solutions found to crossing the "world", requiring a range of movements, are available to the reader in Appendix D. Again this example shows the power and adaptability of Genetic Programming to produce solutions to a wide range of problems with specific criteria.

[42] An unachievable goal.
[43] 400 million evaluations!

19 Conclusion

This chapter describes the conclusions reached surrounding the effectiveness of BLX-0 crossover when implemented by Cartesian Genetic Programming. The chapter is sectioned into sub headings covering different aspects of the investigation.

Differences in Implementation reviews whether the Cartesian Genetic Program implemented by the author was sufficient to correctly assess the effectiveness of BLX-0 crossover. The Crossover Technique section reaches a high level conclusion over the overall effectiveness of BLX-0 crossover. The Floating Point Representation section reports on whether the floating point chromosome representation, required by BLX-0 crossover, impacts on the effectiveness of Cartesian Genetic Programming. The Tournament Selection section reviews the effect of employing a tournament selection scheme; as used when implementing BLX-0 crossover. The Parameters section reaches conclusions on the parameters found to be most suitable during the project. Finally the Range of Test Cases Investigated discusses if the range of test cases used during the project were suitable and sufficient to assess the effectiveness of BLX-0 crossover.

19.1 Differences in Implementation

As mentioned in Repeating Janet Clegg's Experiments, there were some slight differences between the author's and Janet Clegg's implementations. The first was the method by which mutation percentage was translated into the actual number of mutations carried out on each chromosome. The author used the product of the number of nodes[44] and mutation percentage, whereas Janet Clegg used the product of the number of chromosome parameters and mutation percentage. It is understood that Janet Clegg's implementation is the standard form and the author's implementation is unusual. It is thought however that this difference would not have affected the results presented during this project. This is thought because although the methods of calculating the number of mutations are different, it only causes differences in the mutation percentage required to cause the same number of actual mutations and as the parameters were optimised for each experiment this should not affect the analysis of BLX-0 crossover.

[44] Function plus output.

The second difference, also related to mutation, was that Janet Clegg did not employ mutation on the members of the population generated by crossover. The author, unlike Janet Clegg, applied mutation to those children generated using crossover. It is thought, but was not proven, that the author's method is more likely to produce a better search. This was thought because when the population converges on a solution, all of the members of that population become similar, and so the crossover operator has little to no effect. In this case, not mutating the children generated by crossover causes the population to contain many instances of the same chromosome; this offers no advantage to the search process.

It is concluded therefore that the author's Cartesian Genetic Program and implementation of BLX-0 crossover was sufficient to fairly assess the effectiveness of the new crossover technique.

19.2 The Crossover Technique

Overall this project has shown the BLX-0 crossover, as used by Janet Clegg in her original paper [1], does not benefit the search process for Cartesian Genetic Programs. The only exception was symbolic regression problems, which showed BLX-0 crossover producing a more efficient search process than Cartesian Genetic Programming implemented without. This was also shown to be the case in Janet Clegg's original paper. For all other test cases investigated (synthesis of Boolean logic, function optimisation and wall avoider) it has been shown that BLX-0 crossover, as applied to Cartesian Genetic Programming, is detrimental to the search process.

It was seen for the symbolic regression and function optimisation test cases, that BLX-0 crossover performed better with larger tournament sizes. Further work could therefore investigate the same test cases used throughout this project, but over a wider range of tournament sizes. This would investigate if BLX-0 crossover could outperform normal Cartesian Genetic Programming if a suitable tournament size were used.

It was also indicated in many of the graphical plots (Figure 32, Figure 34, Figure 52 and Figure 57) that variable crossover may have offered a greater advantage that employing the same percentage strength throughout. This was shown by the plots of different crossover percentages intersecting each other at various stages of the search. These result show that there is a real possibility that variable BLX-0 crossover would perform more effectively that

non-varying. Janet Clegg also found variable crossover to be more beneficial than flat rates in her paper [1].

19.3 Floating Point Representation

As mentioned throughout this report, BLX-0 crossover relies on the chromosomes being represented in a floating point form. It is therefore important to ensure that this new chromosome representation does not affect the operation of Cartesian Genetic Programming. It has been shown, for all test cases investigated, the floating point chromosome representation is not affecting the search process to any significant extent. It is therefore concluded that the floating point chromosome representation does not affect the search process.

An interesting result was that in all cases, the results obtained for normal Cartesian Genetic Programming, using the integer and floating point chromosome representation, were never identical; with the integer form slightly outperforming the floating point form in some cases and vise-versa. It is thought, that these differences are the result of the random nature of the heuristic search not being completely removed by the averaging process. This however does not explain some of the larger differences seen; such as in Test Case 1: Symbolic Regression, Table 8, which shows a 7.7% difference between the average evaluations required to reach a solution.

If it is indeed the case that the floating point representation does not affect the search process, as has been shown in the majority of cases, it would enable Cartesian Genetic Programming to employ many different forms of crossover previously uninvestigated. An important point to remember is that when using the floating point representation, it is necessary to employ an additional decoding layer to convert the floating point chromosomes into their integer counterparts. This process incurs an additional time debt to the overall search time; which would have to be overcome by any benefit of the crossover been employed. This was an aspect of the floating point chromosome representation which was not investigated during this project. If a crossover technique, which used the floating point form, was ever found to offer a significant advantage, this additional time debt would have to be considered before a fair conclusion could be drawn.

19.4 Tournament Selection

As mentioned previously in this report, BLX-0 crossover as implemented by Janet Clegg employs the use of a tournament selection scheme to select the parents of the children generated using crossover. It was therefore investigated how the presence of a tournament selection scheme influenced the search process of a normal Cartesian Genetic Program.

The results on how the presence of a tournament selection scheme affects the search process of a "normal" Cartesian Genetic Program were inconclusive. Two of the four test cases investigated, symbolic regression and function optimisation, showed tournament selection to be beneficial for some of the examples and not for others. The remaining two test cases, synthesis of Boolean logic and the wall avoider, showed tournament selection to be detrimental for all examples investigated. It therefore appears that employing a tournament selection scheme is not beneficial to the operation of Cartesian Genetic Programming, but further investigation would have to be undertaken for this to be confidently concluded.

Another aspect of tournament selection is the effect of the tournament size when using BLX-0 crossover. This was investigated as it was unknown which tournament size would produce the best results when using BLX-0 crossover. Interestingly, two of the four test cases investigated[45], symbolic regression and function optimisation, showed the best results were obtained when using the highest tournament size. It is therefore unknown if Cartesian Genetic Programming, implemented with BLX-0 crossover, would produce better results if higher tournament sizes were used. Further work is therefore needed to investigate the effectiveness of BLX-0 crossover over a larger range of tournament sizes. This would identify if BLX-0 crossover offers an advantage when suitable tournament sizes are used.

It has been shown that the effect of the tournament selection scheme, as applied to "normal" Cartesian Genetic Programming, is likely to be detrimental to the search process. It would therefore make an interesting investigation, if the effectiveness of BLX-0 crossover could be assessed without the use of a tournament selection scheme. This could be

[45] Of the remaining two test cases: synthesis of Boolean logic found a tournament size of four produced the best result and only one tournament size was investigated for the wall avoider.

achieved by fixing[46] the mu parameter as two[47], promoting these as the elite members as before, and then generating the remaining population using these two elite members as the parents. This would have the effect of implementing BLX-0 crossover, without the employment of a selection scheme. If time had permitted this technique would have been investigated during this project, unfortunately it is left as a possible further investigation.

19.5 Parameters

The only conclusion which can be drawn over the parameters is low mu values appeared to produce the best results; equalling one or two in most cases. This was the only parameter which was consistently found throughout all of the investigations. Lambda values and mutation rates ranged massively between test cases and even between specific test case examples.

An interesting pattern in many of the optimised parameters was how the population sizes found to produce the best results were often close or equal to the tournament size. It is thought that this has the implication of removing the effect of the tournament selection scheme. For example, if the population is equal to the tournament size, then the same two best chromosomes are always selected as the parents, thus rendering the process of a tournament moot. The result that the population size often approached the tournament size (when optimising the parameters), could therefore count towards a conclusion that tournament selection is detrimental to the search process.

19.6 Range of Test Cases Investigated

It is thought that a good range of test cases were selected to investigate the effectiveness of BLX-0 crossover as applied to Cartesian genetic Programming.

