6CCS3PRJ Final Year
A Genetic Approach To Solving SAT

Final Project Report

Author: Nam Ta

Supervisor: Hana Chockler

Student ID: 1737839

Programme of Study: BSc Computer Science With A Year In Industry

April 9, 2021

Abstract

At present, SAT solvers are used to a great extent in a wide range of areas. And in particular, have helped push the field of formal verification and allowed for much more efficient bounded model checking. However, with SAT being an NP-complete problem, we have yet to find a truly *fast* solution, and there have been many different approaches from people across the globe. This report explores the relatively recently made popular genetic approach to SAT solving. We will investigate the various ways that each step in the process of evolution has been taken, adapted for the case of SAT solving, and then slotted into the overall algorithm to form the SAT solver. Throughout the investigation, we will attempt to find the optimal combination of techniques and implementations to create the most effective genetic SAT solver whilst appreciating the trickiness of doing so in a stochastic SAT solver such as this, and determine the balance between making sure we find a solution, with finding one in an appropriate time.

Originality Avowal

Acknowledgements

I would like to thank my supervisor Dr. Hana Chockler for her support and kindness throughout the project.

Contents

Chapter 1

Introduction

The Boolean satisfiability problem, shortened as SAT, is a decision problem in which we are given a propositional logic formula and asked what is the combination of truth values to the variables in the formula that then satisfy said formula if it is indeed satisfiable.

1.1 Motivation

1.1.1 The Importance of SAT

The discovery of SAT as an NP-Complete problem in 1971 by Stephen Cook and his subsequent proof that any NP-Complete problem could be converted in polynomial time to SAT [29] had a significant impact on the field of computer science and in industry. Through these proofs, if we can prove also that we can solve SAT in deterministic polynomial time, then consequently every NP problem can be solved in deterministic polynomial time. This is known as the P=NP problem. On one hand, the advantages this could bring to society are countless: from extremely efficient transportation scheduling to much more accurate predictions of weather phenomena [10]. On the other, if we could make any NP-optimisation problem easy to solve, it could have a disastrous impact on the field of Cryptography [20]. Many cryptographic systems such as RSA rely on the computational complexity of algorithms in NP, and so solving P=NP would make RSA undemanding to crack through brute force which could lead to confidential data being at risk. And as of yet, we are not set to come up with a proof separating P from NP any time soon [10]. But although we have not proven the separation of the two, we have also not proven P=NP. Despite this, we can still use SAT to effectively solve many NP-optimisation problems.

Nowadays, the applications of SAT for this problem are wide and varied, from model checking in software verification to planning in AI [25][9]. For example in software verification, we can use SAT for model checking, which is an automated verification method for the analysis of software or hardware systems. SAT has allowed us to model check systems with the number of states being higher than ever before [17], increasing the confidence we can have in our systems. The importance of verifying our hardware and software is tremendous. In 2016, there was over $1 trillion in damage to the worldwide economy due to software failures let alone hardware failures [3].

1.1.2 The Need for Efficient SAT Solvers

However, with time came the increase in complexity of domains (more constraints and variables), which heavily increased the search difficulty [33]. Being an NP-complete problem means that the time for computers to solve the problem grows exponentially on the number of inputs. So for model checking, the states needing to be checked for us to have confidence in the system grow at an incredible rate. From this, the necessity of creating efficient SAT solvers became of great importance, and the progress of SAT solvers has been monumental over the past 20 years [33]. This progress has been accelerated by SAT competitions which have brought to existence many novel approaches and contributed to ensuring the research area of SAT remains active [19]. At present, there are many different implementations of the SAT solver, each being tuned to be the most effective in their given instances, and some of these have incorporated the use of genetic algorithms.

1.2 A Genetic Algorithm

A genetic algorithm is a search heuristic that is developed based on Charles Darwin's theory of natural selection, with the general principle being that only the fittest individuals will be able to reproduce, passing on their genes to their offspring, which in turn create a more desirable population at each generation [24]. Genetic algorithms are incredibly powerful due to their ability to be seamlessly applied to increasingly complex problems. This is because genetic algorithms need not know the inner workings of the problem [15]. Furthermore, another key selling point of the genetic algorithm compared to other stochastic solvers is that the chance of the algorithm becoming 'stuck' at a sub-optimal solution can be lower due to its characteristics of mutation and crossover, meaning for SAT, the chances of it wrongly concluding the problem unsatisfiable is lower.

As just mentioned, a stochastic solver such as one that is based on genetic algorithms can wrongly conclude a problem unsatisfiable. This is the main difference between a stochastic solver and the more common DPLL based solver which is complete and will always provide the correct answer on the satisfiability of a problem if given enough time. On the other hand, a key advantage of using a genetic algorithm is the ability to easily adjust parameters such as the mutation rate and crossover rate to our needs, which can lead to very different results [1]. But the main benefit is if a solution does exist, in general, the genetic-based SAT solver is faster at finding a satisfying assignment due to it involving local computations and not having to systematically cover the entire search space [32][13]. This is especially important due to SAT being a combinatorial problem, leading it to have an exponential worst-case time complexity [12].

1.3 Aims and Scope

The aim of this project is not to prove P=NP or P≠NP. Many complexity theorists believe that P=NP does not even exist [10]. Furthermore, this problem has become one of the most fundamental mathematical questions of our time and its importance continues to grow as computers become increasingly faster. This project is not aimed at proving one or the other.

Moreover, our project's main objective was not to create a SAT solver designed to be used in industry which have been created by experts in the field and are highly-optimised to their specific applications.

Rather, we will analyse the different approaches that have been taken to implement SAT solvers, in particular ones that are based on the genetic algorithm. By comparing these approaches we will strive to capture the best combination of approaches which could influence the configuration of other SAT solvers.

Chapter 2

Background

2.1 A Satisfying Solution to SAT

To make the process of finding a solution more clear, we will first give an example of what a valid solution to a potential SAT problem could be. For instance, the following SAT problem which consists of 3 clauses and 5 different variables:

$$(x136 \vee x138 \vee -x137) \wedge (x135 \vee -x136 \vee x139) \wedge (x136)$$

Translating this formula to the English language would be 'variables $x136$ or $x138$ or NOT $x137$ have to be true AND variables $x135$ or NOT $x136$ or $x139$ have to be true AND variable $x136$ has to be true' for the formula to be satisfied. So for each clause, only one variable in that clause has to be satisfied for the whole clause to be satisfied. A problem with these few variables amounts to a very small search space ($2^n = 32, where\ n = number\ of\ variables$) which makes it possible to find a satisfying solution to this particular problem doable by hand. A possible solution to this would be $x135 = true, x136 = false, x137 = true, x138 = true, x139 = true$. In fact, there are many satisfying assignments for this problem. However, in reality, we deal with much bigger search spaces, and thus the need for sophisticated SAT solvers arrives.

2.2 Different Instances of SAT

As mentioned earlier, in industry SAT solvers are tuned to their specific instances to become as efficient as possible for their use. Given pre-existing knowledge of a SAT problem can help greatly in deciding the approach to take in solving the SAT problem. For example, if the given SAT instances were randomly generated balanced instances or heavily imbalanced instances, we can choose how we assign the variables to increase the probability a clause will be satisfied. In the case of being given an unsatisfiable instance, because a genetic algorithm is a local search algorithm which is incomplete, then using a genetic-based SAT solver is futile. We cannot prove a SAT problem to be unsatisfiable using a genetic approach SAT solver [8].

On the other hand, if we know what the solution may look like, we can use this knowledge as a key advantage in our genetic algorithm SAT solver. We can select the assignments to variables that are most likely to be in the solution in the step of initializing our population which could greatly decrease the time it takes for our algorithm to converge to a satisfiable solution [8]. Another key advantage is that relative to other SAT solvers, a SAT solver such as ours will often solve randomly-generated instances of SAT more effectively. This is because other SAT solvers that are highly specialised for their specific instances will gain no advantage if the SAT instances have no particular structure.

2.3 The Workings of a Genetic Algorithm

Understanding each step of the genetic algorithm is key to taking full advantage of the methodology. There are 6 key steps in an established genetic algorithm including the step of checking whether our solution is valid, as detailed below:

1. We first initialise the population of chromosomes representing possible solutions. This is the first part where we feel that genetic algorithms are made to solve SAT problems. For SAT, we can represent chromosomes as a binary string of length n, where each bit signifies the assignment of a variable (the chromosome's gene) in our problem. For example, we can represent an initial solution to a SAT problem containing 12 variables as below:

   000100010001

   where the highlighted bit shows variable $x4$ having a truth value of true.

2. Our next step is to assign fitness values to each of our chromosomes. This is generally based on how close the chromosome is to a satisfying solution. For instance, it could be the number of clauses satisfied by the chromosome. This would make it extremely easy to know when we have a satisfying solution as this would be when the fitness value is equal to the number of clauses.

3. Then we go onto selection and crossover. This is how offspring are created that then form our new population. A common way to select parents is to select them with a certain percentage chance depending on their fitness score so that the more fit chromosomes have a higher chance of becoming parents. Then we can choose whether to clone a certain proportion of the current population into the new generation (controlled by the elitism rate we give), apply crossover on the chosen parents to create new offspring, or both. Crossover is simply how we can combine two parents to make an offspring. This can be accomplished by truncating and stitching together these two chromosomes (very easy in our case as the chromosomes are simply binary strings). Crossover is a very important step because it allows us to search a much wider search space due to it creating a new chromosome that could potentially be very different from the rest of the population.

4. The next stage is the mutation of our chromosomes. This is another way we can create a new solution. Similar to the real world, the genes in the chromosomes for our problem will also have a small chance to mutate. The result of this is a bit of the solution being

flipped, meaning that the value of one of the variables goes from false to truth and vice versa.

5. Now we have our new generation of chromosomes, we check if any of our chromosomes is a satisfying solution to our SAT problem. If it is, we can simply terminate the program as we have found the solution, but if not, the process restarts from step 2 - the fitness assignment stage. This looping back to stage 2 will repeat until the satisfying assignment is found or the time taken to solve the problem has gone past a set threshold.

2.4 Choosing the Right Rate of Convergence

In genetic algorithms, the key obstacle that continuously appears is the issue of premature convergence. Premature convergence is when the algorithm becomes stuck at a local optima when the rate of convergence is too fast due to the lack of diversity in the population, which unfortunately could lead us to think that the problem is unsatisfiable. We can help solve this problem by increasing the diversity at each stage. One way we could do this is by increasing the mutation rate (provided it is not a greedy implementation of mutation), which would allow for more frequent mutations to take place. Oppositely, a too slow rate of convergence may cause our SAT solver to become very inefficient and take a much longer time than required to solve even simple problems. A key process that will be useful in giving the algorithm chances to escape from local optima albeit not improving and possibly even decreasing the chromosome's fitness score is called adding 'noise' [32], and there are many ways to do this such as the previously mentioned mutation. Further on in this report, this key aspect of genetic algorithms will always play a key role when deciding on the implementation to use.

