=



Comparison of Sudoku Solving Algorithms

Calum Harvey (Student ID: 170349985)

Computer Science

Supervisor: Dr Jason Steggles

Word Count: 18397

Abstract

Sudoku is a NP-complete logic number puzzle that has steadily increased in popularity. As the puzzle has become more widespread, the number of algorithms to solve such puzzles has also increased. These algorithms vary drastically and the ability to compare and evaluate different popular algorithms is becoming more important. This project aims to investigate the speed and efficiency of a range of different solving algorithms that allow for an evaluation to be made. The results and analysis are presented both statistically and graphically, along with additional work that could be carried out in the future.

Declaration

“I declare that this dissertation represents my own work except, where otherwise stated.”

Acknowledgements

Callum for teaching me word referencing

Table of Contents

[1 Introduction 7](#_Toc39248172)

[1.1 Sudoku 7](#_Toc39248173)

[1.2 Motivation 8](#_Toc39248174)

[1.3 Aim 8](#_Toc39248175)

[1.4 Objectives 8](#_Toc39248179)

[2 Background Research 9](#_Toc39248180)

[2.1 Sudoku Puzzles 9](#_Toc39248181)

[2.2 Puzzle Generation 9](#_Toc39248182)

[2.2.1 Pre-generated Puzzles 10](#_Toc39248183)

[2.2.2 Generating Puzzles 10](#_Toc39248184)

[2.3 Algorithms 12](#_Toc39248185)

[2.3.1 Backtracking 12](#_Toc39248186)

[2.3.2 Hill Climb 12](#_Toc39248187)

[2.3.3 Simulated Annealing 13](#_Toc39248188)

[2.3.4 Genetic Algorithm 14](#_Toc39248189)

[2.4 Existing Systems 15](#_Toc39248190)

[2.5 Implementation Technologies 16](#_Toc39248191)

[2.5.1 Python 16](#_Toc39248192)

[2.5.2 Visual Studio Code 16](#_Toc39248193)

[2.5.3 NumPy / Matplotlib 16](#_Toc39248194)

[2.5.4 Tkinter 17](#_Toc39248195)

[3 System Design 19](#_Toc39248196)

[3.1 Requirements 19](#_Toc39248197)

[3.2 Methodology 19](#_Toc39248198)

[3.3 High Level Design 20](#_Toc39248199)

[3.4 Test Bed Design 21](#_Toc39248200)

[3.4.1 Design of Graphical User Interface 21](#_Toc39248201)

[3.4.2 Design of Algorithm and Puzzle communication 22](#_Toc39248202)

[3.5 Design of Puzzle Generation 23](#_Toc39248203)

[3.6 Design of Algorithms 24](#_Toc39248204)

[3.6.1 Backtracking 24](#_Toc39248205)

[3.6.2 Hill Climb 26](#_Toc39248206)

[3.6.3 Simulated Annealing 28](#_Toc39248207)

[3.6.4 Genetic Algorithm 29](#_Toc39248208)

[4 Implementation 31](#_Toc39248209)

[4.1 Sudoku Generation 31](#_Toc39248210)

[4.1.1 Version 1 31](#_Toc39248211)

[4.1.2 Version 2 32](#_Toc39248212)

[4.2 Algorithms 34](#_Toc39248213)

[4.2.1 Backtracking 34](#_Toc39248214)

[4.2.2 Hill Climb 36](#_Toc39248215)

[4.2.3 Simulated Annealing 38](#_Toc39248216)

[4.2.4 Genetic Algorithm 40](#_Toc39248217)

[4.3 Test Bed 42](#_Toc39248218)

[4.3.1 Puzzle and Algorithm Interaction 42](#_Toc39248219)

[4.3.2 GUI and Analysis 43](#_Toc39248220)

[5 Testing 45](#_Toc39248221)

[5.1 Unit Testing 45](#_Toc39248222)

[5.2 Algorithm Testing 45](#_Toc39248223)

[5.2.1 Interesting Bugs? 48](#_Toc39248224)

[5.3 Sudoku Generation Testing 48](#_Toc39248225)

[5.4 Test Bed Testing 48](#_Toc39248226)

[6 Evaluation 52](#_Toc39248227)

[6.1 Results 52](#_Toc39248228)

[6.2 Project Requirements 56](#_Toc39248229)

[6.3 User Study Plan? 57](#_Toc39248230)

[7 Conclusion 59](#_Toc39248231)

