On Evolutionary Algorithms for Evolving Code in a Closed Context

Michael Cassar

June 2014

Submitted to the Malta College of Arts, Science and Technology institute of Information Technology in partial fulfilment of the requirements for the Bachelor of Science (Honours) in Software Development

AUTHORSHIP STATEMENT

This dissertation is based on the results of research carried out by myself, is my own composition, and has not been previously presented for any other certified or uncertified qualification.

The research was carried out under the supervision of Mr. Andrew Cortis.

Signed ____________________ Date ____________________

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my family for their attention, undivided support, and dedication throughout this stressful year, words cannot express how grateful I am to you all. I am however, especially grateful to my sister Francesca, who through constant annoyance, has always found a way to make me laugh and keep my spirits up whenever I needed it the most.

After thanking my family, I would like to express the deepest appreciation to my dissertation tutor, Mr. Andrew Cortis. In addition to helping me in this dissertation, whenever any help was required at any moment, Mr. Cortis has also taught me a vast number of skills that have improved my programming knowledge dramatically. During this dissertation, Mr. Cortis has not only aided me, but has inspired a substantial level of adventure and enthusiasm with regards to implementing it, and also with regards to programming in general. In addition to the above, he also provided me with the genetic algorithm framework that he had built, which has helped me greatly, as it pretty much forms the foundation of this dissertation. Without his supervision and continual help, this dissertation would not have been possible. I would also like to thank Mr. Andrew Cortis for introducing me to Data Structures and Algorithms, his lectures have taught me a great deal and I hope that his teachings inspire future students as they have done me.

I would also like to take this opportunity to thank a two other lecturers, who throughout the year have been of significant support and that have both directly and indirectly contributed to my dissertation by improving my skill set and knowledge:

- ✓ I would like to thank Mr. Ryan Attard, for introducing me to Secure Software Development, in addition to fuelling my passion for Programming, Web Architectures, SQL, and Non-Blocking Asynchronous patterns.
- ✓ I would like to thank Ms. Christine Dimech, for proving how wrong I was, when I initially thought that I would never be good at Mathematics. In addition to improving my Mathematical background, she has shown me that I can achieve anything that I work hard at achieving.

TABLE OF CONTENTS

ABSTRACT

Evolutionary algorithms are used in many situations where complex, abstract, and or hard to achieve solutions have to be found. Some applications are: cancer diagnosis; artificial creativity; and spacecraft antennae design. Genetic algorithms are a subset of evolutionary algorithms that utilise Darwinian Evolution Theory principles such as splicing; mutation; and cross-over functions, to evolve genomic sequences that represent solutions to a given problem.

Core War is a programming game, in which two or more programs are executed in a sandboxed memory battle, where the aim of each program is to terminate the other's process.

An application was built to evolve Core War programs using genetic algorithms for the purpose of searching for improvements on the original programs. By making use of an Elo Rating system, the fitness of each generated program is evaluated, one generation after another. To calculate a program's rating, the program is battled against other programs to measure their wins, ties and losses. The updated Elo Rating is then used to determine which programs are discarded or saved and carried over to future generations where they can continue to evolve. The Elo Rating is reset after every generation, to prevent programs carried over from previous generations from having high Elo Ratings that dominate the new generation, and do not give new programs the opportunity to be carried over. The particular genetic algorithm approach taken is based on simulated annealing, but also makes use of elitism and the taboo search meta-heuristic, all of which are designed to improve searching, speed, and the quality of the evolved Core War programs.

Results show, that generated programs are able to be evolved properly, and can be more effective than some standard programs when tested in the actual nMars IDE. Improved results can be obtained by improving the speed of execution of the algorithm, through practices such as parallelism, different parsing techniques, and by adjusting the genetic algorithm searching process.

1 INTRODUCTION

1.1 OVERVIEW

The main aim of this dissertation is to evolve program code via evolutionary practices, in order to possibly generate a fitter program for a particular task. In this case, the code is that of a warrior, pertaining to Core War, a programming game, where the aim of the game is for programs, called warriors, to compete against each other in the hopes of terminating each other's process within a set amount of rounds, battles are categorised via a Win/Tie/Loss scale after the rounds are complete. The application built, uses a Genetic Algorithm, to carry out selection, mutation and cross-over processes on a particular genome, in this case, the code itself. Immutable programs (unchanging and non-evolving, pre-defined programs that already exist) are used in order to act as opposing programs to the programs generated by the algorithm and also as a basis for new programs. A fitness function is also used to rate programs, to estimate how effective the Genetic Algorithm is in terminating the immutable program's process in each generation. The programs will duel one another by being loaded as text files into the CMD, which will make use of the nMars.NET Console Application, which contains a Core War compiler, to be able to carry out the battle rounds and save result files for further use within the application for further analysis.

1.2 BENEFITS

The benefits of this dissertation and the research carried out include an analysis and demonstration of how Evolutionary algorithms can attempt to solve programming problems that have no known solution or have very complex, abstract or dynamic solutions, including how well they perform against the given problem.

There are various problems that can be solved using evolutionary algorithms. A few benefits that are brought about are:

- ✓ Distributed systems and distributed computation are vastly improving, which means that future applications of this dissertation may be used to solve far more complex problems, in a different context, by applying similar techniques but utilising more processing power, making use of more than one system to tackle the issue at hand.
- ✓ While still retaining the same scope, the problems that are being solved may change over time, and require a different solution in order to be correctly and efficiently solved. In practice, this might cause the solution that has been built to have to be removed and re-implemented, or involve a lot of restructuring, especially in the case of legacy code, or where the original developer is no longer working on the project. This dissertation will show this by determining how evolutionary algorithms can evolve source code to solve the task at hand, even if the solution to the problem is changed - in this case, by having the generated program battle other programs in addition to the original immutable program.
- ✓ The dissertation also demonstrates how evolutionary algorithms perform self-optimisation techniques via their fitness function. In spite of the fact that the fitness function must be tailored to the problem the algorithm is trying to solve, by substituting the area the evolutionary aspect will be used by another, the same concepts of fitness, whilst modified for the specific application, can be utilised to perform searches in different contexts.

1.3 KEY CHAPTERS

1.3.1 Literature Review

The Literature Review analyses the work carried out in related research, and provides the necessary background for the dissertation. The literature review contains information on genetics, genetic practices applied to algorithms, and the application of genetics in computation.

1.3.2 Methodology

The methodology includes information about the research carried out in this dissertation, and the application implemented. The focus of this information is on the programming concepts and the applied information and research from the literature review, which includes any changes or modifications made, to make the dissertation program function as it should.

1.3.3 Results

In this chapter results and findings are presented and analysed, including comparisons and samples of generated programs and the explanations of these generated programs.

1.3.4 Future Work

This chapter presents, analysis and review conclusions to list any modifications that should be done to the dissertation in the future to improve both efficiency, speed, and result generation.

2 LITERATURE REVIEW

2.1 AN OVERVIEW OF GENETICS

2.1.1 The Genome

The genome, or genetic code, in principle, is said to contain all traces of life, all the way back to the universal common ancestor, this is where the initial genome was formed. Over millennia this initial genome has been passed down to each generation, until the vastness of living things we see today was established. (Jobling, et al., 2014)

The genome itself is a collection of building blocks, which act as a sequential template to allow a particular species to exist, and to perform its intended functions. The genome's sequence, in the case of living things, is made up of DNA[1], is translated to mRNA[2], this changes the production of proteins, causing the formation of a particular species, which is dependent on that same genetic code. When a particular genome is passed down to a new generation, it does not always remain intact, which means that the predeceasing genome will almost always morph using a particular set of processes, namely; splicing, mutation and crossovers, to evolve. After the genomic sequence morphs, its function will change, but due to the fact that there are an abundance of genes to carry out these processes on, within the sequence, changes are normally unnoticeable, unless a drastic mutation occurs, meaning that the sequence itself will still manage to perform the same intended function. Changes are usually noticeable over hundreds of thousands of years. This process is the main reason which causes genetics to involve evolutionary practices that promote diversity.