2.5 Knowing When to Move to a New Population

In our effort to escape premature convergence, a random mutation may not be enough. We may have diverged too far from a satisfying solution. Our last resort would be to discard our current set of solutions and start afresh with a new population of randomly generated solutions. The key question here would be: ”How do we know that we have diverged so far from a satisfying solution that it would be best to restart?” Are we expecting on average to solve a problem of X complexity in Y seconds, and so if we go over this time by a certain amount, we are to move on? But then, how would we measure the complexity of the SAT instance? From the number of variables or clauses? Even then the structure of the SAT problem could be very different, which would lead to varying levels of difficulty to solve. Or do we want to move on if we seemingly cannot get closer to a satisfying solution as measured by our solutions' fitness score not improving after a set number of generations? Ultimately, we never truly know if we have diverged too far from solving our problem, and at the moment we discard our population, we may have been on the brink of finding a satisfying solution. Subsequently, there must be an appropriate plan that incorporates these together.

2.6 Decision Heuristics

In every stochastic solver, we need a heuristic that our algorithm strives for in order to get closer to the correct solution. This is how our population chooses which direction to go, and is expressed by how our solution's fitness score improves. There have been many different successful heuristics applied to SAT solvers, some of which we can use to more accurately determine the fitness score of our own solutions. One of the more popular local search algorithms GSAT attempts to maximise the number of clauses satisfied by flipping the value of a variable, and only selects the new assignment if it does increase the number of clauses satisfied [14]. We could translate this step into our genetic SAT solver as a greedy mutation. But note that due to it being a greedy implementation, it would lead to much higher chances of premature convergence.

One SAT solver, Zchaff, which bases its decisions on VSIDS (Variable State Independent Decaying Sum), is considered to be the best complete solver in industrial and handmade benchmark categories [34]. Unfortunately, it is difficult to incorporate heuristics such as VSIDS into our genetic algorithm because VSIDS considers how to improve the working solution at a much more granular level, taking into account statistics for each variable in choosing the truth value to assign to the variable. Whereas genetic algorithms tend to improve the solution by only taking into account the fitness score of the solution and improving on this using genetic methods such as crossover and mutation.

On the other hand, we could attempt to make more informed decisions by refining how we define our fitness function. There are more recent genetic SAT solvers that have had great success in this, with one such solver making use of the mechanism 'Stepwise Adaptation of Weights' abbreviated to SAW [22]. The mechanism aims to measure how hard a clause is to satisfy at each step of the algorithm. Along with this new information, the fitness function of the solution will be calculated (satisfying the harder clauses gives a higher fitness score). Using this new fitness function can heavily increase the likelihood of reaching a global optimum [22].

2.7 Exploiting the Properties of a SAT problem

There are two other properties of SAT that we can take advantage of that are unrelated to genetic algorithms but could make the implementation much faster. These two properties are commonly used in DPLL algorithms, and it is possible to combine them with the genetic approach to yield even better results.

The first property of a SAT problem is that it may contain several unit clauses. A unit clause is when the clause only contains a single unassigned literal. Because we have to satisfy every clause, we are forced to assign the value to this literal that satisfies that clause, meaning that during that iteration of assigning truth values to variables, that variable can only ever be assigned that particular value.

The second property we can make use of is the existence of pure literals. This is when a literal only ever appears in either their positive or negated form throughout the whole problem. With this knowledge, we can safely assign it to the first form we see it in and not have to reassign it ever again.

Both of these properties together can potentially decrease the time to solve a problem because it will, dependant on how many unit clauses and pure literals are in the formula, decrease the search space. For each unique one of these we find, we lose a factor of 2 in the search space, as for each unique variable assigned using one of these properties we are forced to give it a value without having to determine which value it should have.

Chapter 3

Specification & Design

The main objective of this project is to build an efficient SAT solver based on the genetic algorithm that is capable of solving 3-SAT problems containing up to 3000 variables with a 1:1.5 ratio of variables to clauses within 5 minutes, having a success rate above 80%. To help us to visualise how this will be accomplished, we lay out a requirements and specification that we hope to achieve.

3.1 Requirements

3.1.1 User Requirements

- The user will be allowed to submit their own SAT problem file to be solved provided it is in DIMACS format
- The user will be allowed to create randomly generated instances of SAT from a user-inputted number of variables and clauses
- The user will be able to flexibly customize the configuration of the solver
- The user will be displayed a satisfying solution if the solver finds one
- The user will be displayed the solution with the highest *fitness score* if the system cannot find one in the time limit
- The user will be displayed the time taken to solve the problem file (if the problem is solved), or how long we ran the solver for before terminating

3.1.2 System Requirements

- The system must be able to read and understand a file when it is in DIMACS format and be able to write a new problem file in DIMACS format
- The system must be able to read the user input and adjust its solver's configuration accordingly
- The system must be able to write the statistics of whether it solved the problem and if so the time taken to solve the problem back to the problem file
- The program will keep a timer of how long it has taken to solve the problem
- The program will always answer within a set amount of time
- It is a sound but not complete system (can prove satisfiability, but not unsatisfiability).
- The program will keep track of the current best solution according to the solution's fitness score
- If the problem is satisfiable, the solver will be able to solve easy medium-sized SAT problems and hard small-sized SAT problems in under 5 minutes
- The system must be able to provide multiple techniques for *mutation*, *parent selection*, and *parent crossover*.
- The system must be robust and have comprehensive unit and integration tests
- The system must be easy to configure and maintain

3.2 Specification

3.2.1 Reading the Problem in DIMACS Format

The system will only accept and create SAT instances that are in the DIMACS format. The DIMACS format is the standard format for SAT solving [27] and is the format used in SAT competitions. There are some very real benefits of the DIMACS form which I will highlight as I go over how DIMACS files are structured. Firstly, DIMACS allows for commenting simply by starting the line with the letter 'c', allowing the author of the file to leave useful information such as whether the problem is satisfiable. Then, the start of the problem is indicated by the 'problem line' which is laid out 'p FORMAT VARIABLES CLAUSES'. We will be able to use this information to our advantage in the implementation of the SAT solver. Then the clauses appear immediately afterward, listed one by one, each on a separate line which allows us to parse the file and gain the information very easily. The variables of each clause are represented by numbers and if we want to represent a negated variable, we simply add a '-' onto the front of it. All these features together make our lives much easier in getting to work with the problem file, let alone the added advantage of it being the standard format so comparison of different algorithms and heuristics is an easy job.

3.2.2 Generating SAT problems to Solve

Our solver will have the capability to either accept a SAT problem to solve or generate its own. Allowing our solver to be able to create random problems to solve is very important for two reasons. Firstly, we choose to create our instances of SAT randomly due to our solver being suited best to randomly generated instances of SAT. And secondly, to make valid conclusions when we compare different implementations, we need to use a wide range of results across many different SAT instances. Another note to add is that we will only be generating 3-SAT problems which are problems where the clauses contain exactly 3 literals. This is simply because the number of variables in the clauses will affect satisfiability, so we only create one type of SAT problem for consistency in the results. Furthermore, every instance of SAT can be reduced to 3-SAT [7], leading it to be the most common form of SAT.

3.2.3 Outputting our Results

Analysing and comparing the results of a genetic solver can be hard due to the nature of the genetic algorithm. Through *parent selection*, *mutation*, and the initial generation of our population, we introduce randomness to the algorithm. If we were to only run the solver on a problem once, then the reliability of using that one result to judge the efficiency of the solver would be extremely low. Each run of the algorithm can produce very different results. To try to combat this, we will need to repeat our experiment as many times as possible to reduce the margin of error. To make this easier for us, we will append to the problem file whether the problem was satisfied and how long we ran the solver for. This way, we can run the solver numerous times and collate the results afterwards.

3.2.4 Configuring the Solver

The leading factor in whether a genetic SAT solver will be effective is the combination of parameters we give to the solver. Because of this, it needs to be simple and flexible to modify the configuration of the solver. The user must be able to change this configuration via command line inputs. Furthermore, the parameters that the user can change must at least contain: the mutation rate, elitism rate, and population size.

3.2.5 The Performance of the Solver

Despite our main goal not being to create a solver that is capable of winning SAT competitions or to be used in industry, by finding the optimal combination of parameters for the solver, there should be promising results. Measuring the performance of a genetic SAT solver presents many challenges. Before anything else, there needs to be a decision made about the correct balance between the guarantee of finding a solution, with the the time taken to find a solution. Theoretically, it is impossible to get a 100% success rate even with many satisfying solutions possible as one particular run may just be extremely unlucky and continuously fall into a local optima it cannot escape. On the other hand, we want our solver to be successful on the majority of the runs it takes, so there is an aim of an 80% success rate. In terms of the size of SAT problems solved, there is a hope for the solver to have the ability to find a satisfying solution in easy instances of SAT that have up to 3000 variables (what constitutes a SAT instance to be easy is explained in the evaluation).

3.2.6 Use of a User Interface

In our solver, we will not make use of a user interface. There is no need for one as our SAT solver is a purely back-end program. Even in industry, they are typically used solely as a back-end tool [2]. Instead, we can focus our efforts on analysing various implementations and building an efficient solver.

3.3 Design

3.3.1 Software

For the implementation, I have chosen to use Java. This is due to Java being an object-oriented programming language, which gives many benefits. The first such benefit is the extendability that this will bring, for instance, the modularity that Java gives will allow us to modify our implementation and extend it very easily. In this particular case, for example, it gives us the ability to test and change the implementation for mutation and crossover to find the most optimal combination of methods seamlessly. Furthermore, the vast multitude of open-source libraries and prominent community support will help us in being as ambitious as possible for our implementation. Finally, there are also a few quirks over other programming languages that help us such as its automatic garbage collection, and its property of being platform-independent allowing us to run it on any machine that can run a JVM [5].