The symbolic regression and the synthesis of Boolean logic test cases were both examples which utilised the ability of Cartesian Genetic Programming to evolve programs; rather than simply optimise parameters. The function optimisation test case was important, as nearly all problems, theoretical and practical, can be broken down into the process of optimising predetermined parameters. The final test case investigated, the Wall Avoider, was a more

[46] It would be possible to use varying mu values if the crossover was implemented to use a variable number of parents, which is perfectly possible.
[47] A value often found to be effective.

unusual test case as it involved the evolution of a program, like the symbolic regression and the synthesis of Boolean logic test cases, but without the author predetermining the operation of the program; as seen when defining a truth table in the synthesis of Boolean logic test case. The Wall Avoider test case also applied two of the programs outputs back as inputs, thus achieving simple feedback which adds greatly to the level of complexity which can be achieved by the evolved programs. Selecting such an unusual test case was not strictly necessary for the investigation into the effectiveness of BLX-0 crossover, but was chosen as an example of something to which Cartesian Genetic Programming had not been previously applied.

If more time were available, the Artificial Ant test case would have been the next to be investigated; as the author wished to investigate if Finite State Machines could be evolved using Cartesian Genetic Programming. This would have been approached by considering the feedback loops to contain values which represent the current state, and the remaining inputs as regular inputs to a Finite State Machine. It is understood that this investigation could have been undertaken using the Wall Avoider test case; which also implemented feedback. This was not undertaken however, as the number of inputs for the Wall Avoider is much larger than for the Artificial Ant and would therefore have been much harder to analyse.

20 Review of the Project

This chapter is included to provide a final review of the project, discussing the key aspects and important sections. The Meeting the Project Aims and Objectives section discusses if/how the project met its aims and objectives. The Design and Coding sections will evaluate the methods used during the respective stages of the project. Software Choices will evaluate whether the JAVA programming language used for this project was a suitable choice. The Optimising Parameters section discusses the optimisation process and how it became a major aspect of the project. The Experimental Strategy section evaluates if the practices adopted during this project were suitably rigorous to provide confidence in the conclusions reached. The Time Management section addresses how the initial time line was followed and evaluates its overall usefulness. Finally the Overall (Personal) section provides a personal view on the project as a whole.

20.1 Meeting the Project Aims and Objectives

This section discusses how each objective of the project was addressed followed by if the overall aims of the project were achieved.

20.1.1 Objectives

All three of the primary objectives were achieved during this project. The first primary objective, investigating the effect of BLX-0 crossover on at least three test cases, was achieved by investigating four separate test cases and analysing the results with two statistical methods (and a graphical plot). The second primary objective, evaluating the effect of the floating point form, was evaluated by comparing Cartesian Genetic Programming with the integer and floating point chromosome representation; for each of the four test cases. The final primary objective, investigating the effect of the tournament selection scheme, was also evaluated by comparing Cartesian Genetic Programming with and without tournament selection; again for each of the four test cases.

Unlike the primary objectives, not all of the secondary objectives were achieved. The first secondary objective, optimising the evolutionary parameters for each experiment, was undertaken for all of the investigated test cases. The following secondary objective, evaluate the parameters found to produce the best results, was also undertaken for each test case.

The penultimate secondary objective, investigating test cases to which Cartesian Genetic Programming has not already been applied, was achieved through the Wall Avoider investigation; although additional examples would have been ideal. The final secondary objective, publish the results obtain during this project, was not attempted due to the results not been considered worthy of publishing; although this was always an ambitious objective.

20.1.2 Aims

Both of the projects primary aims were achieved. The first primary aim was to evaluate if BLX-0 crossover offered a statistically significant benefit to Cartesian Genetic Programming. This was achieved via the four test cases, each evaluated using three techniques: graphical plot of average fitness against evaluation, the average evaluations to find a solution and using Koza's Computational Effort. The second primary aim was to evaluate the effect of the floating point chromosome representation and tournament selection scheme on Cartesian Genetic Programming. This was also undertaken for each of the test cases.

Both of the projects secondary aims were also achieved, although not to the extent which was desired. The first secondary aim, to study further the effects of the parameters governing Cartesian Genetic Programming, was undertaken, but little was learnt from the parameters which were found to be optimum. The final secondary aim was to apply Cartesian Genetic Programming to problems which it has not previously been applied. This aim was undertaken, the Wall Avoider test case, but it is felt that more inventive application could have been used; such as evolving finite state machines.

20.2 Design

The design stage of the project was relatively successful. The author ensured adequate time was assigned to the design stage to provided simple editing and testing of the code in the later stages. This paid dividends when these stages of the project were reached; especially as the code was edited and re-tested for every new test case.

The author's approach of conducting an initial design and starting the coding stage as early as possible was found to be very beneficial. This was because before coding had begun, it was hard to anticipate all of the various aspects of the program. When the final design was

then undertaken, the author was in a far more knowledgeable position to produce better, more thought-out designs.

It should be understood, that the core Cartesian Genetic Program was very simple and therefore easy to implement. The challenging aspect of the code produced for this project was the additional features required for the investigations, including: tournament selection, floating point representation and BLX-0 crossover. For all of these features, it was required that they could be turned on/off quickly via the Parameters Class; this involved significant logic within the author's Cartesian Genetic Program. It was also a requirement that the fitness function and the operations of the function nodes could be switched with minimal effort.

20.3 Coding

The coding stage of the project was undertaken quickly and effectively. This was in part due to a well thought-out design but also due to the author been fluent in the JAVA programming language. There were very few issues encountered during the coding stage, with the majority of the major bugs and issues solved within a day.

The testing strategies used for the code were both effective and inventive. They relied mainly on simple printouts to the consol and through implementing sections of the code as other JAVA programs, MATLAB scripts and excel spreadsheets to compare the results. An example of a particularly inventive strategy was that used for the Wall Avoider. The evolved chromosomes were decoded in excel to produce the corresponding evolved truth tables, which were then implemented in JAVA to show graphically how the logic navigates the unit around their environment. These graphics are included in Appendix D for the reader's interest.

20.4 Software Choices

On reflection of the overall project, the design decision to use the JAVA programming language may have not been the most suitable. Although the author maintains that the object orientated structure of JAVA did significantly reduce development and implementation time; in hindsight the increase in the program execution time is likely to have surpassed any initial saving. The author did consider migrating to C#, a C based purely

object orientated language, but reports found on the relative speeds of JAVA and C# indicated little difference [59] [60].

In hindsight, the author would have ignored the design advantages of object oriented languages and chosen C for its raw performance; or possibly C++ which is much faster than JAVA or C#, with some of the object oriented programming advantages, although certainly not as fully featured.

20.5 Optimising Parameters

As mentioned throughout the project, the optimising of parameters was undertaken for each experiment to ensure a fair comparison between all the strategies. This process is considered essential if fair conclusions were to be drawn on the effectiveness of BLX-0 crossover. It should be noted however, that the process of optimising these parameters is thought to have consumed more time during this project than any other individual section or task; weeks were spent on optimising parameters alone. It is though that the optimising process was fair and complete, and the project would have been nothing without this essential stage.

All of the parameters investigated when optimising the parameters for each experiment are available as excel spread sheets in Appendix D.

20.6 Experimental Strategy

The author considers the experimental strategies used throughout this project to be fair and rigorous. Testing was undertaken for all sections of the code following a strict testing strategy and any newly introduced code was re-tested. Sections of code were also implemented in different languages (MATLAB, excel, Visual Basic) and used as a comparison to the JAVA version to ensure correct implementation.

For each experiment, significant time was taken to find the parameters which produced the best results for each strategy. This ensured a fair comparison between the results which would not be possible if this step was not undertaken.

When producing the statistics to analyse the effectiveness of the different strategies, averages were taken over 1000 runs; ensuring statistically significant results. Three separate techniques were used to compare the different strategies: a plot of average fitness against

evaluation, the average evaluations to reach a solution and Koza's computational effort; all calculated using data from the 1000 runs. Using three separate methods ensured that the trends seen in the results were actually significant and not just artefacts in the methods used to describe the data.

To assess the effectiveness of BLX-0 crossover in isolation, it was necessary to compensate for the effect of the floating point chromosome representation and tournament selection scheme; used alongside its employment. This ensured that BLX-0 was evaluated fairly and led to further insight into the floating point representation and tournament selection scheme; when applied to Cartesian Genetic Programming.