2.1.2 Genomic Evolution

2.1.2.1 Splicing

Splicing is a form of crossover, which process involves two genomes. It essentially cuts out a part of one genomic sequence, and fills it in with a part of another genomic sequence, in its

[1] DNA: Deoxyribonucleic Acid

[2] mRNA: Messenger Ribonucleic Acid

place. The part of the sequence to be removed is not target specific, which means that any part of the sequence can be redacted, to make room for the receiving genomic sequence part. After the splice is complete, enzymes are used to join the strains, forming a functional genome. (Jobling, et al., 2014)

2.1.2.2 Mutation

Mutations are brought about when errors in changes to the genome occur, or the genome is changed forcibly, from external sources. If mutations go wrong, they can affect the genome negatively and drastically, however, some mutations may actually be beneficial. Mutations can be classified into two groups, namely spontaneous mutations, and induced mutations.

Although uncommon, spontaneous mutations usually occur when the parent strand and the sibling strand slip up in their alignment, causing imperfections in bonding. Induced Mutations on the other hand are in abundance, and are brought about by environmental factors, such as; radiation, pollutants, chemicals, illnesses and so on. (Andersen, 2012)

2.1.2.2.1 Substitution Mutations

Substitution mutations are one of the main types of how genomic sequences are mutated. This form of mutation is where a part of the genomic sequence changes to another value. For instance:

Initial Sequence

A	A	G	C	T	T	G	A	A	T	T	C

Mutated Sequence

A	A	G	C	T	C	G	A	A	T	T	C

The issue here is brought about when proteins scan the gene for mismatches. A protein might decide to not fix the particular part of the sequence that was mutated, but instead decide to fix the receiving gene to allow proper bonding. There is a 50% chance that this error is made, which will translate to the mRNA, and to the proteins that make up the organism, possibly causing physical changes in the organism. (Andersen, 2012)

2.1.2.2.2 Insertion Mutations

Insertion mutations occur when certain situations bring about breakages in the particular sequence, which will leave gaps. In this case, when repairs are made to the sequence, those gaps will need to be fixed by proteins. The issue with this however, is that, in some of the cases when the fix is in progress, the particular sequence might accidentally have another part to it inserted. This will mainly be problematic when a cell comes to replicate, as the strand will most probably shift the sequence over, which could result in a mutated protein. (Andersen, 2012)

Initial Sequence

T	T	C	G	A	A	C	T	T	A	A	G

Mutated Sequence

T	T	C	G	A	A	G	C	T	T	A	A	G

2.1.2.2.3 Deletion Mutations

Nucleotides in DNA may be forcibly omitted for various reasons, most commonly from external sources, such as radiation effects on the organism. When this particular mutation occurs, the sequence has to curl to account for the missing space. When a cell comes to replicate, and copies of parts of the sequence are made, the good part of the curled gene will not cause problems, however the part with the missing nucleotide will if it is selected. (Andersen, 2012)

2.1.3 Genetic Disorders

The aforementioned splicing and mutation processes will surely change the physical structure of a particular organism, over millions of years, normally for the better, to adapt to its environment, and be more competent in said environment. However these processes can also be problematic, as bad genes can be erroneously created, causing the particular organism to develop disabilities, which could possibly and almost surely cause disabilities in any shape or form over a particular generation. Examples of these disorders in humans are; Cancers, Neurofibromatosis, Down syndrome, etc. (National Human Genome Research Institute, 2014)

2.1.4 Survival of the Fittest

Genetic Disorders in organisms will directly affect how fit a particular organism is, in terms of its ability against organisms of the same type, which will also translate to its survival rate against other organisms. This impact also determines the dominance factor of a particular animal or species, due to the fact that, if a particular organism is fitter than another, in most cases, that organism will probably end up being a predator, or if not, exceed the unfit organisms' abilities.

2.2 Different Genetic Algorithm Approaches

2.2.1 Elitism

Genetic algorithms make use of elitist techniques in order to possibly provide a better evolution base for descendant generation mutation and splices. Elitism is the practice of storing the proven, fittest genomes, in memory and carrying them over to the next generation. The elitist selections happen after each iteration, in a particular genetic algorithm framework. Elitism is used to increase the probability of generating fitter genomes, over several generations, by injecting already fit genomes into the same pool. (Leung & Liang, 2003)

2.2.2 Simulated Annealing

When working with large search spaces, simulated annealing applications are used to derive a close to optimal result of a particular function by making use of approximations. Usually, functions making use of simulated annealing utilise a temperature state which acts as a spigot to control how the function operates, in terms of the particular search. For instance, a high temperature would allow a wider search scope, then over generations as the temperature decreases, the search scope will constrict until the close to optimal result is found. (Kirkpatrick, et al., 1983)

2.2.3 Adaptive simulated annealing

Adaptive simulated annealing is an approach derived from the standard simulated annealing algorithm, where the temperature state is modified in relation to the progress of the algorithm itself. In this particular genetic algorithm approach, functions are put in place to determine whether the temperature needs to be increased as well as reduced, which usually depends on the algorithms progress. These functions then modify the temperature accordingly. However, when using the standard simulated annealing approach, the temperature decreases to reduce the amount of changes in the genome in a particular generation as it reaches a peak. Standard simulated annealing is one way, adaptive simulated annealing approaches are two way, constricting in certain parts of the search space and if necessary widening the scope to allow wider searches depending on progress. Due to the fact that adaptive simulated annealing is two way, it may improve searches as the algorithm would be able to make more changes in the genome should it need to.

2.2.4 Taboo Search

The taboo search meta-heuristic was created by Fred W. Glover in 1986. The concept behind this particular practice is to not allow search occurrences that have already been searched for to be carried onto other parts of an algorithm. This is mainly accomplished by storing results that have already been searched for in memory, and before adding new results to that particular pool, the particular result would be checked against the pool. If the particular result has already been searched for, it will not be carried over. (Glover & Laguna, 1997)

2.2.5 Differences between Genetic Algorithms and Evolutionary Algorithms

Evolutionary Algorithms attempt to solve a particular problem my mimicking Darwinian evolution theory, meaning, Charles Darwin's theory, stating that all living organisms evolve by natural selection, and it directly affects an organism's ability to reproduce, compete and survive. Evolutionary algorithms are normally encompassed into three subsets, mainly; Genetic Algorithms, developed by Holland, Evolutionary Programming developed by L.J. Fogel and Evolution Strategies developed by Rechenberg and Schwefel (Jones, n.d.). This means that Genetic Algorithms are a subset of Evolutionary Algorithms, which each have a different approach to solve similar problems but utilising different strategies, in conjunction to making use of different concepts. Genetic Algorithms, use crossover and mutation functions to broaden the search space in a particular scenario, whereas Evolutionary Algorithms are fixed, commonly only allowing mutations to occur. This means that Genetic Algorithms' search spaces promote a hierarchical approach of siblings and parents, whereas Evolutionary Algorithms are random based.

2.3 Applications of Genetic Algorithms

2.3.1 Pancreatic Cancer Diagnosis

Genetic Algorithms have been successfully adapted to diagnose pancreatic cancer. The problem with diagnosing pancreatic cancer is that it cannot be easily diagnosed during its early stages of development, especially adenocarcinoma, which is a particular strain of this cancer. The issue with misdiagnosing or overlooking pancreatic cancer can be potentially aided by facilitating its diagnosis through (Moschopoulos, et al., 2013) particular Genetic Algorithm's results. Genetic algorithms can help in identifying and diagnosing the different types of pancreatic cancer correctly. In this particular case, the information fed to the genetic algorithm consists of a binary array with 19898 bits allocated to Genes, and 14 bits allocated to SVM parameters, making up a total of 19912 bits which can be mapped to a particular human individual's tissue sample. Through the common practices of Darwinian evolution theory, as aforementioned, namely boiled down to Selection, Crossovers and Mutation functions in practice, this Genetic Algorithm managed to yield robust classifiers in detecting pancreatic cancer. In addition to this, the team involved, managed to produce a list of biomarkers which can continue to help facilitate the detection of this particular disease. (Moschopoulos, et al., 2013)

2.3.2 Artificial Creativity

Creativity can be quite difficult to mimic in computer systems, mainly due to the fact that creativity factors in a conscious mind. In a particular experiment, a Genetic Algorithm was formulated in 3DMax, to attempt to come up with a way to mould an initial model, in this case a cube, into something other than that initial cube. The algorithm was allowed to make use of five functions, Taper, Twist, Stretch, Skew and Bend to accomplish this, alongside parameters which effect the intensity of how these functions are utilised. The variety of shapes created via this technique were quite significant, and over twenty generations yielded diverse results, making use of elitism to continue to fuel changes which are fitter according to the specified parameters. Ultimately this particular test shows that Genetic Algorithms can be used to mimic a certain degree of creativity. (Marin, et al., 2008)

2.3.3 Spacecraft Antennae

Designing X-Band antennae by hand usually involves trial and error situations, and requires significant expertise in the field and intense labour and expenses. NASA decided to use Genetic Algorithms in their ST5 mission to develop an efficient X-Band antenna. The Genetic Algorithm, took around four weeks to successfully evolve the initial antenna. In doing so, NASA was able to explore thousands of new designs which were not likely to be explored by experts in the field, as their shapes are particularly random and unusual. This lead to the creation of an antenna with significant performance improvements, that passed the specification tests required for it to meet the requirements for the spacecraft mission.