3.3.2 Program Flow

To give a better understanding of the flow of the program, we can describe it using the following flow diagram:

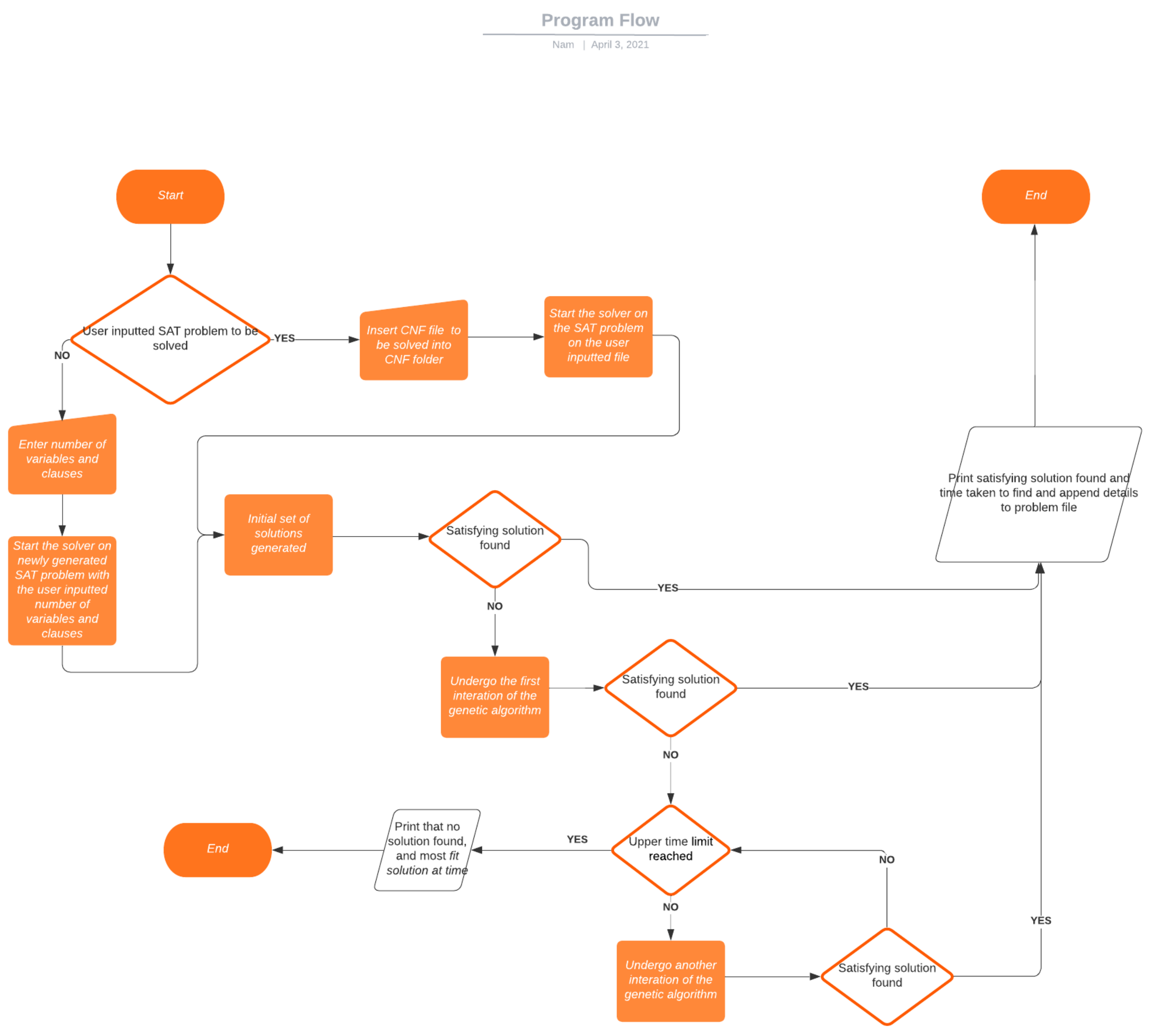
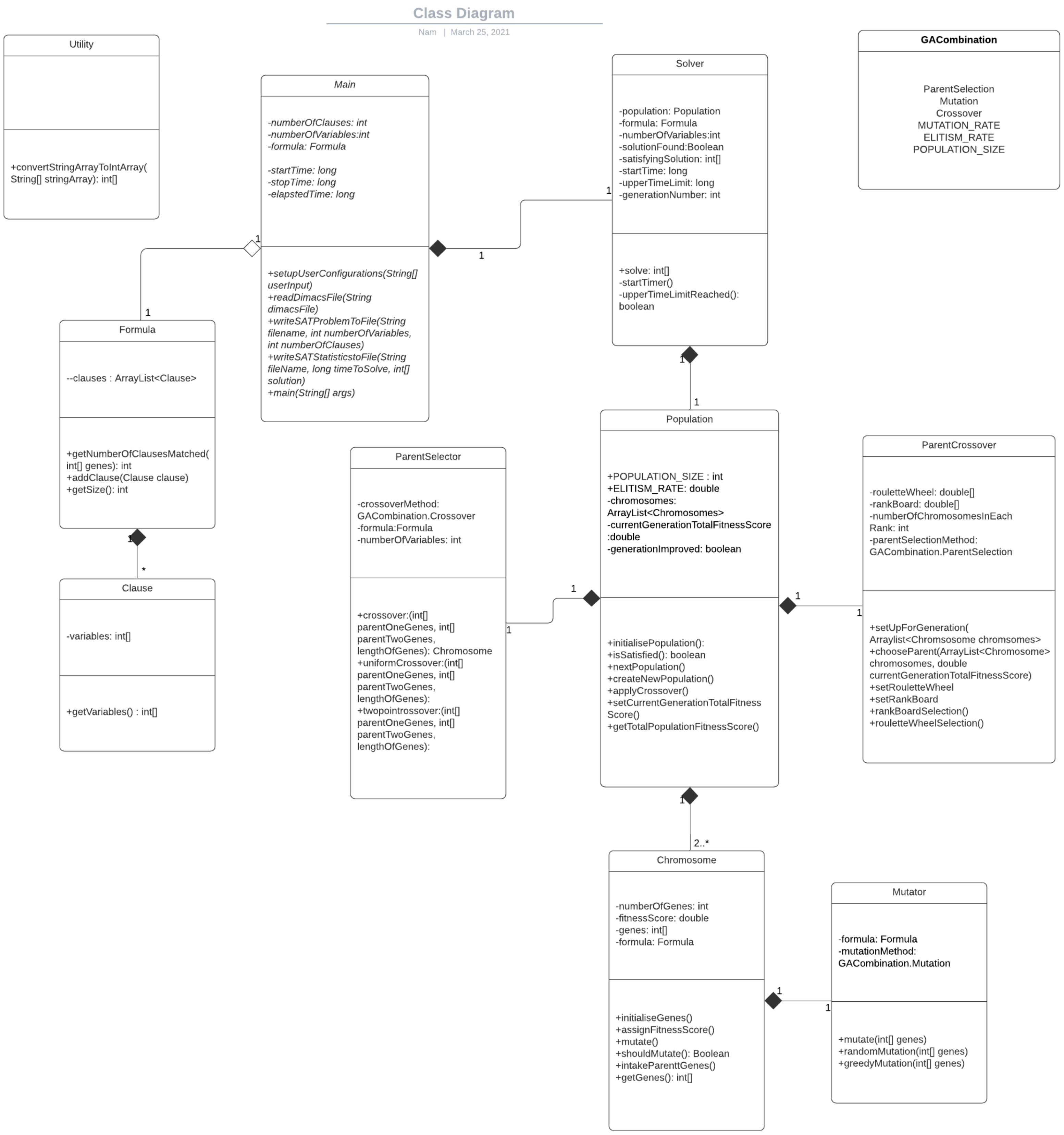

Program Flow
Nam | April 3, 2021
Start
User inputted SAT problem to be solved
YES
NO
Insert CNF file to be solved into CNF folder
Start the solver on the SAT problem on the user inputted file
Enter number of variables and clauses
Start the solver on newly generated SAT problem with the user inputted number of variables and clauses
Initial set of solutions generated
Satisfying solution found
YES
NO
Undergo the first interation of the genetic algorithm
Satisfying solution found
YES
NO
Upper time limit reached
YES
NO
Print that no solution found, and most fit solution at time
End
Undergo another interation of the genetic algorithm
Satisfying solution found
NO
YES
Print satisfying solution found and time taken to find and append details to problem file
End

Figure 3.1: Program Flow

1. Firstly, we choose whether we want to input our own SAT problem file to be solved. Note that at this stage we will also allow the user to enter a default configuration for how the solver will be run.
    (a) If we choose to input our own file, then we have to insert this file into the CNF folder.
    (b) However, if we choose to solve a randomly generated problem, we input our choice of the number of variables and clauses for the program to generate the problem (A new problem file will be created to hold the information).
2. The initial set of solutions is then generated randomly, and we check if there is a satisfying solution amongst them.
    (a) If yes, we simply print this solution out to the user along with the time taken to find this solution and append this information to the problem file.
    (b) If not, then we will undergo the first iteration of the genetic algorithm to produce a new set of solutions in the hope of finding a satisfying solution.
3. If we do not find a satisfying solution within this new set of solutions, then we keep looping through iterations of the genetic algorithm, until we either find a solution or the max time limit has been reached.
4. During the improvement of our set of solutions by going through multiple iterations, if we meet the criteria for our *restart policy* (defined further on in the report), then we will discard our current set of solutions and randomly generate a new set of solutions to work from. Note that this is not shown on the diagram to avoid over-complicating the diagram.