20.7 Time Management

The creation of a time line (Gantt chart) is considered an important step which ensures the project is viewed as a whole and forces all aspects of the project to be considered. It also helps to ensure that the project does not overrun and that it is appreciated at all times how much work is needed to be undertaken to complete the project. However, the project time line outlined for this project was not at all strictly followed; the design, coding and testing stages were undertaken much more efficiently than anticipated; whereas the optimising of parameters took significantly longer. For larger projects than the one undertaken here, the time line would have been updated with every completion, delay or update, to maintain an accurate representation of how the project is progressing. For this project however, it was not considered necessary to spend time maintaining a time line and so its main function was that of a to-do list ensuring the quantity of work to be completed was always clear. Overall therefore, the act of creating a time line was considered more beneficial than actually having it available to follow.

It was decided during the planning stage, that the writing of this final report was to be continually completed throughout the project and not left as a final "write-up". This decision was invaluable to the author, as the writing aspect of the project was considered the most challenging. Continuously "writing up" as the project progressed, helped focus the author to the important aspects of the project and ensured there was no anxiety over the final "write-up".

20.8 Overall (Personal)

Overall I am very pleased with this project. The background literature brought to my attention the shear scope for Evolutionary Strategies and their application into many subject areas. The constructing of my own Genetic Program and then applying it to a selection of different optimisation problems has given me real practical experience with these techniques. I also feel I have had a taste of what a career in research may involve, with the tedious nature of optimising the parameters to the excitement of watching the first evolved solutions to the wall avoider problem. If the research into the application of BLX-0 crossover had produced positive results, it would have been "the icing on the cake". I would have also attempted to publish the results; it is still a goal of mine to produce publishable work in the next few years. I consider this project to have acted as a taster to my next step of studying a PhD in a closely related research area; something I am very excited to be undertaking.

Acknowledgements

I would like to thank Janet Clegg for giving me the freedom to make this project my own, whilst providing support and guidance when it was needed. I would also like to thank both Janet Clegg and Julian Miller for taking the time to read my (rather lengthy) Final Report.

21 Works Cited

[1] J. Clegg, J. A. Walker and J. F. Miller, "A New Crossover Technique for Cartesian Genetic Programming," *Genetic and Evolutionary Computation Conferance,* pp. 1580-1587, 2007.

[2] J. F. Miller, "Cartesian Genetic Programming," in *Cartesian Genetic Programming*, Springer, 2011, pp. 17-34.

[3] J. Skillings, Interviewee, *Getting machines to think like us.* [Interview]. 3 July 2006.

[4] J. McCarthy, "What is Artificial Intelligence?," Stanford University: Computer Science Department, 12 November 2007. [Online]. Available: http://www-formal.stanford.edu/jmc/whatisai/whatisai.html. [Accessed 11 January 2012].

[5] C. R. Darwin, On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life, 1859.

[6] J. F. Miller, "Introduction to Evolutionary Computation and Genetic Programming," in *Cartesian Genetic Programming*, Springer, 2011, p. 1.

[7] L. J. Fogel, On the Organization of Intellect, Wiley, 1966.

[8] I. Rechenberg, Evolutionsstrategie - Optimierung technischer Systeme nach Prinzipien der biologischen Evolution (PhD thesis), 1971.

[9] H. -G. Beye and H. -P. Schwefel, "Evolution strategies: A comprehensive introduction," *Nat. Comput,* vol. 1, no. 1, pp. 3-52, 2002.

[10] J. H. Holland, Adaptation in Natural and Artificial Systems, MIT Press, 1975.

[11] A. E. Eiben, P.-E. Raué and Z. Ruttkay, "Genetic algorithms with multi-parent recombination," in *Parallel Problem Solving from Nature*, 1994.

[12] C.-K. Ting, “On the Mean Convergence Time of Multi-parent Genetic Algorithms Without Selection,” *Advances in Artificial Life,* p. 403–412, 2005.

[13] J. R. Koza, Genetic Programming: On the programming of computers by means of natural selection, MIT Press, 1992.

[14] R. Poli, W. B. Langdon and N. F. McPhee, “A Field Guide to Genetic Programming,” http://lulu.com, 2008. [Online]. Available: http://www.gp-field-guide.org.uk. [Accessed 06 12 2011].

[15] J. F. Miller, “Cartesian Genetic Programming,” in *Cartesian Genetic Programming*, Heidelberg, Springer, 2011, pp. 17 - 34.

[16] J. F. Miller, “An Empirical Study of the Efficiency of Learning Boolean Functions using a Cartesian Genetic Programming Approach,” in *Genetic and Evolutionary Computation Conference*, 1999.

[17] J. F. Miller and P. Thomson, “Cartesian Genetic Programming,” *Proc. European Conference on Genetic Programming,* vol. 1802, p. 121–132, 2000.

[18] J. F. Miller and S. L. Smith, “Redundancy and Computational Efficiency in Cartesian Genetic Programming,” *IEEE Transactions on Evolutionary Computation,* vol. 10, no. 2, p. 167–174, 2006.

[19] J. F. Miller, “What bloat? Cartesian Genetic Programming on Boolean Problems,” *Genetic and Evolutionary Computation Conference,* vol. 3, p. 295–302, 2001.

[20] T. Yu and J. Mille, “Finding Needles in Haystacks Is Not Hard with Neutrality,” *European Conference on Genetic Programming,* vol. 2278, p. 13–25, 2002.

[21] M. Collins, “Finding needles in haystacks is harder with neutrality,” *Genetic and evolutionary computation,* vol. 2, p. 1613–1618, 2005.

[22] E. G. Lopez and R. Poli, “Some steps towards understanding how neutrality affects evolutionary search,” *Parallel Problem Solving from Nature - PPSN,* vol. 4193, p. 778–

787, 2006.

[23] J. A. Walker and J. F. Miller, “Evolution and acquisition of modules in Cartesian genetic programming,” *Conf. Genetic Programming,* vol. 3003, p. 187–197, 2004.

[24] J. A. Walker, J. F. Miller, P. Kaufmann and M. Platzner, “Embedded Cartesian Genetic Programming (ECGP),” in *Cartesian Genetic Programming*, Springer, 2011, pp. 36-60.

[25] S. L. Harding, J. F. Miller and W. Banzhaf, “Self-modifying cartesian genetic programming,” *proceedings of the 9th annual conference on genetic and evolutionary computation,* vol. 1, p. 1021–1028, 2007.

[26] S. L. Harding, J. F. Miller and W. Banzhaf, “Self-Modifying Cartesian Genetic Programming,” in *Cartesian Genetic Programming*, Springer, 2011, pp. 101-124.

[27] S. Harding, J. F. Miller and W. Banzhaf, “Developments in Cartesian Genetic Programming: self-modifying CGP,” *Genetic Programming and Evolvable Machines,* vol. 11, p. 397–439, 2010.

[28] S. Harding, J. F. Miller and W. Banzhaf, “Self Modifying Cartesian Genetic Programming: Fibonacci, Squares, Regression and Summing,” *Proc. European Conference on Genetic Programming,* vol. 5481, pp. 133-144, 2009.

[29] G. M. Khan, D. M. Halliday and J. F. Miller, “Coevolution of Intelligent Agents using Cartesian Genetic Programming,” *Proceedings of Genetic and Evolutionary Computation Conference,* pp. 269-276, 2007.

[30] J. Rothermich and J. f. Miller, “Studying the Emergence of Multicellularity with Cartesian Genetic Programming in Artificial Life,” in *Workshop on Computational Intelligence*, UK, 2002.

[31] S. Harding, “Evolution of image filters on graphics processor units using cartesian genetic programming,” in *IEEE World Congress on Computational Intelligence*, Hong Kong, 2008.

[32] Z. Vasicek and L. Sekanina, “Hardware accelerators for cartesian genetic programming,” *Lecture Notes in Computer Science,* vol. 4971, pp. 230 - 241, 2008.

[33] Z. Gajda and L. Sekanina, “Gate-level optimization of polymorphic circuits using cartesian genetic programming,” *IEEE Congress on Evolutionary Computation,* pp. 1-6, 2009.

[34] L. Altenberg, “The schema theorem and Price's theorem,” *Foundations of Genetic Algorithms,* vol. 3, pp. 23 - 49, 1995.

[35] J. Clegg, “Combining Cartesian Genetic Programming with an Estimation of Distribution Algorithm,” *Proceedings. Genetic and evolutionary computation,* vol. 10, pp. 1333 - 1334, 2008.

[36] N. J. Radcliffe, “EQUIVALENCE CLASS ANALYSIS OF GENETIC ALGORITHMS,” *Complex Systems,* vol. 5, p. 183–205, 1991.

[37] L. J. Eshelman and J. D. Schaffe, “Real-coded genetic algorithms and interval-schemata,” in *Foundations of Genetic Algorithms 2*, San Mateo, CA: Morgan Kaufmann, 1993, p. 187–20.