[7.1 Overview 59](#_Toc39248232)

[7.2 Aims and Objectives 59](#_Toc39248233)

[7.3 Personal Development 61](#_Toc39248234)

[7.4 Challenges 62](#_Toc39248235)

[7.5 Future Work 63](#_Toc39248236)

# Introduction

## Sudoku

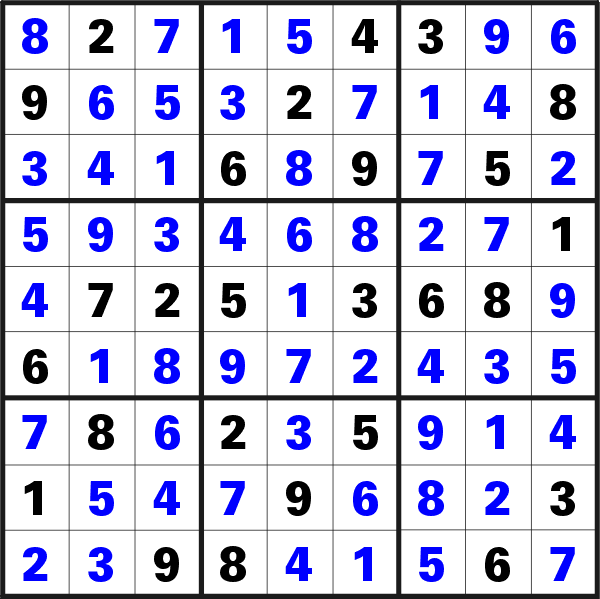
Sudoku is a logic-based number placement puzzle game [1] that has grown in popularity since it first appeared in Dell magazine in America [2]. It usually consists of a 9x9 board containing 81 individual cells which are further partitioned into 9 3x3 smaller boxes that need to be filled. The aim is to fill these cells with a number 1 to 9 with each cell containing a single integer. There are 3 constraints on the board that must be met, each row, each column and each 3x3 smaller box must contain the numbers from 1-9 only once [3]. When a sudoku is created a number of the cells are pre-defined by the puzzle creator to ensure that the puzzle only has one unique solution. The difficulty of the puzzle is determined by the number of pre-filled cells in the grid, more is easier; less is harder [4].

Figure 1.1 Completed Sudoku puzzle [42]

There have been various algorithms implemented to solve the Sudoku problem. The way a human solves the easier problems revolves around using the numbers already in the board and using logic to determine the missing numbers in each cell. When the problem becomes harder and requires the person to start guessing numbers, simple algorithms such as backtracking can be used to come to a solution in a shorter time than the human. The problem with the brute-force backtracking is the efficiency of the algorithm especially as the number of empty cells starts to increase as the Sudoku problem gets harder.

The answer to solve this efficiency problem could be the use of the stochastic algorithm optimization. Where backtracking searches through all the possible solutions to find the optimal result, stochastic algorithms can reduce the number of searches by stochastically iterating through the solution space for a puzzle only taking improvements to the potential solution. This allows the algorithm to move forward towards the optimal solution without having to check every possible outcome of the puzzle with the complexity of the problem being almost irrelevant to the efficiency of the algorithm.

## Motivation

The problem with all these algorithms is since all stochastic algorithms are different it can be very hard to tell which one is the most efficient to use on a Sudoku puzzle and more importantly are different algorithms better for different difficulties of problem. On an easier puzzle a backtracking algorithm might be the most efficient as there are fewer empty cells and the backtracking can cycle through all the possibilities quicker than a stochastic algorithm can come to an optimum solution. However, the more complex problems might require a more efficient stochastic algorithm but it’s unclear which one is the best to choose.

This paper explores a number of the different stochastic algorithms that can be employed to solve Sudoku problems along with backtracking to allow for a base brute-force case to also be examined alongside the more efficient algorithms. These can then be compared against each other in terms of their speed and number of iterations taken to reach the optimum solution. These can also be compared at a number of different difficulties of puzzle across a wide set of problems to ensure the comparison is as accurate as possible.

## Aim

To develop a system that allows for the investigation and comparison of main sudoku solving algorithms at a range of difficulty of puzzle.