3.3.3 Program Architecture

To build this program, we have gone for as modular approach as possible as shown in the following class diagram:



Class Diagram
Nam | March 25, 2021
Utility
+convertStringArrayToIntArray(String[] stringArray): int[]
Main
-numberOfClauses: int
-numberOfVariables:int
-formula: Formula
-startTime: long
-stopTime: long
-elapstedTime: long
+setupUserConfigurations(String[] userInput)
+readDimacsFile(String dimacsFile)
+writeSATProblemToFile(String filename, int numberOfVariables, int numberOfClauses)
+writeSATStatisticstoFile(String fileName, long timeToSolve, int[] solution)
+main(String[] args)
Solver
-population: Population
-formula: Formula
-numberOfVariables:int
-solutionFound:Boolean
-satisfyingSolution: int[]
-startTime: long
-upperTimeLimit: long
-generationNumber: int
+solve: int[]
-startTimer()
-upperTimeLimitReached(): boolean
GACombination
ParentSelection
Mutation
Crossover
MUTATION_RATE
ELITISM_RATE
POPULATION_SIZE
Formula
--clauses : ArrayList<Clause>
+getNumberOfClausesMatched(int[] genes): int
+addClause(Clause clause)
+getSize(): int
Clause
-variables: int[]
+getVariables() : int[]
ParentSelector
-crossoverMethod: GACombination.Crossover
-formula:Formula
-numberOfVariables: int
+crossover:(int[] parentOneGenes, int[] parentTwoGenes, lengthOfGenes): Chromosome
+uniformCrossover:(int[] parentOneGenes, int[] parentTwoGenes, lengthOfGenes):
+twopointrossover:(int[] parentOneGenes, int[] parentTwoGenes, lengthOfGenes):
Population
+POPULATION_SIZE : int
+ELITISM_RATE: double
-chromosomes: ArrayList<Chromosomes>
-currentGenerationTotalFitnessScore :double
-generationImproved: boolean
+initialisePopulation():
+isSatisfied(): boolean
+nextPopulation()
+createNewPopulation()
+applyCrossover()
+setCurrentGenerationTotalFitness Score()
+getTotalPopulationFitnessScore()
ParentCrossover
-rouletteWheel: double[]
-rankBoard: double[]
-numberOfChromosomesInEach Rank: int
-parentSelectionMethod: GACombination.ParentSelection
+setUpForGeneration(Arraylist<Chromsosome chromsomes>
+chooseParent(ArrayList<Chromosome> chromosomes, double currentGenerationTotalFitnessScore)
+setRouletteWheel
+setRankBoard
+rankBoardSelection()
+rouletteWheelSelection()
Chromosome
-numberOfGenes: int
-fitnessScore: double
-genes: int[]
-formula: Formula
+initialiseGenes()
+assignFitnessScore()
+mutate()
+shouldMutate(): Boolean
+intakeParenttGenes()
+getGenes(): int[]
Mutator
-formula: Formula
-mutationMethod: GACombination.Mutation
+mutate(int[] genes)
+randomMutation(int[] genes)
+greedyMutation(int[] genes)
1
1
1
1
1
1
1
1
*
1
1
1
1
1
2..*
1
1

Figure 3.2: Class Diagram

A brief description of the classes are given below:

- The Main class is responsible for reading and creating problem files and then starting up the process of solving the problem by creating the formula to solve, as well as the solver to solve the formula. It will also modify the configuration for the run if the user inputted arguments to do so. Lastly, it will write to the problem file the results of the attempt at solving the problem.

- We represent our problem as an instance of the class Formula, which then contains instances of the class Clause. The reason we have separated it like so is that we may want to calculate how hard a clause is to solve in the future, so it makes sense to modularise it now.

- The Solver class takes the problem and attempts to solve it by initialising a new population and iterating through sets of solutions until the time limit we have given to it passes. It is also in charge of checking if we should restart with a fresh set of solutions due to our restart policy.

- The Population class is where the genetic algorithm is contained. This is where our solutions are generated and improved from.

- We then have our ParentSelector and ParentCrossover class. These two classes help create new solutions by selecting parent solutions and combining the two.

- For each solution, we represent it with an instance of Chromosome. The population contains a fixed amount of Chromosomes.

- The Mutator class mutates the Chromosome's genes to modify the solution.

- The GACombination class contains the configuration for the solver that can be adjusted at each run. This includes three different enums so that we can easily switch between the types of implementation we want to use for a specific run of the program.

- The Utility class is a helper class with utility functions.

3.3.4 Development Approach

In my approach, I have opted for a test-driven development approach (I have omitted all test classes from the class diagram for clarity). Going for this approach is relatively straightforward to implement because we are building a pure back-end program, and so creating meaningful tests before each part of the implementation is reasonably simple. There are many benefits for going down this route, the first of which is that it allows you to have a focused mindset. By writing one test and then writing the implementation for that test, it forces you to split up the functionality and focus on the smaller chunks which consequently produces more modular code. However, the main benefit of test-driven development that we feel is when we are refactoring the code. Once we have implemented our first iteration of the SAT solver, we will have a comprehensive set of unit tests that should cover each piece of functionality. Now, every time we make a change to our code we can run our unit tests and confirm that our code still works as intended. This is especially beneficial to us when we will be continuously tweaking our implementation at a rapid pace to find the most optimal combination of methods in solving the SAT problem. On the other hand, there is one compromise we have had to make and that is to test each method we are required to make them public, which loses some of the encapsulation for each class. However, this is a trade-off that we are happy to make due to the benefits mentioned above.

Chapter 4

Implementation

In this chapter, we will go through in more detail the various implementations of the main stages of the genetic algorithm that attempt to *improve* our solutions that we have opted for. We will argue the choice of method for each piece, whilst weighing the advantages and disadvantages. Furthermore, to create an effective SAT solver, we need it to be as efficient as possible and so we will also be discussing the time complexities along with this. Likewise, we will go over the reasoning of some of the design choices we have made. We will end with a discussion of the testing taken place.

4.1 Improving our Set of Solutions

4.1.1 Selecting Parents

Before this step, we would have scored each of our chromosomes (solutions) with a fitness score marking how close each one was to a satisfying solution, but we will not delve into this as it is not what we are focused on. Alternatively, we will look at how we choose the chromosomes to combine that make an offspring (new solution).

A naive approach to selecting parents would be to simply choose the fittest individual in the population, but this could greatly accelerate premature convergence occurring. Premature convergence happens when fitter individuals end up dominating the entire population. Thus, we need to maintain good diversity to increase the likelihood of finding a satisfying solution [23]. An approach to this problem is the Roulette-Wheel Selection.

The Roulette-Wheel Selection is based on the concept of proportionality. Depending on the fitness of the chromosome, the chromosome's chance of getting chosen gets increased. For instance, if we combine all the fitness values of the chromosomes and it comes to 1000, and an individual has a fitness value of 100, then that individual will have a 10% chance of being chosen to be a parent. There are a couple of problems with this though. As you can imagine, it cannot deal with negative fitness values and as the population converges, it will lose its selection pressure, which is the degree to which the fitter individuals will be chosen [31]. Therefore, it may take longer than necessary for our solver to find a satisfying solution. As well as this, if in our initial generation created our population's chromosomes all have very similar fitness values and are far away from a satisfying solution, then in this scenario, our selection pressure would also be very low [26], meaning again that the time taken to converge to the correct solution will take much longer.

Knowing this, the default route which we have decided on is a slight adaptation to Roulette-Wheel Selection, which we will call Rank Board Selection (not to be confused with the common method *Rank Selection)*. The difference we add is that individuals will be placed into ranks sequentially by fitness score, where the number of ranks is smaller than the number of individuals. Then, the chance that each rank is chosen will be proportionate to the sum of the individuals' fitness scores in that rank. Then, we randomly choose an individual in the chosen rank. This means that individuals with very similar fitness scores can have the same chance of being chosen to be a parent. The consequence of this is that it may take longer to find a satisfying solution, however, this will stop a very small number of individuals from completely dominating the population which could potentially stop us from coming to premature convergence. This selection choice would produce the same results as Roulette-Wheel if the number of individuals in each rank were to be one. Note we will still give the option of Roulette-Wheel Selection for our solver as it does have advantages over Rank Selection.

Implementation of Parent Selection

To implement this, there are 3 stages. First we need to create the table of ranks. Next, choose which rank we want to select the chromosome from, and then finally choose a chromosome from that rank. The operation that dominates the run time of parent selection is creating the rank table. There is one operation in this that dominates the running time, and that is when we have to sort the chromosomes in order of fitness score. Luckily, sorting is so fundamental to algorithms that Java has an in-built sort function we can use which has the order of $O(n \log n)$ where n is the size of the population.

Due to the modularisation of our code, we can more freely choose when we call the creation of the rank board. Rather than performing this method each time we select our parents, we can choose to perform it once a generation. This will still give us the same rank board because the chromosomes we use to create the rank board will be the same. This could potentially save us tremendous amounts of time, as if we have a population of 500 chromosomes and choose to replace our worst-ranked half with new offspring, we will be reducing the number of times we create the rank board from 500 to 1.

If we only cared about the running time of the specific parent selection method, and not about the aspects of convergence and went for the two fittest individuals, then even that would have a time complexity of $\theta(n)$ due to having to parse through an unordered list of chromosomes to find the ones with the highest fittest score. If we were to choose any parent then of course this could be $O(1)$ time, but going with this approach, generally, would not improve the population overall. Roulette wheel has the same time complexity as Rank Board selection due to the same reason of having to sort the chromosomes in fitness order.

4.1.2 Combining the Selected Parents

We name each bit of the binary string that represents our chromosomes as an allele. The simplest approach to combine the two parents' alleles is named Uniform Crossover. Uniform Crossover is where we alternate choosing alleles from each parent. For instance, if we had two parents whose genes are of length 3 then the offspring's gene's first and third allele would be parent one's first and third allele and its second allele would be parent two's second allele.

A different approach is Two-Point crossover where we split the gene into three sections by randomly choosing two pivot points along the gene. We create the new offspring by taking the sections before and after the two pivot points from one parent's gene and combining it with the middle section of the other parent's genes. An extension we can make to this is making sure we crossover roughly 20% to 80% of the genes. The reason for this is that in Fixed-length crossover, choosing a crossover length outside of these boundaries greatly reduces the success rate of the algorithm [6]. We are assuming the same to be true for Two-Point crossover.

At first glance, it is not immediately clear which direction would be the better choice, and there have been results in favour of both [30][6]. In light of this information, we will ensure the solver has the option to use either option of crossover.

Implementation of Parent Crossover

Combining two parents to create an offspring is a fairly trivial task that is made easy by our chromosome's solution being represented by an int array. We merely create a new int array to hold the new solution, and copy into this array the parents' genes in the order we have decided on. For the Two-point crossover, we just add an extra condition that we are applying crossover to between 20% and 80% of the genes

In general, the running time of deciding the order is constant as the operations needed are to randomly select a couple of numbers, a few comparative checks, and to store the numbers in variables. In Uniform Crossover, the order is pre-defined so this step is not needed. However for both cases, when we fill up the offspring array with its parent's genes, this would take $\theta(n)$ time. There is not much use trying to improve this as when we initialise the offspring array, it would already take $\theta(n)$ time.

4.1.3 Mutating our Genes

For mutation, the simplest approach may be the best approach. This would be to randomly flip a bit of the chromosome resulting in potentially a worse solution. The reason that this way of mutation is so appealing to us is that it allows us to escape premature convergence as well as slow the rate of it. Because of this, we will use random mutation as our default method. If we were to go down the route of a greedy mutation, then the risk of premature convergence skyrockets due to our solutions rapidly becoming less diverse since the chromosomes would only mutate if it would increase their fitness score. But the major advantage is that we would converge to a satisfying solution much faster (provided the algorithm did not get stuck in a local optima).