Improvements include better power, efficiency, performance, wider ranges and angles of transmission, and also significant increases in data throughput. (Hornby, et al., 2006)

2.4 APPLICATIONS OF CODE EVOLUTION USING GENETIC ALGORITHMS

2.4.1 Code Optimisation

When it comes to optimising code, many practices usually cater for speed factors, normally in the way a particular task is executed by the code itself, when compiled. However, other optimisations, such as improving space complexity also exist. Since Genetic Algorithms are quite suitable in terms of their applications in complier optimisations, they may be utilised to find better object based solutions by allowing the implementation to search for smaller object codes, which may result in shorter code, yet still be entirely able to accomplish the same task. This particular genetic algorithm implementation, computes different optimised code solutions over generations, then determines the compiled codes fitness in terms of space complexity. It is important to note that the genetic algorithm approach (other than making use of randomisation alone to accomplish the same task) provides a fast way to optimise code whilst also manages to probe a significantly large search space. This particular technique yielded positive results, which includes dramatic code size reduction, and increased speeds. (Cooper, et al., n.d.)

2.4.2 Automated Source Code Evolution

Genetic algorithms may be applied to generate evolved, compilable source code, as demonstrated by (Miller, 2012) who created an implementation in a particular C++ 11 application. Since this practice can successfully evolve code, it backs the fact that this dissertation is viable. In terms of automated source code evolution, practices such as this may be utilized and adapted to existing software, to allow that software to self-update. This is useful when particular software solutions need to be constantly modified to cater for very frequent changes. It is important to note that user interface based software and client applications could prove to be difficult to evolve and may be impractical in this case, but when it comes to internal system components, this could be completely viable, depending on the system utilizing this approach.

3 METHODOLOGY

3.1 IMPLEMENTATION

3.1.1 Core War

Core War is a programming game in which, compiled programs, known as warriors, battle one another in a sandbox in hopes of terminating each other's process, and surviving long enough to do so. The programs are written in a language derived from assembly, called Redcode (Bhatia, n.d.). The dissertation implementation will utilise the nMars compiler, which is the backbone of the game itself (Šavara, 2007), to pass it generated programs alongside immutable programs. Immutable programs will also be stored on the hard disk, and will act as the warriors to be beaten by the genetic algorithm. The aim in this case is to try to evolve an initial immutable program to the point where it starts to win the most battles out of every generated program including its predecessor, i.e. the initial immutable program.

3.1.2 Genetic Algorithm Framework

The genetic algorithm is an abstraction of a genetic algorithm implementation which will be extended by a concrete genetic algorithm that will utilise core war programs (warriors) as genomes. The genetic algorithm framework consists of classes which form the main genetic algorithm, which include fitness function abstractions, and a settings class which can be used to modify the genetic algorithm's settings, such as; population, new genomes per generation, iterations without improvement, and so on.

3.1.3 Program Environment

The program environment serves as a program backbone, meaning, it contains all required classes to store programs, both in memory and otherwise. The program environment also contains the program core, which will be directly linked to the nMars compiler, allowing programs, after different generations, to battle, consequently changing their ELO Rating, which will help in determining their fitness and viability in the particular pool after each generation.

3.1.4 Elo Rating

Survival of the fittest has to be simulated in order to allow fitness functions to determine the effectiveness of generated programs within the environment. This practice has been proven to be correlated and calculated on dominance hierarchies in animal societies to determine their fitness. (Neumann, et al., 2011)

The developed program makes use of this particular rating concept by utilising a built ELO Rating class that houses methods to calculate new ELO Rating values, depending on the winner of a particular battle. Then the ELO Rating fitness function compares the rating value property in a particular program to determine its effectiveness against other programs within the same pool, causing unfit programs to be removed from the optimal program collection.

It is important to note that before each and every battle, ELO Ratings are set to be equivalent to one another, i.e. a global reset, in the particular pool, which allows the elitist selection of a previous iteration to not affect the next. This way, mutations may still occur freely, and elitist selection will only effect the next generation, but not the entire sequence of generations, as if ratings were not reset, elitism would also happen on the fitness side of things, causing a particular set of genes to constantly appear in each and every generation, rendering most of the search redundant. This way, if programs are truly the best in the particular gene pool they will have to accumulate sufficient rating every generation, to show this. These programs would then be carried on as elitist to the next generation having to undergo the same process.

3.1.5 Gray Code

Gray code is used in computation, where a particular number, in binary, is represented in a one-digit differentiation. Numeric mutations need to be less drastic, as mutation can be difficult to implement via base 10 numeric values, whilst also promoting diversity. In order to make sure that a particular change is relatively small when it comes to mutating, Gray Code conversion mechanisms have been implemented, and work alongside mutation methods, so that numeric mutations occur in the Gray Coded form of the base 10 values. This way, mutation simulation will be gradual, which implies that it will be more realistic, as normally genetic improvements are small, but large across vast generations.

Gray code works in this particular way: for instance, a particular number (such as; 12) would be mutated from a Decimal value, using the Gray Code conversion algorithm, to its Binary value, and then to its Gray Coded value:

$$(12)_{10} \quad \leftrightarrow \quad (1100)_2 \quad \leftrightarrow \quad (1010)_g$$

The Gray Coded value will then be mutated by a random inversion of a particular digit, and then be reverted back to a Binary value, and finally a Decimal value respectively, giving us a randomly mutated number:

$$(1010)_g \quad \leftrightarrow \quad (101\bar{1})_{mg} \quad \leftrightarrow \quad (1101)_2 \quad \leftrightarrow \quad (13)_{10}$$

(Weisstein, n.d.)

3.1.6 Concrete Genetic Algorithm

The genetic algorithm framework implementation is used to mimic the aforementioned genetic processes, and allow the ability for an immutable program to act as an initiator, being a genome, i.e. the Universal Common Ancestor. Extending the genetic algorithm framework to produce these implementations will allow the overriding and modification of how its methods and fitness functions work, to enable the restriction and viability of the search space to only generate compliable programs, which should eventually lead to an evolved, improved version of the initiator.

3.1.6.1 Mutation

Code based mutation can be broken down into different mutators in this particular implementation. These mutators are; Numeric Mutator, Command Mutator, Address Mutator, and Line Mutator, where each and every one of these mutators take care of mutating specific parts of a given programs source code.

- ✓ The numeric mutator may mutate any numeric value found in a particular stream of code where the aforementioned gray code conversion comes into play.
- ✓ The command mutator may substitute any existing commands with the following base CoreWar commands in a particular code stream; DAT, MOV, ADD, SUB, MUL, DIV, MOD, JMP, JMZ, JMN, DJN, CMP, SPL, SEQ, SNE, SLT, XCH, PCT, NOP, STP and LDP.
- ✓ The address mutator is responsible is similar to the code mutator as is used to substitute existing address instructions rather than command values. The values it may substitute to are; #, $, _, @, <, >, *, } and { where in this case, _ is an empty space.
- ✓ The line mutator's job is to come up with new random lines to be added to a particular program, or to remove any amount of lines from a particular program. This is particularly useful in mimicking deletion and insertion mutations in the particular program code.

All the above mutators form a part of a Decorator Design Pattern implementation, which allows for all mutators to work hand in hand with one another to provide the desired overall mutation, depending on the amount of lines allowed to be mutated. It is important to note that, the commands and address instructions used in mutation are that of a parsed core war program. The decision in using these as opposed to the latter was due to the fact that it would be problematic in mutating, as there are many different derivations of how core war programs can be written to be compiled, which would have been too vast to cater for.