## Objectives

1. Explore current methods of sudoku solving and select three
2. Develop test bed to allow comparison of algorithms
3. Establish test data for comparison of algorithms
4. Implement algorithms into the test bed

1. Evaluate implemented sudoku algorithms at multiple complexities of puzzle
2. Explore state of the art sudoku solving tools and evaluate selection

# Background Research

Within this chapter, the topics that will be covered are Sudoku puzzle generation and the ways that it can be achieved, the algorithms that are intended to be implemented for the system and details of similar research into Sudoku algorithm comparison. This allows me to justify the design decisions that are made later on during the development.

## Sudoku Puzzles

When looking at the way an algorithm solves a Sudoku, we first need to look at the techniques used by a person to solve a puzzle and how these are adapted for the computer algorithms. The techniques used by the solver will depend on the difficulty of the puzzle and also the skill of the solver to be knowledgeable about all possible techniques. The logical approach taken by a human solver is not easily adapted by a programmer into an efficient algorithm, so we have to look at Sudoku from a different angle when using algorithms.

Since a Sudoku puzzle is usually a 9x9 grid containing 81 cells, we can define the 3 criteria that a valid solution must meet as [5]:

* Each Row must contain each number 1-9 only once
* Each Column must contain each number 1-9 only once
* Each 3x3 sub-box must contain each number 1-9 only once

This gives the rules that an algorithm needs to follow when deciding if it has reached a solution to the puzzle. This is the basis of all algorithms for Sudoku solving as these are the constraints that they used to test for the number of errors and correct solutions.

## Puzzle Generation

Sudoku puzzle generation is important to this project as it allows for vast amounts of test data to be created and for the test data to be designed to be the most effective for comparison of algorithms. An example of this is being able to set the difficulty of the puzzles that are being produced which can be imported from a third party that has a database of pre-generated puzzles stored or by generating original boards within the system.

### Pre-generated Puzzles

The simplest way for an algorithm to be tested on a valid sudoku board is to take already generated puzzles from the internet or Newspaper and convert them in a format that can be read by the algorithm and then solved [6]. This is used in research when a single algorithm’s performance is being tested against a limited number of puzzles as a vast number of these puzzles are not required [7].

Choosing pre-generated puzzles guarantees that each puzzle will only have one solution and the difficulty of the puzzle will be guaranteed for each one, allowing for more emphasis on the algorithms. However, my aim is a comparison of algorithms and a more accurate comparison requires a big sample size. This is possible with pre-generated puzzles if I can get them automatically integrated from the generation website to my implementation but the complexity of that is similar to generating my own puzzles.

### Generating Puzzles

There are two main advantages of generating puzzles as test data, the first is that you can control the difficulty of the puzzles that are being created which allows you to test algorithms against a very specific difficulty which allows for more accurate data. This is done by regulating the number of cells that are filled in the 9x9 board with more cells filled making the puzzle easier and having less making it harder.

The second is that these generated puzzles can be created in huge volumes due to them being created by the system which is infinitely faster than manually entering them from a third party. Also, when these puzzles are created the format that they are in is controlled by the system and therefore allows them easily to be parsed in and out of algorithms to be tested.

This is why many papers in the subject of sudoku solving, also incorporate the creation of the puzzles as it not only improves the research that they are doing but also allows for really good test data. This is shown [8] where the solving of sudoku puzzle using genetic algorithms are being looked at but also generation is incorporated into it.

Research into existing computer-based puzzle generators shows that there are 2 methods for random puzzle generation given a grid.

Bottom-up generation begins with blank grid [9]:

* Adds random numbers to random cells in the grid
* Solve the puzzle to find unique solution, if not unique remove number and try another random number and cell
* Repeat for desired difficulty

Top-down generation begins with solved grid [10]:

* Remove numbers from random cells
* Solve the puzzle to find unique solution, if not unique add number back and try another cell
* Repeat for desired difficulty

Both these methods are similar in the way they created the final puzzle to solve, they both require a solving algorithm to determine if there is unique solution for the current puzzle but there is a difference in what the solving algorithm needs to accomplish in the generation. Top-down generation starts with a solved grid, the most efficient way to do this is using a solving algorithm on an empty grid, this works in creating a solution as every solving algorithm as the 3 constraints of a sudoku in it as these are the constraints used to check whether a solution is correct or not. Therefore, using this to create a completed puzzle ensures that it will be a valid solution.

Another difference is when the puzzle is solved to find a unique solution. Bottom-up generation will start by solving an almost empty grid and as more numbers added less computation will need to be done for the puzzle to be solved. Also, there will initially be many solutions to the grid and therefore the solving algorithm will need to be run multiple times to prove that there are these different solutions.