Implementation of Mutation

Another benefit of a completely random mutation, along with its simplicity, is the running time would be much shorter than others as we would only need to indiscriminately flip a bit of the gene, which overall would give a running time of $O(1)$. Whereas, for a greedy implementation, we would have to calculate for each bit if flipping that bit would increase the fitness score or not. This would take a much longer time because for each bit we would have to run a calculation to check whether the fitness score is improved. This calculation is in the order of $O(m)$, where m is the number of clauses in the problem. Since the number of bits refers to the number of variables in the problem, the overall running time of this implementation would be $O(nm)$

where n is the number of variables and m the number of clauses in the problem. Although we will keep the mutation rate at a generally low number, the difference in time complexity will still produce a very noticeable difference.

4.1.4 Settling on the Frequency of Mutation and Crossover

Finding the optimal chance we give our chromosomes to undergo mutation and crossover is not a simple task, and it is often wise when considering one to consider the other. If we put that chance of crossover too high and mutation too low, then the mutation won't be capable of diversifying our population enough against the high convergence rate. However, one proven successful strategy in deciding the chance of mutation and crossover is having an adaptive crossover and mutation strategy [11]. An adaptive strategy does not rely on set values that do not change throughout the entire process. One possible adaptive strategy could be to increase the chance of mutation as we iterate through the generations. The reasoning for this is at the beginning, if there are too many mutations occurring, it may take a longer time to converge to a satisfying solution, but towards the end when our solutions are closer to the answer, then we may want to increase the number of mutations to increase the chance of escaping local optima.

Another interesting strategy that has been proven effective in deciding on the chance of crossover is through the structure of the chromosomes chosen to crossover [16]. Calculating the Euclidean distance between the two chromosomes and then increasing the likelihood of crossover to occur in longer distance pairs of chromosomes was also shown to be effective [16]. The main reasoning for its success is that it promotes diversity and avoids inbreeding, which would help to overcome premature convergence.

However, the strategy we will decide on is having a fixed mutation and crossover rate. We do not want to over-complicate our implementation, and throwing in too many variables will make it difficult to draw meaningful conclusions when we evaluate our system.

Implementation of the Rate of Mutation and Crossover

To find the best values to use for rate of mutation and crossover, we will do preliminary testing to roughly find the pair of values that work best, then slightly tune the values to match the combination we want (e.g. decreasing mutation rate and increasing crossover rate for a high convergence combination). To implement our chromosomes mutating at a rate of 0.01 per chromosome is a straight-forward task. We store the mutation rate in a variable, and then for each chromosome we randomly generate a number between 0 and 1 and if the number is below the mutation rate, then we mutate the chromosome. For our crossover rate, we store a variable called ELITISM_RATE which represents the percentage of the population that passes on to the next generation. If we want a crossover rate of 0.95, then 0.05 of our population moves on to the next generation. We use this to copy over our decided amount of chromosomes, and the remaining spots left in the generation are then taken up by chromosomes formed from crossover. As you can expect, adjusting crossover and mutation rate is straightforward which will be helpful in our evaluation. We do not need to worry about this affecting our overall running time as this implementation is of constant time.

4.1.5 Resetting our Population

If our mutation rate is low then we may not be able to escape a local optima even when we are using a random mutation. And as mentioned earlier, it is impossible to know the perfect situation to discard our current set of solutions and start afresh. However, there are a few things we can look for that hint when it is time to do so. For instance, if our solution does not improve after a few iterations, it may be worthwhile to reset our population. Or, if after a set amount of iterations we have not found the solution, we should potentially restart.

4.1.6 Implementation of Restart Policy

For our implementation, we will keep a counter of the generation number, the count of how many generations have passed that our solutions have not improved for, and the time between each reset. Unfortunately, we cannot set a permanent value for the number of generations before each restart or time between each reset to use for every problem, as adjusting these values for each instance of SAT will improve results [11]. We can describe the restart policy with the following pseudo-code.

```
while !solutionFound do
    oneIterationOfGeneticAlgorithm;
    generationNumber++;
    if SolutionsNotImproved then
        consecutiveGenerationsNotImproved++
    if ShouldRestartAlgorithm then
        RestartAlgorithm
    if UpperTimeLimitReached then
        break;
```

Note that *UpperTimeLimitReached* refers to the time limit we give to the whole solver before it gives up completely, and *RestartAlgorithm* refers to resetting the population.

ShouldAlgorithmRestart will return *true* when one of the following previously mentioned counters passes its set limit:

- Number of generations without an improvement to fitness score
- Number of generations in total
- Time since last reset

Fortunately, adding our restart policy will not noticeably slow down how fast our algorithm iterates through generations. All the operations are of constant time, and since for each generation we sort the solutions in order of fitness score when we undergo parent selection, we can easily obtain the fittest solution to check if our solution is improving.

Later on in our evaluation, we will tune our restart policy and possibly even omit some conditions.

4.2 Design Choices

4.2.1 Chosen Data Structures

The main data structure that is used in the implementation of our solver is an array. This is because the most common operation we will be doing is indexing into a data structure that stores our solution when we are measuring the fitness score of our solution. For arrays, this operation is constant time. Furthermore, the performance is fast in comparison to the ArrayList because of their fixed size [18]. Although we do insert to an array when we create our new offspring, this operation would be constant time too as we do not have to shift the elements in the array.

4.2.2 Using Enums

Another design choice we have made is the creation of a class containing Enums (GACombination). This class will hold the names of all the various implementations for ‘ParentSelection’, ‘ParentCrossover’, and ‘Mutation’. The purpose of doing this was so we could easily switch between different types of implementation to find the best combination. As mentioned earlier, the rate of convergence is a very careful metric we have to control and each implementation will cause the rate of convergence to differ, but also, each implementation will most likely have a different running time. We could have made use of interfaces and interfaced the ParentSelector, ParentCrossover, and Mutator class, and created separate implementations for each version of parent selection, parent crossover, and mutation to then plug into the relevant classes. However, we feel that there would be too much overhead and the use of switch case statements along with Enums accomplished the objective well enough. We can highlight how easy this makes our solver to change between implementations by displaying an example:

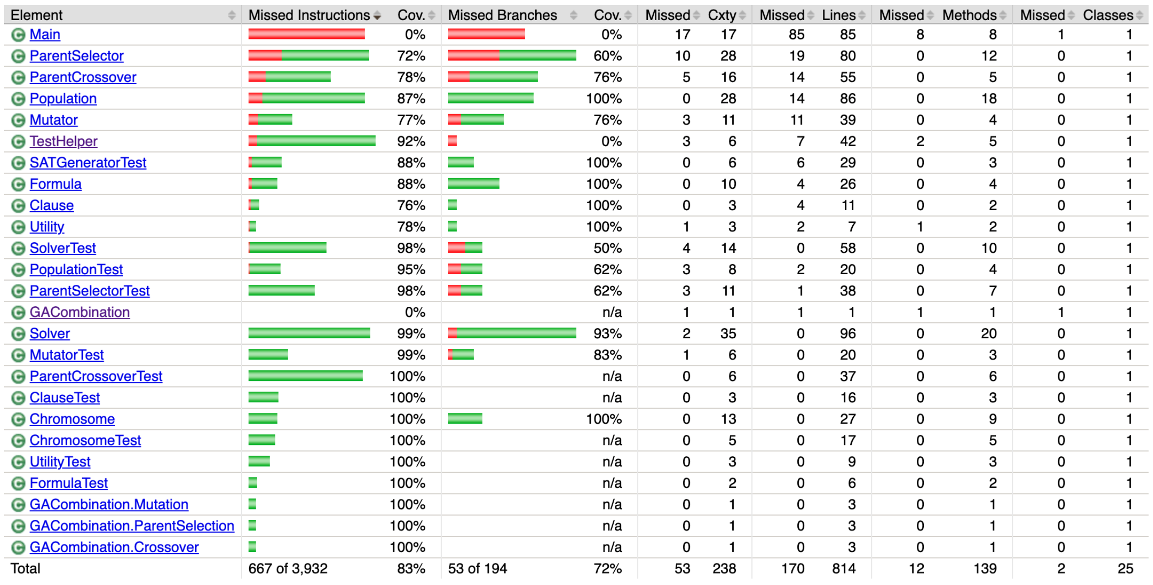
```
public void mutate(int[] genes) {
  switch (mutationMethod.name()) {
    case "Random": randomMutation(genes);
    break;
    case "Greedy": greedyMutation(genes);
    break;
    default: randomMutation(genes);
   }
```

4.3 Software Testing

There is no doubt that we need to have confidence in our system to always provide us with the correct result and constantly be available to our users. Unit testing is the main method in which we will provide this reassurance.

4.3.1 Unit Testing

Thankfully, because early on we had opted for a test-driven development approach, by the time of the completion of our implementation, we already had an extensive set of unit tests. (The main benefits are described earlier in our development approach.) We can take a high-level overview of the statistics of our unit testing using Jacobo, a code coverage tool, with the results given below:



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Main		0%		0%	17	17	85	85	8	8	1	1
ParentSelector		72%		60%	10	28	19	80	0	12	0	1
ParentCrossover		78%		76%	5	16	14	55	0	5	0	1
Population		87%		100%	0	28	14	86	0	18	0	1
Mutator		77%		76%	3	11	11	39	0	4	0	1
TestHelper		92%		0%	3	6	7	42	2	5	0	1
SATGeneratorTest		88%		100%	0	6	6	29	0	3	0	1
Formula		88%		100%	0	10	4	26	0	4	0	1
Clause		76%		100%	0	3	4	11	0	2	0	1
Utility		78%		100%	1	3	2	7	1	2	0	1
SolverTest		98%		50%	4	14	0	58	0	10	0	1
PopulationTest		95%		62%	3	8	2	20	0	4	0	1
ParentSelectorTest		98%		62%	3	11	1	38	0	7	0	1
GACombination		0%		n/a	1	1	1	1	1	1	1	1
Solver		99%		93%	2	35	0	96	0	20	0	1
MutatorTest		99%		83%	1	6	0	20	0	3	0	1
ParentCrossoverTest		100%		n/a	0	6	0	37	0	6	0	1
ClauseTest		100%		n/a	0	3	0	16	0	3	0	1
Chromosome		100%		100%	0	13	0	27	0	9	0	1
ChromosomeTest		100%		n/a	0	5	0	17	0	5	0	1
UtilityTest		100%		n/a	0	3	0	9	0	3	0	1
FormulaTest		100%		n/a	0	2	0	6	0	2	0	1
GACombination.Mutation		100%		n/a	0	1	0	3	0	1	0	1
GACombination.ParentSelection		100%		n/a	0	1	0	3	0	1	0	1
GACombination.Crossover		100%		n/a	0	1	0	3	0	1	0	1
Total	667 of 3,932	83%	53 of 194	72%	53	238	170	814	12	139	2	25

Figure 4.1: Code Coverage

Although we have reasonably high test coverage, we should not rely purely on test coverage results to let us know how well tested our code is. Firstly, test coverage does not show if all edge cases have been covered. For example, it is up to us to account for situations such as when we implement Rank Board selection that each rank may not contain the same number of individuals if the population is not divisible by the number of ranks. It also does not mean that you have tests to verify each individual line works as expected.