[38] eclipse, “eclipse,” The Eclipse Foundation, [Online]. Available: http://www.eclipse.org/. [Accessed 05 06 2012].

[39] Mathworks, “Mathworks MAtlab,” [Online]. Available: http://www.mathworks.co.uk/products/matlab/. [Accessed 05 06 2012].

[40] Dropbox, “Dropbox,” Dropbox, [Online]. Available: https://www.dropbox.com/. [Accessed 05 06 2012].

[41] J. Shekel, “Test Functions for Multimodal Search Techniques,” in *Fifth Annual Princeton Conference on Information Science and Systems*, 1971.

[42] A. Griewank, “Generalized descent for global optimization,” *Journal of Optimization Theory and Applications,* vol. 34, no. 1, p. 11–39., 1981).

[43] H. H. Rosenbrock, “An automatic method for finding the greatest or least value of a function,” *The Computer Journal,* vol. 3, pp. 175-184, 1960.

[44] J. A. Walker and J. F. Miller, “Predicting Prime Numbers using Cartesian Genetic Programming,” *European Conference on Genetic Programming ,* vol. 10, pp. 205-216, 2007.

[45] J. A. Walker, J. F. Miller and R. Cavill, “A Multi-chromosome Approach to Standard and Embedded Cartesian Genetic Programming,” *Genetic and Evolutionary Computation Conference,* pp. 903-910, 2006.

[46] R. Munroe, “Travelling Salesman Problem,” XKCD, [Online]. Available: http://xkcd.com/399/. [Accessed 19 January 2012].

[47] G. Reinelt, “Discrete and Combinatorial Optimization,” Heidelberg University, [Online]. Available: http://comopt.ifi.uni-heidelberg.de/index.html. [Accessed 19 January 2012].

[48] D. Jefferson, R. Collins, C. Cooper, M. Dyer, M. R. Korf, C. Taylor and A. Wang, “Evolution as a theme in artificial life: The genesys/tracker system,” in *Artificial Life II*, vol. X, Redwood City, Addison-Wesley, 1991, pp. 549-578.

[49] J. A. Walker, L. Y. Yang, G. Tempesti and A. M. Tyrrell, “ Automatic Code Generation on a MOVE Processor Using Cartesian Genetic Programming,” *Proceedings of International Conference on Evolvable Systems,* vol. 6274, p. 238–249, 2010.

[50] B. Ali, A. E. Almaini and T. Kalganova, “Evolutionary algorithms and their use in the design of sequential logic circuits,” *Genetic Program. Evolvable Mach,* vol. 5, pp. 11 - 29, 2004.

[51] S. M. Lucas, “Evolving Finite State Transducers: Some Initial Explorations,” *Genetic Programming,* vol. 6, pp. 130 - 141, 2003.

[52] M. Gardner, “Mathematical games: The fantastic combinations of John Conway's new solitaire game 'Life',” *Sci. Am.,* vol. 223, no. 4, pp. 120 - 123, 1979.

[53] A. Bendiken, “Arto Bendiken: Conway's Game of Life in JavaScript,” [Online]. Available: http://home.comcast.net/~urbanjost/canvas/life/game-of-life.html. [Accessed 31 5 2012].

[54] D. Kazakov and M. Sweet, “Evolving the game of life,” in *Proceedings of the Fourth Symposium on Adaptive Agents and Multi-Agent Systems*, 2004.

[55] E. Sapin and L. Bull, “The Emergence of Glider Guns in Cellular Automata found by Evolutionary Algorithms,” *Faculty of Computing, Engineering and Mathematical Sciences, University of the West of England.*

[56] H. Alfaro, F. Mendoza and C. Tice, “Generating Interesting Patterns in Conway’s Game of Life Through a Genetic Algorithm,” *University of Central Florida.*

[57] JUnit, “JUnit,” [Online]. Available: http://www.junit.org/. [Accessed 06 06 2012].

[58] A. Greensted, “The Lab Book Pages,” 17 June 2010. [Online]. Available: http://www.labbookpages.co.uk/teaching/evoHW/lab1.html. [Accessed 3 May 2012].

[59] O. Gumus, “Onur Gumus's Blog,” blogspot, 7 2 2009. [Online]. Available: http://reverseblade.blogspot.co.uk/2009/02/c-versus-c-versus-java-performance.html. [Accessed 21 5 2012].

[60] D. Obasanjo, “A comparison of Microsoft’s C# programming language to Sun Microsystem’s Java programming language,” 2007. [Online]. Available: http://www.25hoursaday.com/CsharpVsJava.html. [Accessed 21 5 2012].

[61] D. F. Barrero, D. M. R-Moreno, B. Castano and D. Camacho, “An Empirical Study on the Accuracy of Computational Effort in Genetic Programming,” in *Proceedings of the 2011 IEEE Congress on Evolutionary Computation*, 2011.

[62] J. F. Miller, P. Thomson and T. C. Fogarty, “Designing electronic circuits using evolutionary algorithms, arithmetic circuits: a case study,” in *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science*, Chichester, Wiley, 1998, pp.

105-131.

Appendix A. Janet Clegg's Original Paper

A New Crossover Technique for Cartesian Genetic Programming

Genetic Programming Track

Janet Clegg
Intelligent Systems Group,
Department of Electronics
University of York, Heslington
York, YO10 5DD, UK
jc@ohm.york.ac.uk

James Alfred Walker
Intelligent Systems Group,
Department of Electronics
University of York, Heslington
York, YO10 5DD, UK
jaw500@ohm.york.ac.uk

Julian Francis Miller
Intelligent Systems Group,
Department of Electronics
University of York, Heslington
York, YO10 5DD, UK
jfm7@ohm.york.ac.uk

ABSTRACT

Genetic Programming was first introduced by Koza using tree representation together with a crossover technique in which random sub-branches of the parents' trees are swapped to create the offspring. Later Miller and Thomson introduced Cartesian Genetic Programming, which uses directed graphs as a representation to replace the tree structures originally introduced by Koza. Cartesian Genetic Programming has been shown to perform better than the traditional Genetic Programming; but it does not use crossover to create offspring, it is implemented using mutation only. In this paper a new crossover method in Genetic Programming is introduced. The new technique is based on an adaptation of the Cartesian Genetic Programming representation and is tested on two simple regression problems. It is shown that by implementing the new crossover technique, convergence is faster than that of using mutation only in the Cartesian Genetic Programming method.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming—*Program synthesis*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods and Search

General Terms

Algorithms, Design, Performance

Keywords

Cartesian Genetic Programming, optimization, crossover techniques

1. INTRODUCTION

Koza [6, 7] introduced Genetic Programming (GP) in 1992. He used tree structures as the representation of the members of the population and suggested a crossover technique in which random sub-branches of the parent tree structures are swapped to produce the offspring. This sub-tree crossover was, at the time, thought to be the dominant operator within the optimization process: responsible for exploiting existing genetic material in searching for better solutions. However, it has since been found [1, 8, 9] that this sub-tree crossover technique does not always perform well. Angeline [1] compared the performance of sub-tree crossover with a crossover technique which simply mutated the sub-branches of the trees. It was found that the difference between the performances of sub-tree crossover and that of simply mutating the sub-branches was statistically insignificant. This result implied that, in some cases, sub-tree crossover was no better than some simple mutation of the sub-branches. Luke and Spector [8, 9] also compared sub-tree crossover with a simple mutation of the branches of the trees over a range of problems. They also concluded that sub-tree crossover performed little better than a simple mutation of the branches. Due to findings like these, some people now implement their GP's without using crossover at all, i.e. using mutation only.

By contrast, in Genetic Algorithms (GAs) mutation is considered to be a background operator and of secondary importance to the crossover operator. GAs have been extremely successful when applied to many real life complex optimisation problems [3, 2, 4]. Although mutation is an important genetic operator in the GA, the crossover operator contributes a great deal to its performance. Much work has been done in analysing the effects of crossover and mutation on the performance of a GA [5, 15, 17]. In [5], Jong presents experimental results illustrating the power of crossover and in [14] Schaffer compares mutation and crossover in a GA and concludes that mutation alone is not always sufficient. The inspiration for the work in this paper has been to find a new crossover technique in Genetic Programming which can contribute to the performance of the GP as much as crossover operators contribute to the performance of a GA.