3.1.6.2 Splicing

Splicing is accomplished by performing a Cartesian product on all code lines. This means that the two programs fed to the function which performs splicing techniques, are split apart line by line, and all combinations for splicing are then calculated, as splicing in this particular implementation is done by substituting a line from one program into another program and vice versa. When all combinations for splicing are calculated, a random candidate swap is chosen and checked for compilability, if compilable the program code changes are committed and outputted back into the gene pool. If no candidates are found, the programs codes remain as they were initially. This technique has been adopted, due to the fact that randomisation is problematic in this case, as through randomisation alone, a compilable splice may never be found causing the application to get stuck within the function, constantly checking for a candidate, this way when all combinations are checked, the program can return changes or default values accordingly. Even though modifications could bypass the function getting stuck, this particular technique reduced finding compliable splices from approximately 30 seconds to approximately 10. It is however, important to note, that compilable programs are checked via the nMars console application, and that in addition to the randomness aspect, will make times vary.

3.1.6.3 Taboo Search

The taboo search metaheuristic has been applied in this dissertation by storing a list of programs that have already been searched for, as hashed values, and then connecting its implementation to the genetic algorithm itself. By doing so, the algorithm tends to be more efficient in comparison to the latter, as when it comes to use generated programs to compete after each generation, to determine their effectiveness, multiple occurrences of the same program will not compete versus the immutable programs they are trying to defeat. This means that the list of programs will not be polluted by the same program code bases over multiple generations, as if this occurs, programs which have slightly less rating, but are still sufficient in accomplishing the task at hand may be deprecated by the algorithm, causing lack in evolutionary diversity, of which will surely lead to biased results. Not only that, but in practice, since the taboo search collection stores hashed values of program code bases for comparison, in addition to having less programs to battle one another, it also improves performance. This performance improvement mainly comes from not having to open the external nMars console application to initiate battles for the same program, and since generating similar programs can be very common when not making use of this metaheuristic, especially during the initial generation, in addition to the nMars console application having its closings awaited by the genetic algorithm implementation, making use of the taboo search improved speeds drastically.

3.1.6.4 Noteworthy Functions

3.1.6.4.1 Prepare Gene Pool Function

Due to the fact that elitist practices carry on particular fit programs to the next generation, it was realised that these may also pollute the gene pool, as situations may arise where programs with the same code could then be carried on as elitist, and so on. In order to avoid this gradual pollution, the prepare gene pool function was created to take care of this before the gene pool is updated. The way this works is by iterating over every program found in the particular gene pool and then determining whether a program with the same code already exists within it, if similar programs are found, the best rating out of all programs is assigned to one program which is carried on while the rest are deprecated.

4 RESULTS

4.1 OVERVIEW

The programs generated via evolutionary practices in this dissertation have proved to be somewhat effective against the immutable programs that they have been tested against in fitness functions. When outputted the best programs in a particular occurrence have then been copied and moved to the nMars IDE environment where they have been set to compete against their initial program base code, and if the programs are not too complex and are good evolvements, tend to win around 60% of the time against the initial base code. This practice shows that evolved programs are able to hold their ground substantially via the opposing warriors as in certain cases the evolved programs manage to beat other warriors in the particular environment by having all opposing scores 0, whilst the evolved programs win all the battles. It is important to note however, that as of yet there have not been any evolved programs that constantly win, however wins fluctuate depending on the opposing warriors. Evolved programs wins are somewhat consistent and will be demonstrated in the following section.

4.2 GENERATED PROGRAMS

We will initially attempt to evolve a complex, highly rated program, by using the following genetic algorithm settings:

Max Population	**1000**
Elitism Count	**100**
Max Iterations Without Improvement	**5**

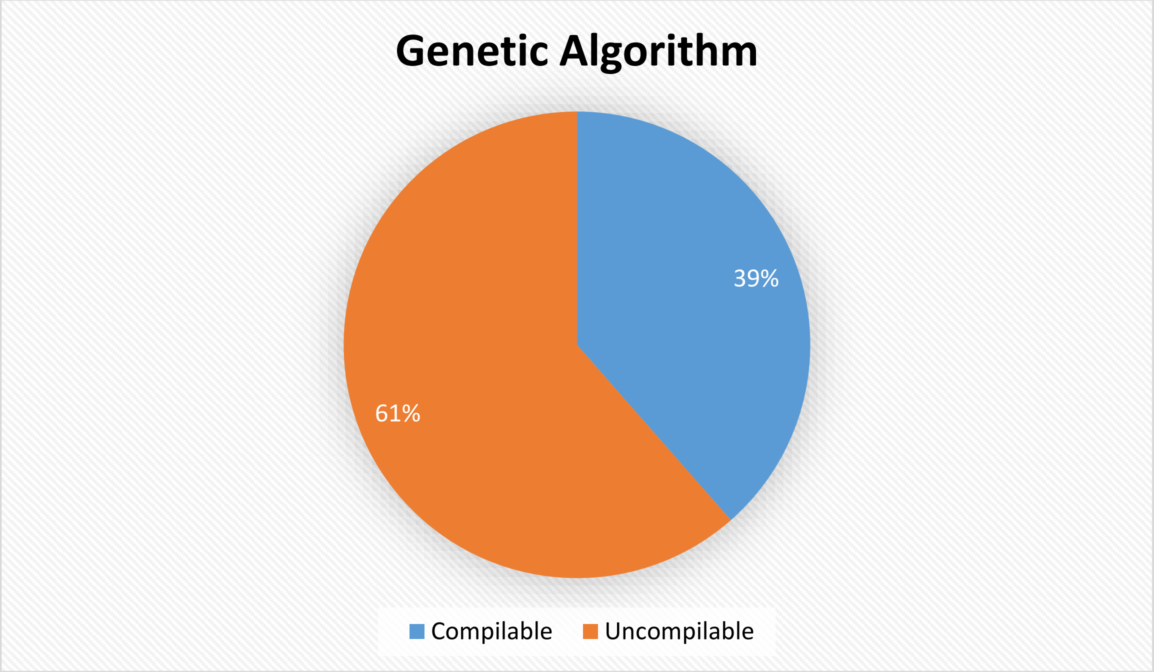
Initial Program	Evolved (Generation 1)	Reverted (Generation 2)	Reverted (Generation 3)
Rating: 1000	**Rating: 1236.71**	**Rating: 1236.71**	**Rating: 1269.51**
ORG START	ORG START	ORG START	ORG START
DAT.F $ 2000, $ 400	ADD # -2997, +7084	DAT.F $ 2000, $ 400	ADD # -2997, +7084
DAT.F $ 800, $ 200	DAT.F $ 800, $ 200	DAT.F $ 800, $ 200	DAT.F $ 800, $ 200
DAT.F $ 4600, $ 600	DAT.F $ 4600, $ 600	DAT.F $ 4600, $ 600	DAT.F $ 4600, $ 600
JMP.B $ 7800, $ 15	JMP.B $ 7800, $ 15	JMP.B $ 7800, $ 15	JMP.B $ 7800, $ 15
DAT.F $ 15, $ 7985	DAT.F $ 15, $ 7985	DAT.F $ 15, $ 7985	DAT.F $ 15, $ 7985
ADD.A $ 7996, $ 7996	ADD.A $ 7996, $ 7996	ADD.A $ 7996, $ 7996	ADD.A $ 7996, $ 7996
ADD.AB @ 7999, $ 5	ADD.AB @ 7999, $ 5	ADD.AB @ 7999, $ 5	ADD.AB @ 7999, $ 5
ADD.B * 7998, @ 7999	ADD.B * 7998, @ 7999	ADD.B * 7998, @ 7999	ADD.B * 7998, @ 7999
SNE.I $ 73, @ 3	SNE.I $ 73, @ 3	SNE.I $ 73, @ 3	SNE.I $ 73, @ 3