Another point is that due to the nature of our implementation, it can be extremely hard to get perfect coverage at each run of our set of unit tests. Certain methods such as how we combine parents have an element of randomness in them so creating worthwhile unit tests is tricky. We can only check that the produced chromosome is a new chromosome with a high chance of different genes, but can not fully confirm it is a product of the two parents undergoing a specific crossover. However, it does present us with an easy option to walk through the method, albeit it being a manual process. Moreover, our unit tests may take a long time to test certain scenarios. For instance, to check how many restarts happen in one run of the algorithm, we need to fully run the algorithm, meaning that this one unit test would take the upper time limit we give the solver to solve the instance. Another downside is that we are forced to provide extra getter and setter methods to fully test our implementation. This has the knock-on effect of bulking our code and adding lines that do not specifically increase the effectiveness of our solver.

But on the flip-side, that we can use it to find unused lines of code which allows us to clean our codebase. And more importantly, the more coverage we have, the more confidence we will have in having discovered as many bugs as possible.

For the larger integration tests, these have to be constantly adapted depending on how we choose to run our solver. Additionally, writing a test to check whether our solver finds a satisfying solution to the given SAT instance is hard. Our solver may find a different satisfying solution each time it is run, and so we cannot compare a predetermined satisfying solution with it. To overcome this, we can create our own SAT instance which only has one solution. The only problem with this is that the SAT instance may be very simple, but it does give us confidence that the solver is working correctly.

4.3.2 Manual Testing

Conveniently, not much manual testing is required due to there being no user interface. We only require our user to enter in a SAT problem if desired and a few parameters meaning there is little scope for the user to do something unexpected. Moreover, there is not an expectation for any users to misuse the system as there is no gain for the user in doing so.

The only manual testing required is to see if our Solver truly produces a satisfying solution to the given SAT problem. The test to see whether the given solution is satisfying should be independent of the solver. It is not correct if we allow our solver to judge whether it itself has provided a correct solution. If the number of clauses is not too large, we can deem if the outputted solution is satisfiable by eye. We can simply check that each clause is satisfied by the given solution one by one. If we repeat this manual test with various different SAT instances, we can be fairly confident that the solver is correct with all SAT instances, even those with a very high number of clauses and variables. Nothing fundamentally changes as the number of clauses and variables increases.

Chapter 5

Results/Evaluation

This chapter will analyse the results of the different configurations of implementations and parameters we give to the SAT solver on various instances of SAT, and then tuning the best one found. There are too many different combinations to consider trying every possible one, however, our tactic is to select the combinations we think will be most promising in terms of speed and success rate of finding a satisfying solution.

5.1 Testing Methodology

We will run the solver on a machine with a 1.6GHz dual-core Intel Core i5 processor, and 8GB of 2133MHz LPDDR3 onboard memory.

As previously mentioned, we will be testing our solver on randomly-generated instances of SAT. Despite our problems being randomly-generated, instances of the same problem size can still vary in difficulty to solve, and some combinations will be better suited for each. For this reason, we cannot rely on generating only one problem for each problem size, so we will generate 5 for each. We will then run our solver 5 times against each problem to obtain a more accurate result. Furthermore, as the ratio of numbers of clauses to variables increases, the chance of unsatisfiability increases, with the hardest region being roughly at the point where the ratio is 4.26 [28]. Considering this, we will generate two sets at each problem size, one where the ratio is 1.5, and the other 4.5. We will test if different combinations are more suited to solving harder instances of SAT. Testing will also only be done on satisfiable instances of SAT. (This is not too important when comparing results, as all our solvers will be unsuccessful if the instance is unsatisfiable.)

5.2 Combinations to Test

There are many different variables we can alter/choose, each of which will change the results we will obtain. We will attempt to find the best combination of variables. These are laid out below:

Population Param	Parent Selection	Parent Crossover	Mutation	Restart Policy
Population Size Elitism Rate Mutation Rate	Roulette Wheel Rank Board	Uniform Two-Point	Random Greedy	Total Gen. Unimproved Gen. Time Passed

Table 5.1: Variables to Alter/Choose

5.2.1 Uniform Crossover vs Two-Point Crossover

Earlier, we mentioned that we would test both Uniform Crossover and Two-Point crossover due to it being unclear what difference choosing one over the other would make. Bearing this in mind, when we come to tune our final variation of the solver we create, we will alternate between the two crossover methods to discover which is more effective.

5.2.2 Solver Variations

There will be 3 variations we test, each one being at a different level on the scale of convergence rate.

A_Solver will have the following configuration:

- Population Size: *100*, Elitism Rate: *0.95*, Mutation Rate: *0.1*
- Parent Selection: *Rank Board*, Parent Crossover: *Uniform*, Mutation: *Random*
- No Restart Policy

This will have the slowest convergence rate, but in return will reduce the chance of getting stuck in a local optima. We will not use a restart policy for this solver.

B_Solver will have the following configuration:

- Population Size: *20*, Elitism Rate: *0.90*, Mutation Rate: *0.01*
- Parent Selection: *Rank Board*, Parent Crossover: *Uniform*, Mutation: *Random*
- Restart Policy: Time Passed 1/4 of total time given, Unimproved Gen. = 30

This will have the medium convergence rate, and what we expect to be the correct balance of finding a solution with finding one within an acceptable time.

C_Solver will have the following configuration:

- Population Size: *15*, Elitism Rate: *0.85*, Mutation Rate: *0.1*
- Parent Selection: *Roulette Wheel*, Parent Crossover: *Uniform*, Mutation: *Greedy*
- Restart Policy: Time Passed 1/8 of total time given, Unimproved Gen. = 5

This will have the most aggressive convergence rate which we will combine with a sensitive restart policy.

5.3 Results

The time limit given to each solver to solve a problem will be 300 seconds, and each solver will attempt to solve each problem 5 times. This means that each solver will have a total of 25 runs at each problem size.

Problem Size	100, 150	500, 750	1200, 1800	3000, 4500
A_Solver	Success Rate: 100%	Success Rate: 100%	Success Rate: 100%	Success Rate: 16%
B_Solver	Success Rate:28%	Success Rate: 0%	Success Rate: 0%	Success Rate: 0%
C_Solver	Success Rate: 100%	Success Rate: 100%	Success Rate: 0%	Success Rate: 0%

Table 5.2: Success Rate of the solvers on easy instances of SAT

Problem Size	100, 150	500, 750	1200, 1800	3000, 4500
A_Solver	443.3ms	4650.8ms	30396.3ms	266507.2ms
B_Solver	178700.7ms	N/A	N/A	N/A
C_Solver	157.3ms	3643.7ms	N/A	N/A

Table 5.3: Average time if solved on easy instances of SAT

Problem Size	10, 45	30, 135	50, 225	100, 450
A_Solver	Success Rate: 100%	Success Rate: 70%	Success Rate: 30%	Success Rate: 10%
B_Solver	Success Rate: 100%	Success Rate: 100%	Success Rate: 0%	Success Rate: 0%
C_Solver	Success Rate: 100%	Success Rate: 100%	Success Rate:100%	Success Rate: 0%

Table 5.4: Success Rate of the solvers on hard instances of SAT

Problem Size	10, 45	30, 135	50, 225	100, 450
A_Solver	433.3ms	6489.7ms	53313ms	3800ms
B_Solver	131.2ms	99465.5ms	N/A	N/A
C_Solver	151.1ms	520ms	4887ms	N/A

Table 5.5: Average time if solved on hard instances of SAT

Surprisingly, *Solver_B*, the solver first thought to be most effective, is, in general, the worst performing solver. The success rate of finding a solution on even the smallest easy SAT problems is below 1/3. This is thought to be because the restart policy is too sensitive for the relatively slow rate of convergence, so its current set of solutions is discarded before we come close to finding the satisfying solution. On the other hand, *Solver_C* which had an even more sensitive restart policy was overall most successful in solving hard instances of SAT, as well as being much quicker at it. This is likely because *Solver_C* has a much faster rate of convergence which can overcome its restart policy. Because it is much easier to get trapped at a local optima in hard instances of SAT, *Solver_A* which although has a low rate of convergence, still has a high chance of getting stuck, and with no restart policy, it has a low success rate on hard instances of SAT. Nonetheless, *Solver_A* has the highest success rate once the problem size increases, as well as the time difference of finding a solution between *Solver_A* and *Solver_C* decreasing. This is because once the size problem size increases, the duration to find a satisfying solution increases, and so there are more chances to get stuck at a local optima, especially further on in the computation. This means that the chance of the high converging rate solver reaching a solution before getting stuck at a local optima is very low. Out of curiosity, *Solver_C* and *Solver_B* were both tested without the restart policy, and they both performed much worse than *Solver_A*. Solver_C performed worse than Solver_A most likely because the computation cost of a greedy mutation was too great, especially as the problem size grew, and it was unable to iterate through enough solutions, as well as it being less adept at escaping local optima.

Another problem with these high convergence solvers is that they are much more unpredictable, and the variance at the speed they solve instances of SAT varies greatly. Some instances were solved in 500ms, and then the same instances were also solved in 50,000ms on a different run. From these results, we will be taking *Solver_A* forward.

5.4 Tuning Solver_A

5.4.1 Choosing Between Uniform and Two-Point Crossover

After alternating to Two-Point crossover from our original Uniform crossover, we ran the same tests as we did previously. Interestingly, the results did not differ significantly enough for us to be confident in concluding that one was beneficial over the other. Even after adjusting the Elitism Rate to the point where most chromosomes undergo crossover, the difference seen is not notable. In light of this, we will assume Two-Point crossover would bring the same results.