Recently, Miller and Thomson [11, 12] introduced a new form of GP called Cartesian Genetic Programming (CGP), which uses directed graphs to represent programs rather than the more traditional representation of programs as trees. The CGP is implemented with mutation only and has not, up to the present time, used a crossover technique. Even so, it has been shown that the CGP performs better

GECCO'07, July 7–11, 2007, London, England, United Kingdom.

than the traditional GP. The work described in this paper is based on this CGP representation.

This paper introduces a new method for crossover in Genetic Programming which improves the performance of the GP by speeding up its convergence considerably. The new technique has been developed based on the Cartesian Genetic Programming representation described above. The CGP representation is modified in order to enable the new crossover technique to be applied. Crossover when applied to a CGP using the traditional representation hinders its performance rather than improves it, and this has been the motivation for introducing the new representation here. The new method of crossover has been tested on two simple regression problems and the results show that it successfully speeds up the convergence of the CGP for these problems. Section 2 of this paper describes the traditional CGP method and Section 3 shows how crossover techniques fail when the CGP is in its traditional integer representation. Section 4 introduces the new representation and crossover and Section 5 describes the regression problems which the new crossover is tested on. Section 6 reports the results of using the new technique on these regression problems and finally Section 7 discusses conclusions and future work.

2. CARTESIAN GENETIC PROGRAMMING (CGP)

Cartesian Genetic Programming is a form of Genetic Programming (GP) invented by Miller and Thomson [12], for the purpose of evolving digital circuits. However, unlike the conventional tree-based GP [6], CGP represents a program as a directed graph (that for feed-forward functions is acyclic). The benefit of this type of representation is that it allows the implicit re-use of nodes in the directed graph. CGP is also similar to another technique called Parallel Distributed GP, which was independently developed by Poli [13]. Originally CGP used a program topology defined by a rectangular grid of nodes with a user defined number of rows and columns. However, later work on CGP showed that it was more effective when the number of rows is chosen to be one [19]. This one-dimensional topology is used throughout the work we report in this paper.

In CGP, the genotype is a fixed length representation and consists of a list of integers which encode the function and connections of each node in the directed graph. However, the number of nodes in the program (phenotype) can vary but is bounded, as not all of the nodes encoded in the genotype have to be connected. This allows areas of the genotype to be inactive and have no influence on the phenotype, leading to a neutral effect on genotype fitness called neutrality. This unique type of neutrality has been investigated in detail and found to be extremely beneficial to the evolutionary process on the problems studied [12, 19, 16].

Each node is encoded by a number of genes. The first gene encodes the node function, whilst the remaining genes encode where the node takes its inputs from. The nodes take their inputs in a feed forward manner from either the output of a previous node or from the program inputs (terminals). Also, the number of inputs that a node has is dictated by the arity of its function. The program inputs are labelled from 0 to $n-1$, where n is the number of program inputs. The nodes encoded in the genotype are also labelled sequentially from n to $n+m-1$, where m is the user-defined bound for the number of nodes. If the problem requires k program outputs, then k integers are added to the end of the genotype, each encoding a node output in the graph where the program output is taken from. These k integers are initially set as the outputs of the last k nodes in the genotype. Figure 1 shows a CGP genotype and corresponding phenotype for the function $x^6 - 2x^4 + x^2$ and Figure 2 shows the decoding procedure.

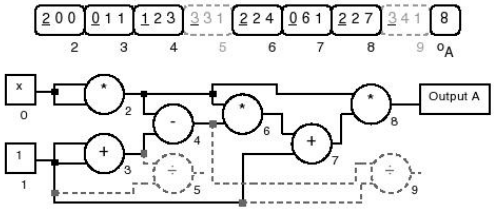

Figure 1: A CGP genotype and corresponding phenotype for the function $x^6 - 2x^4 + x^2$. The underlined genes in the genotype encode the function of each node, the remaining genes encode the node inputs. The function lookup table is: +(0), -(1), *(2), ÷(3). The index labels are shown underneath each program input and node. The inactive areas of the genotype and phenotype are shown in grey dashes.

3. ATTEMPTS AT CROSSOVER IN CGP

This section of the paper reports on some attempts at crossover when the CGP representation is in its original form (as described in the previous section). Four variations of crossover have been tested, but all four failed to improve the convergence of the CGP. Compared to running the CGP with mutation only, the addition of these crossover techniques actually hindered its performance. It is for this reason most people use the CGP without crossover (i.e. using mutation only). Results of two of the four crossover techniques are given here and these results emphasise the need for the new representation and crossover introduced later in the paper.

The crossover methods have been tested on a very simple regression problem given by the equation $x^2 + 2x + 1$. A sample of twenty data points are taken from the interval [0,1], and the cost function is defined as the sum of the squared differences between the population member's values and the true function values at each of the data points. A population size of 30 has been used with 28 offspring created at each generation. Tournament selection has been chosen to select the parents and a mutation rate of 20% has been used. The maximum number of nodes has been set at 5. For each crossover technique the CGP has been run 1000 times and the average convergence over these 1000 runs is recorded at each generation.

The first crossover technique is based on the single point crossover in a binary GA; we treat the nodes in the CGP representation the same as the binary digits in the binary GA. A random node is chosen in the CGP genotype and the offspring are created by swapping the parents nodes at this point (i.e. the first offspring will take all nodes from

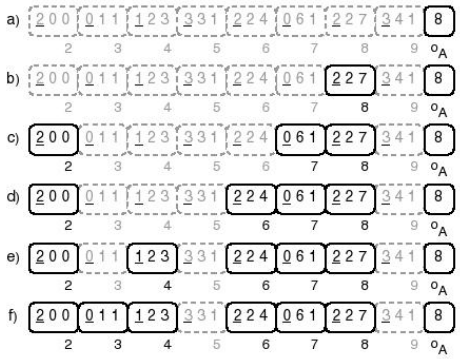

Figure 2: The decoding procedure of a CGP genotype for the function $x^6 - 2x^4 + x^2$. a) Output A (o_A) connects to the output of node 8, move to node 8. b) Node 8 connects to the output of nodes 2 and 7, move to nodes 2 and 7. c) Nodes 2 and 7 connect to the output of node 6 and program inputs 0 and 1, move to node 6. d) Node 6 connects to the output of nodes 2 and 4, move to node 4, as node 2 has already been decoded. e) Nodes 4 connects to the output of nodes 2 and 3, move to node 3. f) Node 3 connects to program input 1. When the recursive process has finished, the genotype is fully decoded.

parent one to the left of this node and all nodes from parent two to the right of this node). Figure 3 displays the average convergence for the two cases; (a) mutation only with a rate of 20% (b) 50% crossover with mutation at 20%. It can be seen that the addition of crossover slows the convergence of the CGP rather than improving it.

The second crossover technique involves picking a random node in the CGP genotype and the offspring are created by swapping this single node in the parents. Figure 4 displays the detrimental effect of this crossover technique. Two other crossover techniques were tried with similar results to those in Figures 3 and 4. It seems that swapping the integers (in whatever manner) in the CGP representation disrupts the performance of the CGP. This has been the motivation for the introduction of the real-valued representation and new crossover technique described in this paper.

4. INTRODUCING THE NEW METHOD

The proposed crossover method for CGP is heavily inspired by the real-valued crossover operator found in real-valued GAs. Normally the CGP genotype consists of a list of integers to encode the directed graph (as described in Section 2). However, to incorporate this type of crossover operator into CGP requires a modification to the CGP representation itself. The modified representation introduces a new level of encoding into the CGP genotype, which represents the directed graph as a fixed length list of real-valued numbers. Each real-valued number corresponds to a single gene in the CGP genotype (as is the case with the standard CGP representation) and its value lies in the range $[0, 1]$. Each node in CGP is still represented by a number of genes and the purpose of each gene still remains as it would

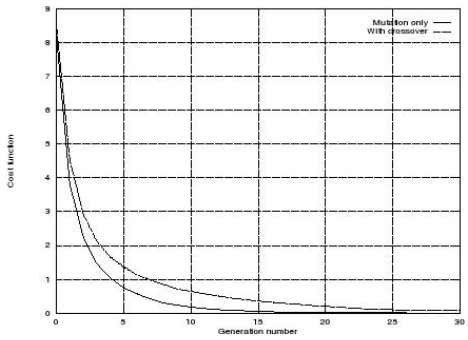

Figure 3: Average convergence of CGP with and without the first crossover technique