ADD.AB # 100, $ 2	ADD.AB # 100, $ 2	ADD.AB # 100, $ 2	ADD.AB # 100, $ 2
MOV.I $ 7993, @ 1	MOV.I $ 7993, @ 1	MOV.I $ 7993, @ 1	MOV.I $ 7993, @ 1
MOV.I $ 7993, @ 407	MOV.I $ 7993, @ 407	MOV.I $ 7993, @ 407	MOV.I $ 7993, @ 407
ADD.BA $ 7999, $ 7999	ADD.BA $ 7999, $ 7999	ADD.BA $ 7999, $ 7999	ADD.BA $ 7999, $ 7999
MOV.I $ 7990, * 7998	MOV.I $ 7990, * 7998	MOV.I $ 7990, * 7998	MOV.I $ 7990, * 7998
ADD.F $ 7990, $ 7997	ADD.F $ 7990, $ 7997	ADD.F $ 7990, $ 7997	ADD.F $ 7990, $ 7997
MOV.I $ 7988, @ 7996	MOV.I $ 7988, @ 7996	MOV.I $ 7988, @ 7996	MOV.I $ 7988, @ 7996
DJN.B $ 7997, # 6	DJN.B $ 7997, # 6	DJN.B $ 7997, # 6	DJN.B $ 7997, # 6
JMP.B $ 53, } 7700	JMP.B $ 53, } 7700	JMP.B $ 53, } 7700	JMP.B $ 53, } 7700
START CMP.I $ 400, $ 500	START CMP.I $ 400, $ 500	START CMP.I $ 400, $ 500	START CMP.I $ 400, $ 500
JMP.B $ 7989, } 2600	JMP.B $ 7989, } 2600	JMP.B $ 7989, } 2600	JMP.B $ 7989, } 2600
CMP.I $ 598, $ 698	CMP.I $ 598, $ 698	CMP.I $ 598, $ 698	CMP.I $ 598, $ 698
JMP.B $ 7986, } 647	JMP.B $ 7986, } 647	JMP.B $ 7986, } 647	JMP.B $ 7986, } 647
CMP.I $ 796, $ 896	CMP.I $ 796, $ 896	CMP.I $ 796, $ 896	CMP.I $ 796, $ 896
JMP.B $ 7984, { 7982	JMP.B $ 7984, { 7982	JMP.B $ 7984, { 7982	JMP.B $ 7984, { 7982
CMP.I $ 994, $ 1094	CMP.I $ 994, $ 1094	CMP.I $ 994, $ 1094	CMP.I $ 994, $ 1094
JMP.B $ 7982, } 7980	JMP.B $ 7982, } 7980	JMP.B $ 7982, } 7980	JMP.B $ 7982, } 7980
CMP.I $ 2992, $ 3092	CMP.I $ 2992, $ 3092	CMP.I $ 2992, $ 3092	CMP.I $ 2992, $ 3092
JMP.B $ 7980, { 7980	JMP.B $ 7980, { 7980	JMP.B $ 7980, { 7980	JMP.B $ 7980, { 7980
CMP.I $ 1190, $ 1290	CMP.I $ 1190, $ 1290	CMP.I $ 1190, $ 1290	CMP.I $ 1190, $ 1290
JMP.B > 7978, } 1239	JMP.B > 7978, } 1239	JMP.B > 7978, } 1239	JMP.B > 7978, } 1239
CMP.I $ 1388, $ 1488	CMP.I $ 1388, $ 1488	CMP.I $ 1388, $ 1488	CMP.I $ 1388, $ 1488
JMP.B $ 7975, } 1437	JMP.B $ 7975, } 1437	JMP.B $ 7975, } 1437	JMP.B $ 7975, } 1437
CMP.I $ 1586, $ 1686	CMP.I $ 1586, $ 1686	CMP.I $ 1586, $ 1686	CMP.I $ 1586, $ 1686
JMP.B $ 7973, { 7972	JMP.B $ 7973, { 7972	JMP.B $ 7973, { 7972	JMP.B $ 7973, { 7972
CMP.I $ 1784, $ 1884	CMP.I $ 1784, $ 1884	CMP.I $ 1784, $ 1884	CMP.I $ 1784, $ 1884
JMP.B $ 7971, } 7970	JMP.B $ 7971, } 7970	JMP.B $ 7971, } 7970	JMP.B $ 7971, } 7970
CMP.I $ 2382, $ 2482	CMP.I $ 2382, $ 2482	CMP.I $ 2382, $ 2482	CMP.I $ 2382, $ 2482
JMP.B > 7970, < 7968	JMP.B > 7970, < 7968	JMP.B > 7970, < 7968	JMP.B > 7970, < 7968
CMP.I $ 2580, $ 2680	CMP.I $ 2580, $ 2680	CMP.I $ 2580, $ 2680	CMP.I $ 2580, $ 2680
JMP.B $ 7967, < 7966	JMP.B $ 7967, < 7966	JMP.B $ 7967, < 7966	JMP.B $ 7967, < 7966
CMP.I $ 2778, $ 2878	CMP.I $ 2778, $ 2878	CMP.I $ 2778, $ 2878	CMP.I $ 2778, $ 2878
DJN.F $ 7965, $ 7964	DJN.F $ 7965, $ 7964	DJN.F $ 7965, $ 7964	DJN.F $ 7965, $ 7964
CMP.I $ 4976, $ 5076	CMP.I $ 4976, $ 5076	CMP.I $ 4976, $ 5076	CMP.I $ 4976, $ 5076
JMP.B > 7964, > 7962	JMP.B > 7964, > 7962	JMP.B > 7964, > 7962	JMP.B > 7964, > 7962
CMP.I $ 5174, $ 5274	CMP.I $ 5174, $ 5274	CMP.I $ 5174, $ 5274	CMP.I $ 5174, $ 5274
JMP.B $ 7961, > 7960	JMP.B $ 7961, > 7960	JMP.B $ 7961, > 7960	JMP.B $ 7961, > 7960
CMP.I $ 3772, $ 3872	CMP.I $ 3772, $ 3872	CMP.I $ 3772, $ 3872	CMP.I $ 3772, $ 3872
JMP.B $ 7959, { 7960	JMP.B $ 7959, { 7960	JMP.B $ 7959, { 7960	JMP.B $ 7959, { 7960
CMP.I $ 1970, $ 2070	CMP.I $ 1970, $ 2070	CMP.I $ 1970, $ 2070	CMP.I $ 1970, $ 2070
JMP.B < 7958, } 2019	JMP.B < 7958, } 2019	JMP.B < 7958, } 2019	JMP.B < 7958, } 2019
CMP.I $ 2168, $ 2268	CMP.I $ 2168, $ 2268	CMP.I $ 2168, $ 2268	CMP.I $ 2168, $ 2268
JMP.B $ 7954, } 2217	JMP.B $ 7954, } 2217	JMP.B $ 7954, } 2217	JMP.B $ 7954, } 2217
CMP.I $ 3366, $ 3466	CMP.I $ 3366, $ 3466	CMP.I $ 3366, $ 3466	CMP.I $ 3366, $ 3466
JMP.B $ 7952, < 7952	JMP.B $ 7952, < 7952	JMP.B $ 7952, < 7952	JMP.B $ 7952, < 7952
CMP.I $ 3564, $ 3664	CMP.I $ 3564, $ 3664	CMP.I $ 3564, $ 3664	CMP.I $ 3564, $ 3664
JMP.B $ 7950, { 7950	JMP.B $ 7950, { 7950	JMP.B $ 7950, { 7950	JMP.B $ 7950, { 7950
CMP.I $ 4362, $ 4462	CMP.I $ 4362, $ 4462	CMP.I $ 4362, $ 4462	CMP.I $ 4362, $ 4462
DJN.F < 7950, $ 7948	DJN.F < 7950, $ 7948	DAT.F $ 2000, $ 400	DJN.F < 7950, $ 7948
CMP.I $ 4560, $ 4660	CMP.I $ 4560, $ 4660	CMP.I $ 4560, $ 4660	CMP.I $ 4560, $ 4660
JMP.B $ 7946, { 7948	JMP.B $ 7946, { 7948	JMP.B $ 7946, { 7948	JMP.B $ 7946, { 7948
CMP.I $ 4758, $ 4858	CMP.I $ 4758, $ 4858	CMP.I $ 4758, $ 4858	CMP.I $ 4758, $ 4858
DJN.F $ 7944, $ 7944	DJN.F $ 7944, $ 7944	DJN.F $ 7944, $ 7944	DJN.F $ 7944, $ 7944
CMP.I $ 5756, $ 5856	CMP.I $ 5756, $ 5856	CMP.I $ 5756, $ 5856	CMP.I $ 5756, $ 5856
JMP.B < 7944, > 7942	JMP.B < 7944, > 7942	JMP.B < 7944, > 7942	JMP.B < 7944, > 7942
CMP.I $ 5954, $ 6054	CMP.I $ 5954, $ 6054	CMP.I $ 5954, $ 6054	CMP.I $ 5954, $ 6054
JMP.B $ 7940, > 7940	JMP.B $ 7940, > 7940	JMP.B $ 7940, > 7940	JMP.B $ 7940, > 7940
CMP.I $ 6352, $ 6452	CMP.I $ 6352, $ 6452	CMP.I $ 6352, $ 6452	CMP.I $ 6352, $ 6452
JMP.B $ 7938, } 7938	JMP.B $ 7938, } 7938	JMP.B $ 7938, } 7938	JMP.B $ 7938, } 7938
JMP.B $ 2, $ 2	JMP.B $ 2, $ 2	JMP.B $ 2, $ 2	JMP.B $ 2, $ 2
SUB.F $ 11, $ 1	SUB.F $ 11, $ 1	SUB.F $ 11, $ 1	SUB.F $ 11, $ 1
CMP.I $ 125, $ 113	CMP.I $ 125, $ 113	CMP.I $ 125, $ 113	CMP.I $ 125, $ 113
SLT.A # 24, $ 7999	SLT.A # 24, $ 7999	SLT.A # 24, $ 7999	SLT.A # 24, $ 7999
DJN.F $ 7997, < 7692	DJN.F $ 7997, < 7692	DJN.F $ 7997, < 7692	DJN.F $ 7997, < 7692
MOV.AB # 14, $ 2	MOV.AB # 14, $ 2	MOV.AB # 14, $ 2	MOV.AB # 14, $ 2
MOV.I $ 4, > 7996	MOV.I $ 4, > 7996	MOV.I $ 4, > 7996	MOV.I $ 4, > 7996
DJN.B $ 7999, # 0	DJN.B $ 7999, # 0	DJN.B $ 7999, # 0	DJN.B $ 7999, # 0
SUB.AB # 14, $ 7994	SUB.AB # 14, $ 7994	SUB.AB # 14, $ 7994	SUB.AB # 14, $ 7994
JMN.B $ 7992, $ 7992	JMN.B $ 7992, $ 7992	JMN.B $ 7992, $ 7992	JMN.B $ 7992, $ 7992
SPL.A $ 0, $ 0	SPL.A $ 0, $ 0	SPL.A $ 0, $ 0	SPL.A $ 0, $ 0
MOV.I $ 1, < 7996	MOV.I $ 1, < 7996	MOV.I $ 1, < 7996	MOV.I $ 1, < 7996