Whereas in top-down generation, after the grid is initially populated by the solving algorithm, the solving algorithm will be run over an almost complete grid and as more numbers are removed, the computation will increase. Also, the search for multiple solutions will grow as more numbers are removed whereas in bottom-up it will decrease as more numbers are added.

The solving algorithm used for generating the grid in top-down generation and for finding a unique solution in both methods has varied but the most common and overall fastest algorithm is the brute-force solver, due to it being effective at finding multiple solutions quickly as it tries every possibility. There is also research into using genetic algorithms for Sudoku generation [8] that due to the randomness of stochastic algorithms allows for multiple solutions in the bottom-up version to be found very fast. It also enables the generation of the complete grid in top-down to be done with potentially more randomness than if a brute-force variation was used instead.

The desired difficulty of the Sudoku when one is being algorithmically generated is dependent on the number of numbered cells that is given. The fewer numbers are given the more work that has to be done in order for the puzzle to be solved. However, this is not always true [11] as it is purely based on the techniques that need to be used in order for the puzzle to be solved and a puzzle that contains more filled cells can need the use of more complex strategies by the solved to complete the puzzle. Although there are many exceptions to this rule, the majority of puzzles follow the correlation between number of cells filled and difficulty to solve.

## Algorithms

The algorithms most commonly looked at are the stochastic algorithms due to their adaptability to Sudoku solving. Simulated Annealing, Genetic Algorithm and Hill Climb are all similar in design, but each have a different take on the stochastic approach.

Selecting the right algorithms is important as there needs to be a variation in the algorithms that are being compared to allow for good data to come out of the project. Backtracking is the brute-force algorithm and therefore is a good base for other algorithms to try and beat. The other stochastic algorithms are used to compare against each other and try and beat backtracking in time taken and efficiency.

### Backtracking

Brute-force backtracking is the most basic and least intelligent algorithm for sudoku solving. In the theme of all brute-force algorithm, it involved searching the whole solution space for a valid answer to the puzzle [12].

The algorithm will select the first empty cell in the grid and try and place a 1. If the 1 breaks on of the 3 constraints of Sudoku, then the algorithm will try the number 2 all the way up to the number 9. When a number of found that does not break any of the constraints then it is added to the cell and the algorithm moves on to the next empty cell, repeating the process until it reaches the end of the grid.

If the algorithm iterates through all the numbers 1-9 without finding a valid option, it will backtrack back to the previous cell and try a new number. This allows for all the possible options for the solution to be tested and therefore the only exit scenarios are that the puzzle is solved, or all possible solutions were tested, and the Sudoku is unsolvable.

Backtracking can be used effectively for comparison against other [13], more complex algorithms as it gives a very good base case of the most straight forward type of algorithm and therefore gives a good comparison of if the more complex algorithm is worth it.

### Hill Climb

Hill Climb [14] is a heuristic searching algorithm used for finding the optimum in mathematical problems in the field of artificial intelligence. Given a large dataset, the algorithm can find a possible solution in a short amount of time. However, this solution may not be the global optimum solution due to it being unable to get out of local minima.

There are three steps in generic Hill Climb:

* Generate a possible solution
* Evaluate the possible solution against expected solution
* If solution as been found then quit, else try again

This can be adapted easily for Sudoku solving with minimal limitation.

To solve sudoku we first need to use a optimisation of Hill Climb called Steepest Ascent Hill Climb [15] which involves examining all neighbouring nodes and then selecting the node that takes the algorithm closer to the solution. This means that given a solvable sudoku puzzle and a way to check the number of errors, we can change numbers in the puzzle and if the number of errors decreases then accept the solution, if not then we try again.

As described [16] there are 3 things that must be defined for the algorithm to succeed: the start state, the successor function and the heuristic function. The start state must be created by initially filling the board to meet one of the 3 sudoku constraints – each box, column and row must have the numbers 1-9. In this example each row is filled with unique numbers to create the starting state. The successor function can swap 2 non-fixed numbers in the same row to create a new solution which can then be checked by the heuristic function which should find the number of errors in the puzzle. This can then be run until the number of errors returned by the heuristic function is zero, meaning the puzzle is solved.

Although this algorithm works, it does not successfully solve a sudoku every time. As seen again in [16], Hill Climb cannot get over local minimum and instead gets stuck meaning it is unable to reach the actual solution of the puzzle. This is solved by adding in random restart into the algorithm that will, after the algorithm goes a while without improving, will restart and create the start state again.