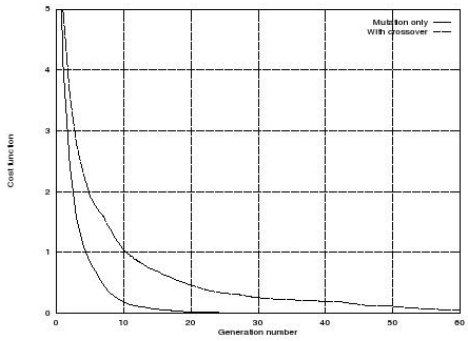

Figure 4: Average convergence of CGP with and without the second crossover technique

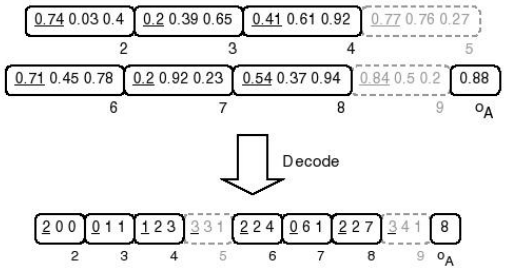

Figure 5: The decoding process between the real-valued and integer-based genotypes. The underlined genes encode the functions and the remaining genes encode the node inputs. The function genes are decoded using Equation 1 whilst the input genes are decoded using Equation 2.

in the standard CGP genotype; the first real valued gene encodes the function of the node whilst the remaining real-valued genes encode the inputs of the node. An example of the new representation is shown in Figure 5, which also shows the decoding process to the standard CGP genotype.

The decoding process from the real-valued genotype to the integer-based genotype is achieved by a combination of Equation 1, if a gene, say $gene_i$, encodes the function of a node and Equation 2, if $gene_i$ encodes the input of a node.

$$floor(gene_i * func_{total}) \tag{1}$$

$$floor(gene_i * nodeterm_j) \tag{2}$$

In Equations 1 and 2, i is defined as $0 <= i < gene_{total}$, where $gene_{total}$ is the number of genes in the genotype, $func_{total}$ is the number of functions, $nodeterm_j$ is the node or terminal number, where j is defined as $0 <= j <= nodeterm_{total}$ and $nodeterm_{total}$ is the number of nodes in the genotype and the number of terminals.

This decoding procedure is a many-to-one mapping between each value in the real-valued genotype and each value in the integer-based genotype. Therefore each integer value is actually represented by a range of values in the real-valued representation. This is summarised in Equation 3, which shows the real-valued range for each function, $func_k$, and Equation 4, which shows the real-valued range for each node input, $input_j$.

$$func_k \in [\frac{func_k}{func_{total}}, \frac{func_k + 1}{func_{total}}] \tag{3}$$

$$input_j \in [\frac{nodeterm_j}{nodeterm_{total}}, \frac{nodeterm_j + 1}{nodeterm_{total}}] \tag{4}$$

In Equations 3, $func_k$, is the k^{th} function in the function set and $func_{total}$ is the total number of functions in the function set. Whilst in Equation 4, $input_j$ is the node's input connection to the j^{th} terminal.

By introducing the new representation, each individual in the population can be thought of as a particular value of a function of n variables, where $n = gene_{total} * node_{total} + output_{total}$, $gene_{total}$ is the number of genes, $node_{total}$ is the number of nodes in the genotype, and $output_{total}$ is the number of outputs.

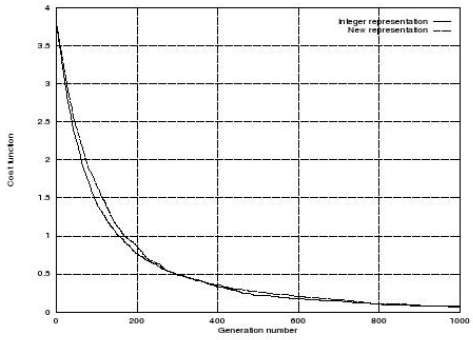

Figure 6: Comparison of the integer CGP with the real-valued CGP without crossover

$$f(x_1, x_2, \ldots, x_n) \tag{5}$$

The optimization then becomes that of finding the values of these n variables which produce an optimal result.

Crossover is performed as in a floating point Genetic Algorithm. Two parents, p_1 and p_2 are chosen and crossover is performed using Equation 6 to produce two offspring, o_1 and o_2. A uniformly generated random number, r_i, is chosen for each offspring, o_i where $0 < r_i < 1$ and $0 <= i < 2$.

$$o_i = (1 - r_i) * p_1 + r_i * p_2 \tag{6}$$

The mutation operator for the real-valued representation is based on the mutation operator normally found in CGP, the only difference is that it changes the value of a gene to a uniformly generated random real-valued number from the region $[0, 1]$.

Without crossover, the new real-valued representation does not change the behaviour of the CGP very much at all. This can be seen in Figure 6 which displays the average convergence of the CGP over 1000 runs using mutation only for the two CGP representations; the original integer representation and the new real-valued representation. This test has been performed on the first regression problem described in the next section on experimental results.

Using the new crossover method described in this section means that, mathematically, the problem has become that of simply optimising a function of real-valued variables. Instead of randomly changing the input to some complex composite function (as in the case of tree crossover) to attempt to achieve a better solution, the values of the genes are free to slide continuously around the problem space searching for the best solution.

5. EXPERIMENT DETAILS

The new method has been tested on two of the regression problems investigated by Koza, and their equations are given in Equations 7 and 8 below.

$$x^6 - 2x^4 + x^2 \tag{7}$$

$$x^5 - 2x^3 + x \tag{8}$$

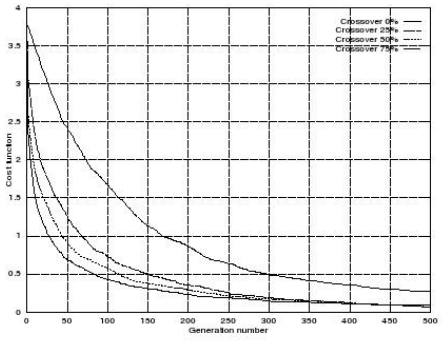

Figure 7: Average convergence for CGP with various crossover rates on $x^6 - 2x^4 + x^2$

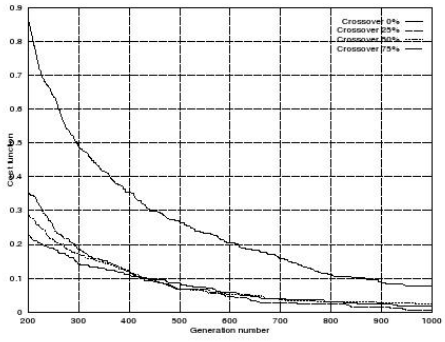

Figure 8: Average convergence of CGP for the latter generations on $x^6 - 2x^4 + x^2$

A sample of fifty data points are taken from the interval [-1,1], and the cost function is defined as the sum of the absolute values of the differences between the population member's values and the true function values at each of the data points. The algorithm is classed as converged when all of these absolute values are less than 0.01 (this is the criteria Koza used for convergence).

A population size of 50 has been used with 48 offspring created at each generation. Tournament selection has been chosen to select the parents and crossover as described by Equation 6 has been used. The maximum number of nodes has been set at 10 initially and a mutation rate of 20% has been used. Different rates of crossover have been investigated, 0%, 25%, 50% and 75%. Note that 0% crossover is equivalent to the traditional CGP which uses mutation only, although the traditional CGP has been applied with a smaller mutation rate and population size in most work prior to this. For each crossover rate the new algorithm has been run 1000 times and the average convergence over these 1000 runs is recorded at each generation.

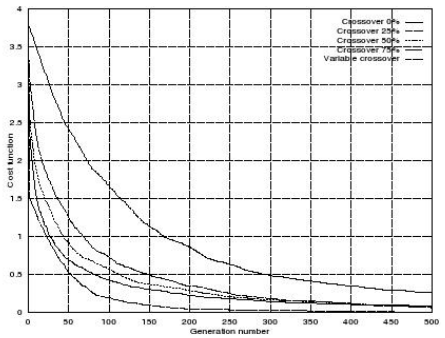

Figure 9: Average convergence for CGP with various crossover rates, including a variable crossover rate on $x^6 - 2x^4 + x^2$

6. RESULTS

For all the figures in this section of the paper, the horizontal axis represents generation number in the CGP and along the vertical axis is the cost function for the best member of the population (averaged over 1000 runs) for that particular generation number. Figure 7 displays this average convergence (over the 1000 runs of the CGP) for each crossover rate for the regression problem in Equation 7.