DAT.F < 7958, < 7958	DAT.F < 7958, < 7958	DAT.F < 7958, < 7958	DAT.F < 7958, < 7958
DAT.F $ 0, $ 0	DAT.F $ 0, $ 0	DAT.F $ 0, $ 0	DAT.F $ 0, $ 0
DAT.F $ 0, $ 0	DAT.F $ 0, $ 0	DAT.F $ 0, $ 0	DAT.F $ 0, $ 0
DAT.F $ 0, $ 0	DAT.F $ 0, $ 0	DAT.F $ 0, $ 0	DAT.F $ 0, $ 0

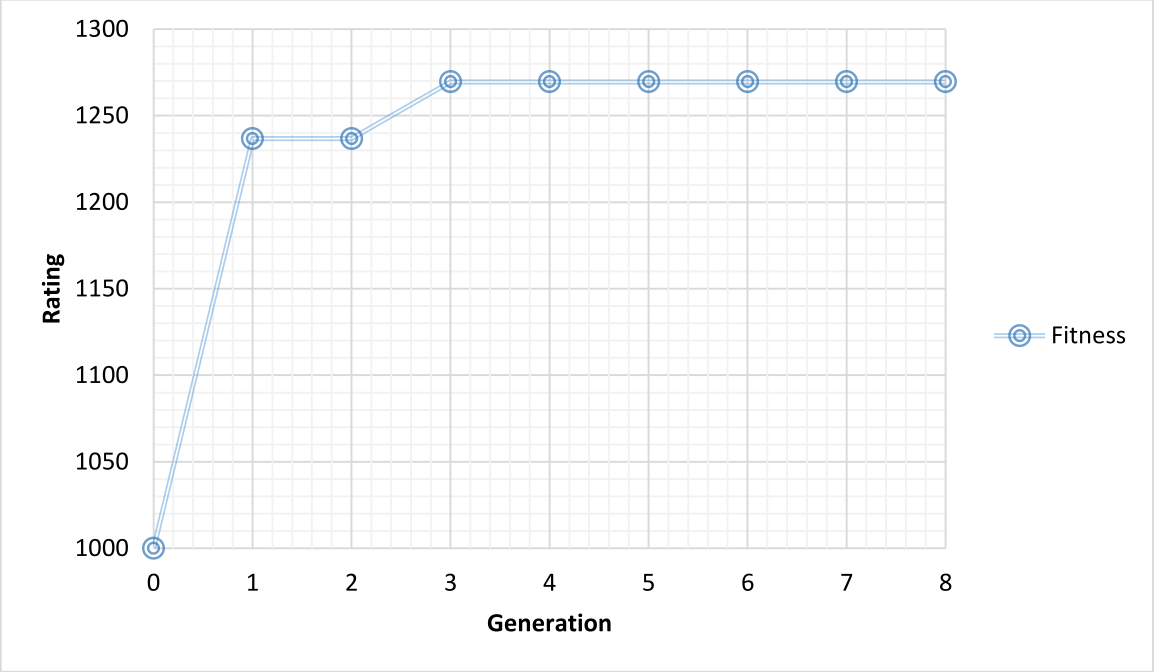
The findings above, lead us to believe that it is harder to evolve more complex programs, considering the fact that the initial program’s evolved counterpart in generation three reverted back to the initial program in generation four. This is probably due to the fact that these programs normally have quite a substantial amount of thought behind them, and may be harder to find improvements on. It is important to note that in addition to this, the above program, is a world top-mid range program, which means that finding an evolved version of it could take a lot more time, ideally by also increasing the maximum amount of generations without improvement and population sizes. When it comes to the rating of the programs, the initial program obtained 1236.71 as a rating, and then increased over generations, this fluctuation, in this case an increase, is brought about because there is a certain factor of randomisation involved, as wins, ties and losses may differ even against the same immutable program a generated program is battling against, which is why the revert occurred in the first place. In this case it so happens that less battles were won at earlier stages, however this has nothing to do with the genetic algorithm implementation, as these ratings are purely an interpretation of Core War results. It is also important to note that since the reverted program and the evolved program have the same rating, it is highly likely that they will have close to equal performance in comparison to each other.

The following chart shows a comparison of new programs and uncompilable programs generated throughout all genetic algorithm generations in this particular case.



It is important to note, that the lengthier the size of the program fed to the genetic algorithm, then the more uncompilable programs are generated, particularly within the first generation, but not limited to. The ratio for compilable to uncompilable program generation in the first generation, is around 1:20, where 1 is a compilable program and 20 is an uncompilable program. However by the end of the genetic algorithms lifetime, as shown in the chart above, this ratio changes slightly. The genetic algorithm stopped at generation 8 whilst battling programs for 71,052 times. This whole process took approximately 12 hours.

The following scatter chart shows the gradual growth of fitness across all generations in this particular occurrence.



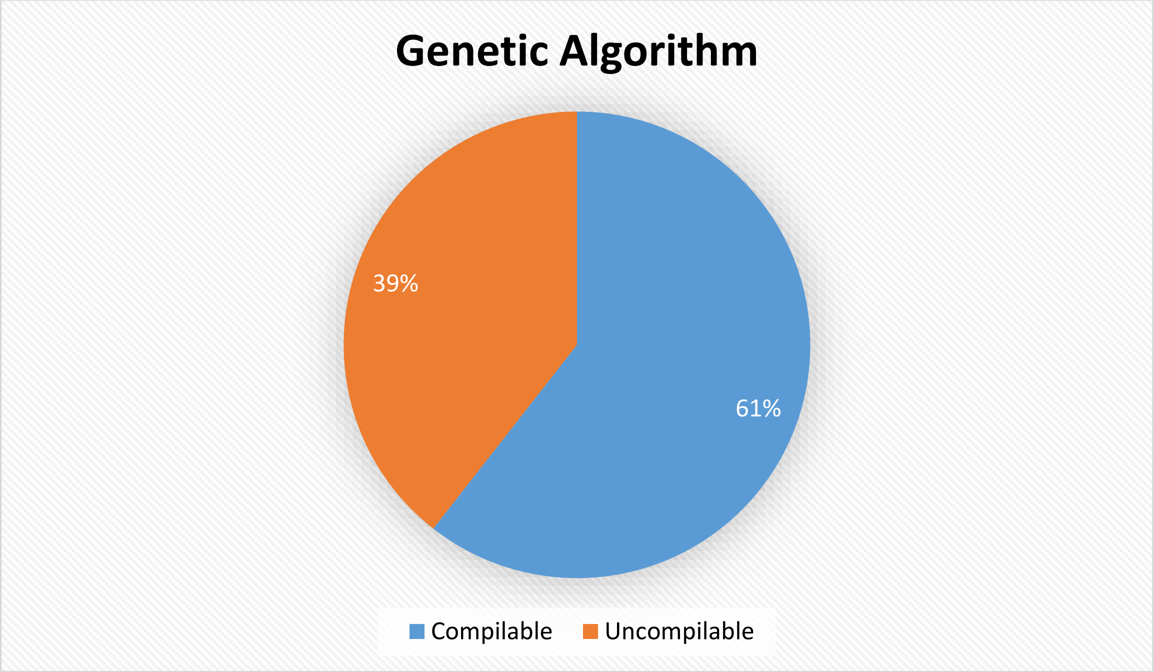
In a different experiment, more basic programs were used, both as an initiator and also as immutable programs, using the same genetic algorithm settings:

Max Population	**1000**
Elitism Count	**100**
Max Iterations Without Improvement	**5**

Initial Program	**Evolved (Generation 1)**	**Evolved (Generation 2)**
Rating: 1000	**Rating: 1078.29**	**Rating: 1136.47**
ORG START	ORG START	SLT > +2788, * +1060
START ADD.AB # 4, $ 3	START ADD.AB # 4, $ 3	START ADD.AB # 4, $ 3
MOV.I $ 2, @ 2	SUB < -4728, { +5406	MOV.I $ 2, @ 2
JMP.B $ 7998, $ 0	JMP.B $ 7998, $ 0	JMP.B $ 7998, $ 0
DAT.F # 0, # 0	DAT.F # 0, # 0	DAT.F # 0, # 0

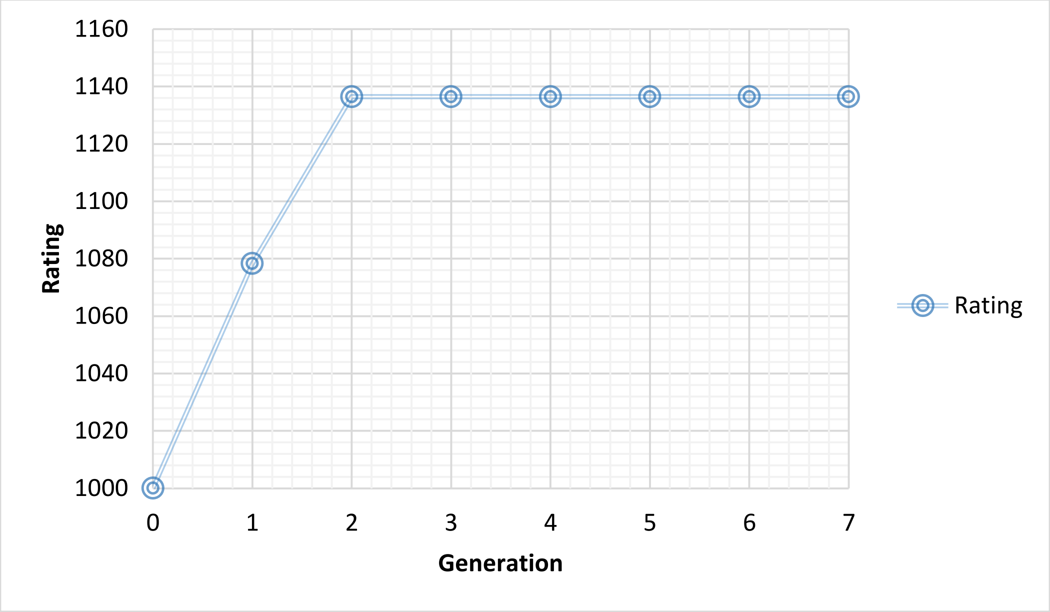
The initial program in this occurrence evolved twice, each by swapping out different lines from the initial program producing quite a significant rating increase. The genetic algorithm stopped at generation 7 whilst battling programs for 39,744 times. This whole process took approximately 8 hours.

In order to compare the compilable to uncompilable ratios present after successfully evolving a shorter, more basic, program, data has been collected to display the following pie chart:



It is immediately noticeable that when evolving shorter programs, the compilable ratio increases to the point where it supersedes uncompilable programs. This, of course could involve many factors such as randomisation, however, since randomisation is carried out over the same amount of times as the previous attempt at evolving a program it is highly unlikely that it would affect it by a large margin such as the above 22%, which means that it is probably due to the size of the program being evolved.

As is done with the previous program, the following scatter chart shows the gradual growth of fitness across all generations in this particular occurrence:



Since the evolved base code differs from the initial program, the initial program and the last evolved program in generation 2 have been battled against one another over 10 times in the nMars IDE, the following is the console result:

```
Parsing: C:\Users\Michael\Documents\Evolved.red
Parsing: C:\Users\Michael\Documents\NewWarrior2.red
========== Compiled 2 warriors, 0 failed ==========
NewWarrior2 by Anonymous scores 0
Evolved by Anonymous scores 3
Results: 0 1 0
========== Finished fight of 2 warriors ==========
Parsing: C:\Users\Michael\Documents\Evolved.red
Parsing: C:\Users\Michael\Documents\NewWarrior2.red
========== Compiled 2 warriors, 0 failed ==========
Evolved by Anonymous scores 0
NewWarrior2 by Anonymous scores 3
Results: 0 1 0
```

```
========== Finished fight of 2 warriors ==========
Parsing: C:\Users\Michael\Documents\Evolved.red
Parsing: C:\Users\Michael\Documents\NewWarrior2.red
========== Compiled 2 warriors, 0 failed ==========
NewWarrior2 by Anonymous scores 0
Evolved by Anonymous scores 3
Results: 0 1 0
========== Finished fight of 2 warriors ==========
Parsing: C:\Users\Michael\Documents\Evolved.red
Parsing: C:\Users\Michael\Documents\NewWarrior2.red
========== Compiled 2 warriors, 0 failed ==========
NewWarrior2 by Anonymous scores 0
Evolved by Anonymous scores 3
Results: 0 1 0
========== Finished fight of 2 warriors ==========
Parsing: C:\Users\Michael\Documents\Evolved.red
Parsing: C:\Users\Michael\Documents\NewWarrior2.red
========== Compiled 2 warriors, 0 failed ==========
Evolved by Anonymous scores 0
NewWarrior2 by Anonymous scores 3
Results: 0 1 0
========== Finished fight of 2 warriors ==========
Parsing: C:\Users\Michael\Documents\Evolved.red
Parsing: C:\Users\Michael\Documents\NewWarrior2.red
========== Compiled 2 warriors, 0 failed ==========
NewWarrior2 by Anonymous scores 0
Evolved by Anonymous scores 3
Results: 0 1 0
========== Finished fight of 2 warriors ==========
Parsing: C:\Users\Michael\Documents\Evolved.red
Parsing: C:\Users\Michael\Documents\NewWarrior2.red
========== Compiled 2 warriors, 0 failed ==========
Evolved by Anonymous scores 0
```

```
NewWarrior2 by Anonymous scores 3
Results: 0 1 0
========== Finished fight of 2 warriors ==========
Parsing: C:\Users\Michael\Documents\Evolved.red
Parsing: C:\Users\Michael\Documents\NewWarrior2.red
========== Compiled 2 warriors, 0 failed ==========
NewWarrior2 by Anonymous scores 0
Evolved by Anonymous scores 3
Results: 0 1 0
========== Finished fight of 2 warriors ==========
Parsing: C:\Users\Michael\Documents\Evolved.red
Parsing: C:\Users\Michael\Documents\NewWarrior2.red
========== Compiled 2 warriors, 0 failed ==========
NewWarrior2 by Anonymous scores 0
Evolved by Anonymous scores 3
Results: 0 1 0
========== Finished fight of 2 warriors ==========
Parsing: C:\Users\Michael\Documents\Evolved.red
Parsing: C:\Users\Michael\Documents\NewWarrior2.red
========== Compiled 2 warriors, 0 failed ==========
NewWarrior2 by Anonymous scores 0
Evolved by Anonymous scores 3
Results: 0 1 0
========== Finished fight of 2 warriors ==========
```

5 FUTURE WORK

5.1 PERFORMANCE IMPROVEMENTS

5.1.1 Utilising nMars Libraries

The current genetic algorithm implementation utilises the nMars compiler. Interaction with this compiler is being done through the nMars compiler's console application, which allows parameters to be passed through to it to compile code and initiate battles. Since the nMars compiler itself does not allow the outputting of resultant data to a text file, for use in fitness functions, the approach to utilise the nMars compiler via C# code, had to be done via another step, the Windows CMD. Due to the fact that the genetic algorithm implementation, opens the CMD, which then opens the nMars compiler and passes in the required programs to battle, and then the battle results are outputted to a text file every time a battle occurs, the search slows down drastically, since the process will have to be halted until the file is written and results are processed. In order to improve performance, direct use of the nMars compiler's dynamic link libraries, instead of using the console application approach, will drastically increase performance, due to the fact that no hard disk read/writes will be necessary and the entire genetic algorithms searching could be done entirely in memory. Further performance improvements would also be obtained since no other programs would be required to be opened and awaited before the searching can continue, such as the launching of the nMars compiler console application to determine whether a particular program is compilable or not.