### Simulated Annealing

Simulated Annealing (SA) is an optimization technique which is used for finding the optimal state of a problem by running a series of moves given certain conditions [17]. For each move, a neighbouring state is found by making a small random change to the state of the current state. The new state is then evaluated using a cost function to determine if the new state is an improvement on the current state. If the new state is an upgrade on the current state, then the algorithm changes the new state into the current state. If the new state is worse, then the state only changes given an acceptance probability condition is met, if this is not met then the new state is abandoned, and another move is made.

The acceptance probability is proportional to a temperature which changes throughout the run. Initially, the temperature is set high which allows for more bad moves to be made but as more moves are made the temperature decreases, meaning there is a lower chance for a bad move to be made by the algorithm.

One of the first examples of SA [18] shows how the algorithm can be adapted to allow for solving of Sudoku puzzles. Each state is represented as a matrix with each initially empty cell being filled with random values so that every 3x3 block within the puzzle contains the numbers from 1-9, allowing for one of the constraints of a valid solution to always be true. This means that when a new neighbouring state is being created, the way it differs from the current state is by randomly choosing 2 cells within a block that are not fixed and swapping them.

The way the cost function is implemented in [18] is by looking at each row and column individually and calculating the number of values in each that is missing. Then the total cost of the state is the sum of all rows and columns values, this can be optimised by only recalculating at most 2 rows and 2 columns after each new move as the only costs that will have changed are if the numbers have been swapped.

Another approach [19] uses Quantum Simulated Annealing (QSA) which is different to SA in the way it determines the distance between neighbouring states. In SA the temperature is used for moving from current to new states, whereas in QSA there is a tunnelling field strength which is used to determine the distance between the current state and the neighbouring state.

### Genetic Algorithm

Genetic algorithms (GA) [20] are a family of optimisations inspired by survival of the fittest and evolution. It involves using a fitness function on each chromosome in a population to find the optimum solutions. These are then taken and used to create a new population that is closer to the potential solution.

There are 3 principle stages of a GA:

* Population Initialisation: create a population containing a selection of random chromosomes.
* Fitness Calculation: test each chromosome in the population against a fitness function to determine which are closer to a potential solution
* Selection: choose chromosomes from the population based on the fitness scores. Then, either 2 chromosomes are combined to form new chromosomes which is called crossover, or a chromosome is changed on its own. That is called mutation.

Fitness calculation occurs on all the chromosomes in the population to allow for the best to be selected for reproduction. Selection then occurs repeatedly until a new population is created, allowing the fitness calculation to be repeated again.

To adapt this algorithm to allow for solving of sudoku some changes have to be made. There are different ways that the initialisation can be achieved, similar to Simulated Annealing [21] it can be implemented by imposing a restraint on the random initialisation by only allowing every block to contain the numbers 1-9, meaning the fitness function only needs to check rows and columns to check correctness. This implementation [22] does not restrict the randomness of the initialisation but instead the fitness function checks all 3 conditions of the Sudoku puzzle.

The fitness functions involve calculating the number of errors in the potential solution using the constraints that weren’t already satisfied by the initialisation function. Here [23] there are 4 constraints defined with an added constraint of the original numbers in the puzzle remain in their original position, something that [22] this implementation achieves in the fitness function.

This means that these two stages in the GA must allow for the 4 constraints in the puzzle to be met, the splitting of these constraints between the 2 steps should make no difference to the outcome of the algorithm.

The third stage of GA, selection, can be described as crossover and mutation due to that being the processes used to create the new population. When using GA for Sudoku solving, the constraints that were satisfied in the initialisation stage of the algorithm must be upheld during the crossover and mutation stage. In [23] the constraints handled in initialisation are 3x3 sub-boxes and that original numbers remain in their positions. This means that when crossover occurs, it must occur in the 3x3 boxes to ensure they stay valid but also make sure an original number is not one of the numbers being changed.

## Existing Systems

There are a range of research papers comparing Sudoku solving algorithms, but the algorithm choices and methods of comparison differ for each one.

The paper [19] uses different types of stochastic algorithm and combinations of these algorithms for comparison that result in finding what algorithms are able to solve Sudoku problems and which are not. The use of only stochastic algorithms is useful as it is algorithms that have never been used before for solving Sudoku and therefore gives an idea of which of them can successfully solve a puzzle and which cannot. This, however, only considers stochastic algorithms but other types of algorithm may be more efficient or quicker.