From Figure 7, it is apparent that this new form of crossover has a large effect on convergence (unlike tree crossover). If Figure 7 is displayed for the latter generations (see Figure 8), then it can be seen that although the new crossover improves convergence for the initial generations, it does not particularly improve convergence for the latter generations. It is not clear at this stage why this should be the case, but future work will involve investigating possible reasons why. For now, we accept that crossover works better for the initial generations and try a crossover technique which varies with generation number. Since for the initial generations it seems that the larger the crossover rate the faster the convergence, we choose an initial crossover rate for generation number one of 90%. Also since it seems that by generation number 200, crossover is not having a large effect on convergence, we arrange that crossover is 0% by this generation. Therefore for our variable crossover we begin at generation number one with 90% crossover and reduce the crossover rate linearly such that by generation number 180 crossover is being performed 0% of the time. This variable crossover technique is simply based on analysing these initial results, future work will involve investigating alternative variable crossover techniques. Figure 9 displays the average convergence for the variable crossover technique and it can be seen that this means a faster convergence over all generation numbers.

Table 1 displays the average number of generations required to reach convergence and the computational effort as described by Koza in [6] and shown in Equation 9. The significance of the results is also assessed using the non-parametric Mann-Whitney U test [10]. The U values produced from the Mann-Whitney U test are denoted with: a * if they are classed as marginally significant ($P < 0.05$), a † if they are classed as significant ($P < 0.01$) or a ‡ if they

Table 1: The average number of generations and computational effort (CE) required by CGP with ten nodes to converge on a solution for $x^6 - 2x^4 + x^2$

Crossover Rate (%)	Average Generations	CE	U
0	168	30,000	-
25	84	9,000	309,778 ‡
50	57	8,000	261,533 ‡
75	71	6,000	226,303 ‡
Variable	47	10,000	263,269 ‡

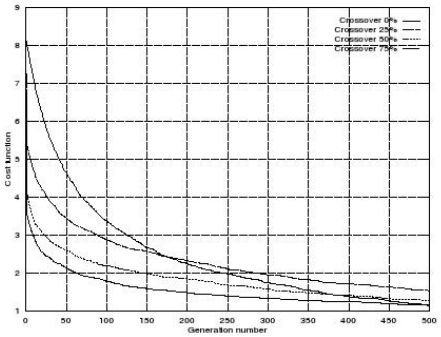

Figure 10: Average convergence for the second regression problem $x^5 - 2x^3 + x$

are classed as highly significant ($P < 0.001$).

$$P(M,i) = \frac{N_s(i)}{N_{total}}$$
$$R(z) = ceil\left(\frac{\log(1-z)}{\log(1-P(M,i))}\right) \quad (9)$$
$$min\, I(M,i,z) = MR(z)\, i + 1$$

Figure 10 displays the average convergence for the regression problem given in Equation 8. Note that for this problem it is more pronounced that the crossover has a big effect on convergence for the initial generations but has less effect for the latter generation. Variable crossover improves the convergence over all generations, as can be seen in Figure 11.

Table 2 contains the average number of generations required to converge together with Koza's computational effort figure for the various percentages of crossover.

For this second regression problem, crossover does not seem to have as big an effect as for the previous problem. This seems to be because, for this problem, occasional runs take a huge number of generations to converge. This can be seen in Figure 12 which shows the number of generations required to converge for 100 runs of the two regression problems. As can be seen in the figure, for the first problem most runs take approximately the same number of generations to converge, whereas in the second problem there are occasional runs which take a very large number of generations to converge. This trait will be investigated in more detail in future work.

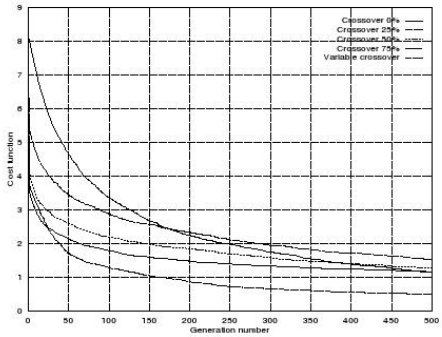

Figure 11: Average convergence for the second regression problem $x^5 - 2x^3 + x$ including results for a variable crossover rate

Table 2: The average number of generations and computational effort (CE) required by CGP with ten nodes to converge on a solution for $x^5 - 2x^3 + x$

Crossover Rate (%)	Average Generations	CE	U
0	516	44,000	-
25	735	24,000	502,024
50	691	14,000	422,394 ‡
75	655	11,000	343,119 ‡
Variable	278	13,000	294,577 ‡

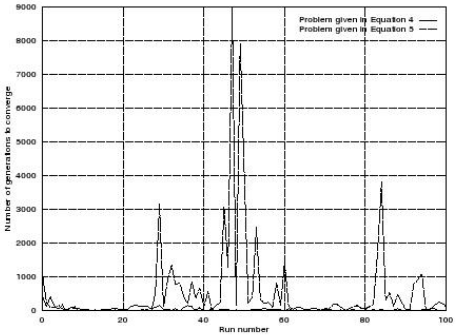

Figure 12: The number of generations to converge over 100 runs for both symbolic regression problems

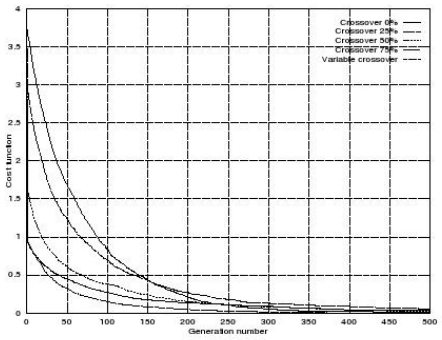

Figure 13: Average convergence for the symbolic regression problem in Equation 7 using CGP with fifty nodes

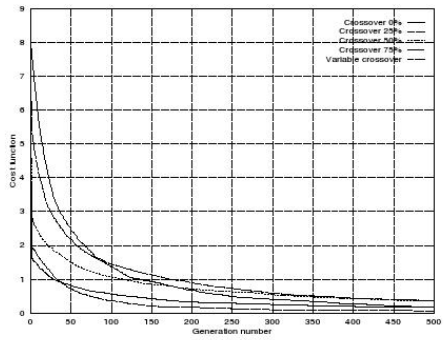

Figure 14: Average convergence for the symbolic regression problem in Equation 8 using CGP with fifty nodes

Table 3: The average number of generations and computational effort (CE) required by CGP with fifty nodes to converge on a solution for $x^6 - 2x^4 + x^2$

Crossover Rate (%)	Average Generations	CE	U
0	78	18,000	-
25	85	13,000	443,769 ‡
50	71	11,000	420,519 ‡
75	104	13,000	463,118 †
Variable	45	14,000	401,205 ‡

The number of nodes used is now increased from 10 to 50 in both regression problems. Figure 13 displays the results for the regression problem in Equation 7, and Table 3 gives the average number of generations to converge together with Koza's computational effort figure. Figure 14 and Table 4 are the same for the regression problem given in Equation 8.

The results in this section show that the new technique enhances the performance of CGP. The majority of the U values produced are classed as highly significant, which supports the findings from computational effort figures and indicates that the use of crossover in CGP is beneficial when applied to symbolic regression problems. The reason the new method works well could be the fact that the problem has been transformed into that of simply minimising a function (the cost function) of n variables (where n is the total number of real-valued numbers in the representation). Crossover methods tested in the past have involved swapping the integers in the CGP representation in some manner, and it is thought that this may produce too great a change to the functional form of the current solution. By making the cost function into a simple function of variables and performing crossover in the way described in this paper, the values of the variables are allowed to move in a continuous manner to their optimal values.

Table 4: The average number of generations and computational effort (CE) required by CGP with fifty nodes to converge on a solution for $x^5 - 2x^3 + x$

Crossover Rate (%)	Average Generations	CE	U
0	131	18,000	-
25	193	17,000	539,076 †
50	224	12,000	454,875 ‡
75	152	19,000	554,642 ‡
Variable	58	16,000	470,984 *

It is also thought that another possible reason for the success in the new technique may be attributed to the fact that for nodes to the far left of the representation, the interval [0,1] is spit into a less number of sub-sections and therefore it will "change" less due to the crossover. In contrast for nodes to the far right of the representation, the interval [0,1] is split into more sub-sections and therefore is more likely to change through crossover. It is thought that this could help the optimisation due to the fact that functions to the left can be thought of as fundamental sub-functions of the entire solution function.