5.1.2 Paralellism

At this point, the genetic algorithm implementation is single threaded and does not make use of multithreading or parallelism. Considering the fact that multithreading does not necessarily mean utilising multiple processor cores, but instead time slicing, parallelism would most likely, if not definitely, increase the genetic algorithm speed. This is due to the fact that parallelism splits the work up on multiple CPU cores. Making use of the Task Parallel Library contained within the .NET Framework, allows parallelism to not only take advantage of the amount of cores of the system, but also utilises them to their maximum potential drastically increasing speeds. Implementing this improvement would have to start by improving the genetic algorithm framework, and then consequently implementing parallelism throughout the entire project, making use of locks and volatile variables where required to avoid object double access and inconsistencies whilst enforcing thread safety.

5.1.3 Function Improvement

Functions may be optimised and have their code and implementations reworked to possibly increase speed and efficiency. In particular a function that could be optimised is the Prepare Gene Pool Function (see Methodology). Since this particular function searches for similar code bases, by comparing each pair of programs together, and then removes all instances with that same code base but the one with the highest rating, the order of complexity of this function is $O(n^2)$. In the current study, this time taken by this function to execute is acceptable since the gene pool has a maximum size of 1000 genomes. In larger settings however, it may prove to take too much time and this method should ideally be reworked, by possibly taking a different approach. An idea could be to possibly utilise some form of data structure, which would store each programs state in memory, this way these types of searches would not be required to be made.

5.2 Result Improvements

5.2.1 Tweaking Genetic Algorithm Settings

The gene pool size (1000), and the amount of iterations without improvement (5), in the current implementation takes the genetic algorithm around eight hours to complete and find the best program across all generations. Increasing these settings could generate better programs, as there would be a wider range of programs in the population, in addition to allowing the genetic algorithm to continue searching for longer after no improvement has been found. Currently, producing results takes a substantial amount of time and it would make sense to utilise libraries instead of launch processes via the nMars compiler each and every time. Larger programs will of course still take longer to evolve over generations, due to the combinatory processes involved both in Cartesian product used in addition to its random selection and program compilability occurrences.

5.2.2 Different Genetic Algorithms

Different genetic algorithm implementations have been researched, and are analysed in the literature review. Implementing different genetic algorithms could improve results and might generate better programs. For instance, a possible alternate implementation is based on adaptive simulated annealing genetic algorithm in place of the current simulated annealing. While the temperature in the simulated annealing is monotonic decreasing, the temperature in the adaptive simulated annealing algorithm can increase and decrease, allowing for a wider search that improves the chances of generating better programs. Experimenting with different genetic algorithm implementations could also shed light on which algorithm is the best algorithm overall in solving such a problem.

6 CONCLUSIONS

6.1 INFORMATION YIELDED

The most important information that this dissertation has yielded is the confirmation that genetic algorithms and evolutionary programming practices have the possibility of being very effective in evolving program source code. This dissertation has also shed a great deal of insight on how evolutionary algorithms can be adapted to different situations, particularly in the sense of different algorithms available and their characteristics, in addition to tweaks and approaches that can be made to allow such algorithms to cater for the abstract problems at hand, considering the fact that a genome can be of any type. Through the dissertation's research, comparison of different techniques has also paved the way for building a genetic algorithm suitable for the task at hand, where these techniques have also been aided by justifications and assertions on practices that work, against practices which do not work as well in cases where genetic algorithms may be utilised. In addition to the above, the dissertation's program has also been able to generate evolved programs, which through them, has allowed performing result analysis, shedding insight on what needs to be improved to enable better evolved programs, speed and performance.

6.2 RESULT SUMMARISATION

The results generated via the genetic algorithm appear to indicate that source code can be effectively evolved. In addition to this it appears that very efficient programs as initiators do not evolve very well as opposed to those programs which are not that suited or good. Results have also shown that there appear to be less uncompilable programs generated if the program to be evolved is smaller. Considering the fact that the peak has been found within the first three generations, as per the results, it could be useful to make use of adaptive simulated annealing practices to better these particular searches.

6.3 RESULT USAGE

Since these results confirm that program source code can be evolved effectively via genetic algorithms, individuals can then utilise similar techniques to attempt to evolve programs that pertain to a particular task, or are of similar nature, to accomplish various specific, real life tasks. These results may also aid individuals when to creating their own genetic algorithm based optimisation techniques on source code and adding fitness functions for optimisation purposes, such as code length versus efficiency, and so on.

7 REFERENCES

Andersen, P., 2012. *Mutations - YouTube.* [Online]
Available at: http://www.youtube.com/watch?v=eDbK0cxKKsk
[Accessed 9 February 2014].

Bhatia, S., n.d. *Core War.* [Online]
Available at: http://corewars.org/
[Accessed 3 March 2014].

Cooper, K. D., Schielke, P. J. & Subramanian, D., n.d. Optimising for Reduced Code Space using Genetic Algorithms. In: Houston, Texas, USA: Department of Computer Science, Rice University.

Glover, F. & Laguna, M., 1997. *Tabu Search.* Boston: Kluwer Academic Publishers.

Hornby, G. S., Globus, A., Linden, D. S. & Lohn, J. D., 2006. Automated Antenna Design with Evolutionary Algorithms.

Jobling, M. et al., 2014. *Human Evolutionary Genetics.* 2nd ed. New York: Garland Science, Taylor & Francis Group, LLC.

Jones, G., n.d. *Genetic and Evolutionary Algorithms,* Sheffield: University of Sheffield.

Kirkpatrick, S., Gelatt, C. D. & Vecchi, M. P., 1983. Optimization by Simulated Annealing. *Science, New Series,* 220(4598), pp. 671-680.

Leung, K.-S. & Liang, Y., 2003. *Adaptive Elitist-Population Based Genetic Algorithm for Multimodal Function Optimization,* Shatin, N.T., Hong Kong: Department of Computer Science & Engineering, The Chinese University of Hong Kong.

Marin, P., Bignon, J.-C. & Lequay, H., 2008. A Genetic Algorithm for use in Creative Design Processes.

Miller, D., 2012. *Genetic Algorithms for Automated Source Code Evolution: a C++11 tutorial | Dave Miller Blog.* [Online]
Available at: http://www.millermattson.com/dave/?p=174
[Accessed 13 May 2014].

Moschopoulos, C. et al., 2013. A Genetic Algorithm for Pancreatic Cancer Diagnosis. *Springer,* pp. 222-230.

National Human Genome Research Institute, 2014. *Chromosome Abnormalities Fact Sheet.* [Online]
Available at: http://www.genome.gov/11508982
[Accessed 3 March 2014].

National Human Genome Research Institute, 2014. *Specific Genetic Disorders.* [Online]
Available at: http://www.genome.gov/10001204
[Accessed 3 March 2014].

Neumann, C. et al., 2011. Assessing dominance hierarchies: validation and advantages of progressive evaluation with Elo-rating. *Animal Behaviour,* pp. 1-11.

Šavara, P., 2007. *nMars - Core War MARS for .NET.* [Online]
Available at: http://nmars.sourceforge.net/
[Accessed 3 March 2014].

Weisstein, E. W., n.d. *Gray Code -- from Wolfram MathWorld.* [Online]
Available at: http://mathworld.wolfram.com/GrayCode.html
[Accessed 9 March 2014].