5.4.2 Adding in a Restart Policy

The results show that a restart policy can be beneficial if adapted to each problem. For instance, if we have a hard instance of SAT but it is small in size, we can add in a sensitive restart policy to greatly increase our success rate of finding a solution in the given time. On the flip side, we would want to remove our restart policy once the problem grew to a certain size as we would want to rely on our solutions to be able to escape local optima when they approached a satisfying solution themselves. This is because if we had a too sensitive restart policy, we would be throwing away our solutions before they even got close to a satisfying solution.

5.4.3 Adjusting Population Size

Changing the population size from 100 to 15 had some promising results. After rerunning the solver with this change on our larger problem sets, there were very clear improvements.

Problem Size	1200, 1800	3000, 4500
A_Solver	Success Rate: 100%	Success Rate: 16%
Adjusted A_Solver	Success Rate: 100%	Success Rate: 100%
Change	0%	+84%

Table 5.6: Success Rate after adjusting Population size

Problem Size	1200, 1800	3000, 4500
A_Solver	30396.3ms	266507.2ms
Adjusted A_Solver	13033.1ms	125763.6ms
Change	-17363.2ms	-140744.2ms

Table 5.7: Time Taken after adjusting Population size

The success rate on our largest problem size went all the way from 16% to 100%, and the time taken to find a solution decreased by more than a half. Unfortunately, on re-testing against the hard instances of SAT, the performance heavily decreases. By having a smaller population size, we allow our solutions to iterate through the process faster giving themselves more opportunity to improve and reach the solution. But on the flip side, having a smaller population leads to our population becoming less diverse and more prone to getting stuck in a local optima which is most likely why our solver performs much worse on hard instances of SAT.

Problem Size	100, 450	
Adjusted A_Solver	0%	410 score
Change	-10%	-40

Table 5.8: Performance on hard instances of SAT after adjusting pop. size

From the table, it is clear that the solver does not even come close to finding a solution in a hard instance of SAT.

As a side note, slightly tweaking Elitism rate and Mutation rate only seemed to cause our solver to be less effective, so we seem to have reached the sweet spot for these two variables.

5.4.4 A Semi Restart policy

One of the problems of our restart policy is that we throw away all of our solutions after certain criteria are met without even retaining any information about the SAT instance, so the chance of solving the SAT problem essentially stays the same at each restart. In an attempt to not waste the progress made until the restart, we will save a portion of the population on each restart. To create the population we save, we will pick every other chromosome from the fittest to least fit chromosome, and if we choose to save more than half the population we will loop back and choose the chromosomes not already saved. Hopefully, by not only choosing from the top section of the population, we will increase our chance of passing on solutions not stuck in a local optima. The optimal proportion of the population to save as well as how often to restart will also need to be found.

We will test our solver on the biggest hard instance of SAT that our A_Solver was able to solve. We test our restart policy on Solver_A which has a population of 100. We will note the best result after 2 runs of each combination of parameters, which may be a score, or a time if a satisfying solution is found.

		Unimproved Generations Before Restart			
Proportion of pop. to save		10	100	500	3000
	0.02	426 score	449 score	275602ms	449 score
	0.1	449 score	449 score	449 score	66623ms
	0.5	449 score	449 score	449 score	446 score
	0.75	449 score	4286ms	449 score	448 score

Table 5.9: Testing performance of different restart parameters on a small sized (100, 450) hard instance of SAT

From these results, we have found the two extremes which are: a very sensitive restart policy with saving a small proportion of the population, and a much less sensitive restart policy which saves the majority of the population. Although we only ran each test two times, the parameter of 100 unimproved generations before a restart with saving 0.75% of the population gave the best result, so we will do further testing on this.

Problem Size	100, 450	
A_Solver	Success Rate: 10%	3800ms
A_Solver With R.Policy	Success Rate: 10%	165306ms
Change	0%	+161506ms

Table 5.10: Success Rate with new Restart Policy on hard instance of SAT

The unpredictability of whether the solver will solve a problem is shown in the results above. After running the solver 10 times on the same instance, we only managed to solve the instance once, and it took considerably longer than our preliminary tests where we searched for the optimal parameters. This leads to uncertainty about whether we have chosen the optimal parameters. Nevertheless, we will test the same restart policy on the 3000, 4500 instance of SAT.

Problem Size	3000, 4500	
A_Solver	Success Rate: 16%	266507.2ms
A_Solver With R.Policy	Success Rate: 40%	250705ms
Change	+24%	-15802.2

Table 5.11: Success Rate with new Restart Policy on easy instance of SAT

With the restart policy, there are promising results with larger instances of SAT. However, the results are still worse than Adjusted A_Solver(A_Solver with a population size of 15 rather than 100). Note that with the previous restart policy where we saved none of the population, solving this instance had a success rate of 0%.

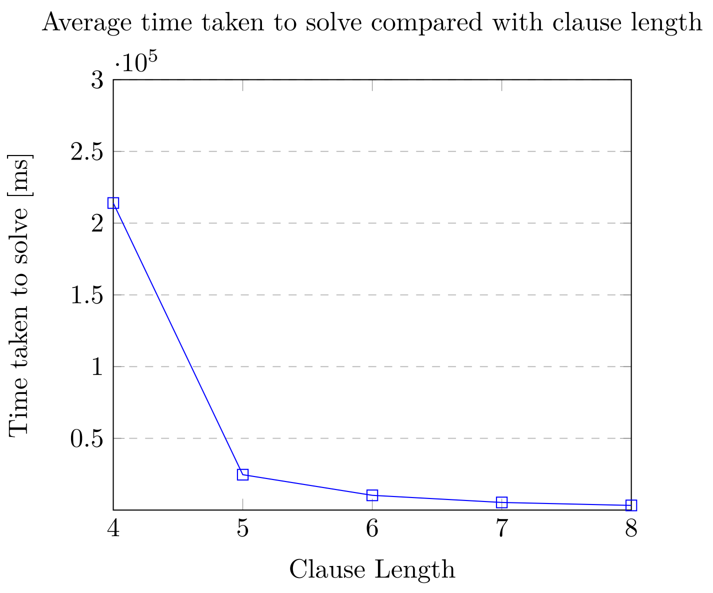
5.5 Evaluation on Clauses Bigger than 3-SAT

Although increasing the length of clauses will increase the number of possible satisfying solutions, this may harm many heuristics used in other SAT solvers. However, in our case, it would only benefit our solver. Preliminary testing showed that the A_Solver with the new restart policy performed best on clauses with more than 3 variables (following tests will be done on this solver), and surprisingly that the Adjusted A_Solver performed by far most poorly.

Solver	Success Rate	Average Time
A_Solver	100%	32692.2ms
Adjusted A_Solver	0%	N/A
A_Solver With R.Policy	100%	24994.8ms

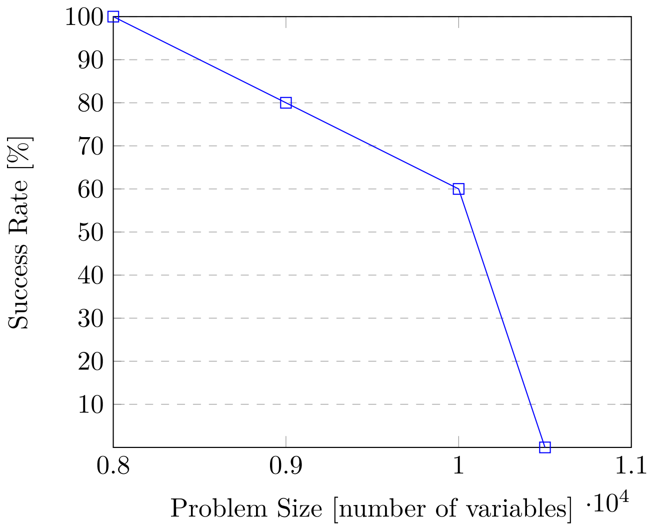
Table 5.12: Performance on instance of 4-SAT with 500 variables to 2250 clauses

To measure the impact of increasing clause length, we will test with SAT instances of size 1000 variables and 4500 clauses. Note that clause length of 3 is not included as the solver is unsuccessful at solving this.



From the graph, increasing clause length drastically decreases the time to find a satisfying solution. Out of curiosity, we will test the biggest SAT instance with a clause length of 8 that the solver can solve. The variable to clause ratio will be 4.5.

Success rate on SAT instances with clauses of length 8



The results show that the solver starts to struggle to solve SAT instances of size around 10,000 variables, and fails to solve when the problem reaches around 10500 variables. However, overall, these graphs show how drastically our solver can improve in success rate and time taken to find a solution if we increase clause length. This has led us to discover that a genetic SAT solver such as ours is best suited to instances of SAT that have longer clauses.

5.6 Evaluation of Requirements

In terms of whether the requirements and specifications in our design specification were met, most can be answered with a simple yes or no. Of these requirements, our system was successful in all of them. On the other hand, there are a few requirements that require more analysis, which are the ones concerning the performance, maintainability, and robustness of the system.

In terms of the maintainability and robustness of the system, we have included a vast set of unit and regression tests, as well as ensuring basic tools such as try-catch blocks and commenting are utilised. Along with this, there is extensive logging throughout the program to catch any possible points of failure. On the flip side, despite our already high code coverage, we could have stretched it even further. If we had interfaced our components, it would have given us greater control over their behaviour via mocking, which would have allowed us to test different situations more easily, leading us to potentially achieve a near 100% code coverage.

For the performance of our solver, although we were more focused on finding the correct combination of parameters for the solver, the results are fairly lackluster, especially considering the ones gained from testing on hard instances of SAT. Although we set expectations low from the specification stage, the performance of our solver is considerably poorer than other local search algorithms [21]. On the other hand, the solver becomes much more effective in finding a satisfying solution if the clause length increases. We have potentially found the type of SAT instances that our genetic SAT solver considerably outperforms the other SAT solvers on. Nevertheless, one of the main ways we can improve the effectiveness of our solver, particularly on hard instances of SAT, is by further modifying the restart policy. Currently, on restarting the algorithm, our solver retains no information about the structure of the problem which could have helped us solve the problem. A possible solution is discussed later on in Future Work.

Another concern about our results is that their reliability is not absolute. Although we took great care in getting as fair results as possible, the time taken to run the solver was always going to be a problem. To obtain results we could trust, we would have had to run each variation of the solver at least 100 times on each problem file. This meant if we had 40 problem files to test, one variation would take 14 straight days to fully test (assuming each run took the maximum time of 300000ms). A possible solution to this would have been having many identical computers to run the solver in parallel with each other, but unfortunately this was a luxury we did not have.

Displaying solutions to users could have also been a point we could have taken further. The current implementation simply displays the solution as an int array, where each index refers to the variable. But this is not something we were concerned about as a SAT solver is a back-end program.