7. CONCLUSIONS AND FUTURE WORK

In this paper we have introduced a new crossover technique, which improves the performance of Cartesian Genetic Programming. The CGP representation is adapted slightly in order to allow the new crossover. It has been found that this new representation together with crossover reduces the average number of generations required to converge by 72% in the case of the regression problem given in Equation 7, and by 46% in the case of the problem in Equation 8. It has been shown that for the regression problem in Equation 8 the new crossover technique does not have as good an effect on convergence as for the first regression problem. This is thought to be because of the fact that occasional runs of the CGP for this second problem take a huge number of generations to converge. Future work will involve investigating this trait.

The results in this paper for the cases where crossover is set at 0% are equivalent to those of the traditional CGP, which uses mutation only. The computational effort figures reported in this paper for 0% crossover are similar to those reported for the traditional CGP [18], although in this paper

a larger mutation rate and population size have been used. Future work will involve investigating how changing these parameter values in the CGP (i.e. mutation rate, population size, parent selection method) affects the performance of the new method. We will also investigate the fact that crossover has more effect for the initial generations and try alternative method of variable crossover.

This paper reports on initial testing of the new technique when applied to two regression problems. Future work will involve testing the new method on other problems, in particular on larger problems and other types of problems.

8. REFERENCES

[1] P. Angeline. Subtree crossover: Building block engine or macromutation? In *Genetic Programming 1997: Proceedings of the Second Annual Conference (GP97)*, pages 9–17, Stanford University, USA, 13–16July 1997. Morgan Kaufman.

[2] J. Clegg, J. Dawson, S. Porter, and M. Barley. The use of a genetic algorithm to optimize the functional form of a multi-dimensional polynomial fit to experimental data. In *2005 IEEE Congress on Evolutionary Computation*, volume 1, pages 928–934, Edinburgh, 2005.

[3] J. Clegg, A. Marvin, J. Dawson, and S. Porter. Optimisation of stirrer designs in a reverberation chamber. In *IEEE Trans. EMC*, volume 47 of *No. 2*, pages 399–403, 2005.

[4] L. Dawson, J. Clegg, S. Porter, J. Dawson, and M. Alexander. The use of genetic algorithms to maximise the performance of a partially lined screened room. In *IEEE Trans. EMC*, volume 44 of *No. 1*, pages 233–242, 2002.

[5] K. De Jong. An analysis of the behaviour of a class of genetic adaptive systems. In *Doctoral Thesis, Department of Computer and Communication Sciences. University of Michigan, Ann Arbor.*, 1975.

[6] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[7] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, 1994.

[8] S. Luke and L. Spector. A comparison of crossover and mutation in genetic programming. In *Genetic Programming 1997: Proceedings of the Second Annual Conference (GP97)*, pages 240–248, Stanford University, USA, 13–16 July 1997. Morgan Kaufman.

[9] S. Luke and L. Spector. A revised comparison of crossover and mutation in genetic programming. In *Genetic Programming 1998: Proceedings of the Third Annual Conference (GP98)*, pages 208–213, University of Wisconsin, Madison, WI, USA, 22–25 July 1998. Morgan Kaufman.

[10] H. Mann and D. Whitney. On a test of whether one of 2 random variables is stochastically larger than the other. *Annals of Mathematical Statistics*, (18):50–60, 1947.

[11] J. F. Miller. An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In *GECCO 1999: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1135–1142, Orlando, Florida, 1999. Morgan Kaufmann.

[12] J. F. Miller and P. Thomson. Cartesian genetic programming. In *Proceedings of the 3rd European Conference on Genetic Programming (EuroGP 2000)*, volume 1802 of *Lecture Notes in Computer Science*, pages 121–132, Edinburgh, 2000. Springer-Verlag.

[13] R. Poli. Parallel Distributed Genetic Programming. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 403–432. McGraw-Hill, UK, 1999.

[14] J. Schaffer and L. Eshelman. On crossover as an evolutionarily viable strategy. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 61–68, La Jolla, CA, 1991. Morgan Kaufmann.

[15] W. Spears and K. De Jong. On the virtues of uniform crossover. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 230–236, La Jolla, CA, 1991. Morgan Kaufmann.

[16] V. K. Vassilev and J. F. Miller. The advantages of landscape neutrality in digital circuit evolution. In *Proceedings of the 3rd International Conference on Evolvable Systems (ICES 2000)*, volume 1801 of *Lecture Notes in Computer Science*, pages 252–263. Springer Verlag, 2000.

[17] M. Vose and G. Liepins. Schema disruption. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 237–242, La Jolla, CA, 1991. Morgan Kaufmann.

[18] J. Walker and J. Miller. Automatic acquisition, evolution and re-use of modules in cartesian genetic programming. *to be published in IEEE Transactions on Evolutionary Computation*, 2007.

[19] T. Yu and J. F. Miller. Neutrality and the evolvability of boolean function landscape. In *Proceedings of the 4th European Conference on Genetic Programming (EuroGP 2001)*, volume 2038 of *Lecture Notes in Computer Science*, pages 204–217. Springer-Verlag, 2001.

Appendix B. Optimizing Parameters

Many of the experiments described during this project discuss the concept of optimising the parameters governing the evolutionary process. This appendix discusses how this is achieved.

It should be noted, that there is no way of ensuring that the optimum parameters have ever been chosen, this is a search space in its self. Therefore, the process described in this section does not produce the optimum parameters, but should lead to suitable parameters. The process is begun by selecting typical parameters used by Cartesian Genetic Programs as shown in Table 42. In cases which require a higher population size, to accommodate the tournament size been employed, the mu parameter is increased.

Table 42 Parameters typically used by Cartesian Genetic Programs

Parameter Name	Value
Mu	1
Lambda	4
∴ Population Size	5
Mutation Rate[48]	~5%

The optimising of the parameters begins by setting the parameters to those given in Table 42 as the initial parameters. Each individual parameter is then be varied separately (keeping the other parameters as the initial parameters). The values for each parameter which produced the best results are then used as the next set of parameters. This process is then repeated twice. Once this has been completed each individual parameter is varied once again, but this time keeping any positive changes made to the other parameters. It is thought that this produces a strong parameter set for each given test case.

It should be noted, that when changing the population size (the sum of mu and lambda) the maximum number of generations is also changes ensure the maximum number of evaluations[49] remains constant.

[48] If 5% mutation actually leads to no mutation been carried out, due to the small size of the chromosome, then the smallest possible mutation rate which actually performs mutation is selected.

Appendix C. Computational Effort

In order for comparisons to be made between different search methods, it is necessary that quantitative values are assigned to each experiment. Of the many methods of calculating such values, John Koza's Computational Effort, as described in his influential book [13], is one of the most popular. The fact that it continues to be one of the most stated statistics may only be because of its previous popularity and not because it is the most suitable statistic. There are papers [61] which indicate that Koza's Computational Effort is not the most suitable statistic and other parameters should be quoted in the literature. Regardless, Koza's Computational Effort is widely used and is included throughout this project for completeness.

The concept behind Computational Effort is to represent the number of evaluations[50] required to find a solution with a given probability. The equation for calculating Computational Effort is as follows:

$$I(M,i,z) = Mi\left[\frac{\ln(1-z)}{\ln(1-P(M,i))}\right]$$

Where M is the population size, i is the number of generations, and z is the confidence level in reaching a solution. The confidence level z is usually set to 99%; this value is used throughout this project. The product of M and i represents the total number of evaluations analyzed during the experiment. $P(M,i)$ is the probability of finding a solution within the number of given evaluations; this value is found empirically form experiments.

[49] The number of evaluations is the maximum number of different solutions which are inspected. It is the product of population size and the maximum number of generations.
[50] Number of solutions inspected.

Appendix D. The Disc

A disc is provided with this project to contain files which are both very large and significantly more useful as digital data; these includes all the experimental results and the author's code. A HTML index file is included on the disc to aid navigation and to provide a clean user interface. The disc also includes autorun functionality, loading the HTML page when the disc is inserted into a disc drive; although depending on the users set up, this may not operate. If the autorun feature fails to load the HTML page, the user is instructed to navigate to the index.html file and open it in a web browser of their choice[51].

The disc contains a digital copy of this final report as a PDF file complete with functioning links; along with two academic papers which comprise the essential background reading. It also contains all of the author's code used throughout this project with a digital copy of the Class Diagram showing its structure. There is also the raw data generated when optimizing the parameters and when conducting the large experiments. The Matlab scripts used to generate the 3D plots in the Possible Test Cases chapter are also available. Finally there is a selection of animations showing the Wall Avoider at various levels of competency. These animations are provided as .jar JAVA executables.

[51] The author recommends Google Chrome, as it has inbuilt functionality to load PDF documents and the code files within the browser.