Chapter 6

Legal, Social, Ethical and Professional Issues

6.1 British Computer Society Code of Conduct

From the design of the program to its completion, the **BCS Code of Conduct** has been thoroughly thought about at each stage [4]. The majority of the rules issued by the BCS do not apply to our project. And although all of the rules are important, some however are more relevant to our project than others. The first of these is to ”respect and value alternative viewpoints”. It feels common sense and customary to every project that this is undertaken, nonetheless this has been greatly appreciated. Throughout this project, to discover better combinations for our SAT solver we are forced to consider different viewpoints, and when there are two contradicting answers, to value both. Another point that may have relevance is to ’NOT misrepresent or withhold information on the performance of products, systems or services’. Although our solver is far from being one capable of being used in industry, the information provided in our evaluation could potentially influence decisions made by others. Due to this, we have made our evaluation as open as possible, detailing the system our solver was tested on as well as how we endeavoured to make our testing as fair as possible.

6.2 Other Professional Issues

Other issues such as software trustworthiness, privacy, and security have also been taken into account. With regards to software trustworthiness, extensive testing has been done.

The subject of privacy and security does not come up much in our project. The only times where we are concerned with this is when we read/write files, and possibly when exposing the contents of the user-given problem to solve (if the user has one). In reading files, the program can only read files in the specified CNF folder, and for writing to files, the user can only write very specific data to the file so we are not too concerned with this. Keeping information about the user's problem secure it not a major concern of ours, because the user has to format their problem in a way that reveals almost zero information about their problem.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Building a considerably large piece of software from scratch has shown the importance of taking advantage of Java's ease of modularity, and choosing an appropriate development approach (in this case Test Driven Development) which has allowed rapid experimentation with different implementations without the risk of regression. Constructing a solver by using a genetic algorithm has also given an appreciation of their advantages such as adaptability via parameter selection, different techniques to escape local optima, and much-improved effectiveness when applied to longer clauses. But it has also shown the shortfalls, namely the unpredictability of their performance on each particular run. Comparing different implementations has also highlighted key aspects of building an efficient algorithm, in particular that computation costs of greedy algorithms may outweigh their benefits, especially as the problem size grows. On top of this, it has also been discovered that constructing an efficient solver needs to be done on a problem by problem basis. For the majority of our problems however, a low convergence solver worked best with a combination of Rank Board selection, Uniform crossover, random mutation, and a population size of 100.

7.2 Future Work

Earlier, it was mentioned that a possible improvement to the solver would be to further modify the Restart Policy. Although it has been shown that adding a restart policy that saves a proportion of the population improves the effectiveness of the solver in many scenarios, there are still improvements that could be made. A potential idea would be to on each restart, collate information about the SAT instance learnt throughout that set of iterations before a restart. Then, we could use this information to help guide the population in the next run. For instance, we could store the difficulty of solving each clause, perhaps depending on how many generations before satisfying the clause, then changing the fitness score gained appropriately. Guiding the algorithm like so would aim to minimise the chance of getting stuck in a local optima, because hopefully the harder clauses would have had more chance of being solved.

Another key advantage of Genetic Algorithms that was not taken advantage of in this project is their ability of parallelism. Unlike complete algorithms where running many solvers together would likely not speed up finding a solution by a significant amount, doing so with a Genetic Algorithm could. However, rather than running many of these Genetic Algorithm incorporated solvers together, an interesting idea is to combine a complete solver with an incomplete solver [14]. One way to implement this would be for the complete solver to guide the incomplete solver by periodically feeding the incomplete solver its current solution for the incomplete one to use as one of its working solutions. However, finding or building a complete solver to complement our incomplete solver is in itself an entire new process, but it remains a compelling proposal.

References

[1] Shawn Anderson. When and how to solve problems with genetic algorithms. https://spin.atomicobject.com/2017/10/09/genetic-algorithm-example/, 2017. Accessed: 2020-10-10.

[2] Cyrille Artho1, Armin Biere, and Martina Seidl. Model-based testing for verification back-ends. https://core.ac.uk/download/pdf/206813162.pdf. Accessed: 2021-1-31.

[3] Rafaela Azevedo. What is the cost of a bug. https://azevedorafaela.com/2018/04/27/what-is-the-cost-of-a-bug/, 2018. Accessed: 2021-1-21.

[4] The Chartered Institute for IT BCS. Bcs code of conduct. https://www.bcs.org/membership/become-a-member/bcs-code-of-conduct/. Accessed: 2021-03-10.

[5] Akhil Bhadwal. Features of java. https://hackr.io/blog/features-of-java. Accessed: 2020-10-11.

[6] Arunava Bhattacharjee and Prabal Chauhan. Solving the sat problem using genetic algorithm. *Advances in Science, Technology and Engineering Systems*, 2:115–120, 2017.

[7] Subham Datta. Sat and 3-sat – cook-levin theorem. https://www.baeldung.com/cs/cook-levin-theorem-3sat. Accessed: 2021-01-10.

[8] Barry O'Sullivan David Devlin. Satisfiability as a classification problem. http://www.cs.ucc.ie/~osullb/pubs/classification.pdf, 2008. Accessed: 2020-10-11.

[9] Sanjit A. Seshia Edward A. Lee, Jaijeet Roychowdhury. Fundamental algorithms for system modeling, analysis, and optimization. https://ptolemy.berkeley.edu/projects/embedded/eecsx44/fall2011/lectures/SATSolving.pdf, 2010. Accessed: 2020-9-30.

[10] Lance Fortnow. The status of the p versus np problem. *Communications of the ACM*, 52:78–86, 2009. Accessed: 2021-1-21.

[11] Huimin Fu, Yang Xu, Guanfeng Wu, Hairui Jia, Wuyang Zhang, and Rong Hu. An improved adaptive genetic algorithm for solving 3-sat problems based on effective restart and greedy strategy. *International Journal of Computational Intelligence Systems*, 11:402–413, 2018.

[12] Alex S. Fukunaga. Evolving local search heuristics for sat using genetic programming. In Kalyanmoy Deb, editor, *Genetic and Evolutionary Computation – GECCO 2004*, pages 483–494, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[13] Ian P Gent, Holger H Hoos, Andrew G D Rowley, and Kevin Smyth. Using stochastic local search to solve quantified boolean formulae. `https://static.aminer.org/pdf/PDF/000/117/385/using_stochastic_local_search_to_solve_quantified_boolean_formulae.pdf`. Accessed: 2021-1-22.

[14] Weiwei Gong and Xu Zhou. A survey of sat solver. *AIP Conference Proceedings*, 1836(1):020059, 2017.

[15] P. Guo, X. Wang, and Y. Han. The enhanced genetic algorithms for the optimization design. In *2010 3rd International Conference on Biomedical Engineering and Informatics*, volume 7, pages 2990–2994, 2010.

[16] Ahmad Hassanat, Khalid Almohammadi, Esra'a Alkafaween, Eman Abunawas, Awni Hammouri, and V. B. Surya Prasath. Choosing mutation and crossover ratios for genetic algorithms—a review with a new dynamic approach. *Information*, 10(12), 2019.

[17] Ali Hurson and Veljko Milutinovic. Software verification. `https://www.sciencedirect.com/topics/computer-science/software-verification`. Accessed: 2021-1-21.

[18] Sonoo Jaiswal. Difference between array and arraylist. `https://www.javatpoint.com/difference-between-array-and-arraylist`.

[19] Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. The international sat solver competitions. *AI Magazine*, 33(1):89–92, Mar. 2012.

[20] Nathan Landman, Karleigh Moore, and Christopher Williams. P versus np. `https://brilliant.org/wiki/p-versus-np/`. Accessed: 2021-1-21.

[21] Frédéric Lardeux, Frédéric Saubion, and Jin-Kao Hao. Gasat: A genetic local search algorithm for the satisfiability problem. *Evolutionary computation*, 14:223–53, 02 2006.

[22] Jana Lovíšková. Solving the 3-sat problem using genetic algorithms, 09 2015.

[23] Gabriele De Luca. Roulette selection in genetic algorithms. https://www.baeldung.com/cs/genetic-algorithms-roulette-selection. Accessed: 2020-10-11.

[24] Vijini Mallawaarachchi. Introduction to genetic algorithms. https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3 2017. Accessed: 2020-10-10.

[25] Joao Marques-Silva. Practical applications of boolean satisfiability. https://eprints.soton.ac.uk/265340/1/jpms-wodes08.pdf, 2008. Accessed: 2020-09-30.

[26] Amit Naik. Genetic algorithms. https://medium.com/datadriveninvestor/genetic-algorithms-461cd20817bb, 2019. Accessed: 2020-10-11.

[27] Beyond Np. Cnf formats. http://beyondnp.org/pages/formats/. Accessed: 2020-10-11.

[28] Eugene Nudelman, Kevin Leyton-Brown, Holger Hoos, Alex Devkar, and Yoav Shoham. Understanding random sat: Beyond the clauses-to-variables ratio, 09 2004.

[29] The University of Edinburgh. Np-completeness and cook's theorem, analysis, and optimization. http://www.inf.ed.ac.uk/teaching/courses/propm/papers/Cook.pdf, 2002. Accessed: 2020-9-30.

[30] Dr. Alfredo Cruz Oscar Pérez Cruz. Evolutionary sat solver (ess). http://laccei.org/LACCEI2011-Medellin/StudentPapers/OT017_Perez_SP.pdf. Accessed: 2020-10-11.

[31] Rosshairy Abd Rahman, Razamin Ramli, Zainoddin Jamari, Ku Ruhana, and Ku-Mahamud. Evolutionary algorithm with roulette-tournament selection for solving aquaculture diet formulation. *Mathematical Problems in Engineering*, 2016:1–10, 2016.

[32] Ravi Kiran S S and Gaurav R. Bhaya. A comparative study of algorithms for propositional satisfiability. https://courses.cs.washington.edu/courses/cse573/04au/Project/mini1/GauravRavi/report.pdf.

[33] Ashish Sabharwal. Modern sat solvers: Key advances and applications. https://courses.cs.washington.edu/courses/csep573/11wi/lectures/ashish-satsolvers.pdf, 2011. Accessed: 2020-10-10.

[34] Emmanuel Zarpas, Marco Roveri, Alessandro Cimatti, Klaus Winkelmann, Raik Brinkmann, Yakov Novikov, Ohad Shacham, and Monica Farkash. Improved decision

heuristics for high performance sat based static property checking. `https://www2.cs.sfu.ca/research/groups/CL/software/siege/prosyd.pdf`. Accessed: 2021-1-28.