This project [13] also contains a variety of stochastic, backtracking, exact cover, and constraint programming algorithms that can be compared but this implementation takes more of a generic approach to algorithm design and therefore is more of a test of what algorithms can successfully solve Sudoku puzzles. This means that it allows for a wide selection of algorithms to be compared against a Sudoku puzzle but doesn’t get any deep analysis into the algorithms as it is just an implementation.

This research [24] looks at backtracking, constraint programming and rule-based algorithms for comparison of their efficiency. Rule-based involves using rules used by humans to solve Sudoku which allows for an interesting comparison. However, there is no implementation of any stochastic algorithms for comparing against the backtracking or constraint programming.

Here [25] there is a comparison being made between Brute-force, Backtracking and Dancing Links which are all derivates of brute-force and therefore make a good comparison. This, however, does not incorporate more complex and smart algorithms into the comparison as it only focuses on the more basic brute-force and its optimisations which misses out on some important comparisons.

This website [26] gives an empty 9x9 grid that can be filled in to with a puzzle problem and it will solve the puzzle as long as there is one solution. It uses rule-based algorithms for solving as using a well selected group of rules is very efficient at solving every kind of puzzle. The website also allows for checking whether a puzzle only has one solution and the difficulty based on which rules have to be applied for the puzzle to be solved.

## Implementation Technologies

### Python

Python [27] is an interpreted, object-oriented, high-level programming language that was conceived in the late 1980s, it is used by many of large organisations such as Wikipedia and Google for web applications. However, it has a big use in data analysis use to the effective use of scientific computing libraries such as NumPy, SciPy and Matplotlib.

The comparison between algorithms can be simplified by using the already existing libraries and it allows for easy visualisation of the data found which can be very useful for comparing the algorithms. There are also good libraries for both Genetic and Simulated Annealing that allow the implementation of the algorithms to be simplified.

### Visual Studio Code

Visual Studio Code [28] is an Integrated development environment (IDE) for developing software using a range of programming languages, it integrates with Github [29] to allow for pushes to be made within the application itself. An advantage is that it allows for extensions to be added to enhance the software development phase. There is an extension for Python [30] that allows for linting and debugging of Python code which helps with development of software in Python.

An advantage of using this environment is that it is well supported and well designed to allow development to be as easy as possible and therefore is easy to use and understand which is useful when trying to learn a new language.

### NumPy / Matplotlib

NumPy [31] is a Python package for scientific computing that is based around using N-dimensional arrays and high level mathematical functions for operating on the arrays [32], It also allows for integrating with C/C++ and Fortran.

This is useful as it allows for array manipulation and Sudoku puzzles are usually stored in arrays when being run in algorithms. To make use of this efficient data structure for Sudoku solving, the use of NumPy greatly increases the way these algorithms can solve puzzles.

Matplotlib [33] is a comprehensive library for creating static, animated, and interactive visualisations in Python. It allows for easy data visualisation and makes use of numerical mathematical package NumPy.

An advantage of this is its use in showing and visualising data, the data for algorithm comparison will be displayed as a set of numbers. This will make it hard to interpret the data to find which algorithms are best. Therefore, using Matplotlib can output graphs of the algorithms for comparison against each other very quick and simply for easy analysis.

### Tkinter

Tkinter [34] is the Python standard Graphical User Interface (GUI) package and even though it’s not the only GUI toolkit it is the most common one used. Since it is a GUI package it allows for the user to interact with a program by using buttons and can output information into windows.

This is useful for the type of implementation that is being done as it allows for the user to not have to use the command line for inputs into the software and for data visualisation to be easier due to it being output in a graphical format.

# System Design

This chapter describes the requirements of the system in order to design a solution that solves the problem. It also shows the design process that will be used for the creation of the tool in the next chapter.

## Requirements

The requirements for the project are listed below:

1. The tool should provide an implementation of multiple sudoku solving algorithms.
2. The test bed should allow puzzle generation and algorithm execution.
3. It should provide the ability to generate sudoku puzzles.
4. The user should be able to choose the number of puzzles to be generated.
5. It should allow for algorithms to be compared.
6. The algorithms that are to be compared should be able to be selected by the user.
7. A GUI should be provided that allows Sudoku generation and algorithm comparison.
8. Results of comparison using time taken and iterations should be output to the user in a readable manner.

## Methodology

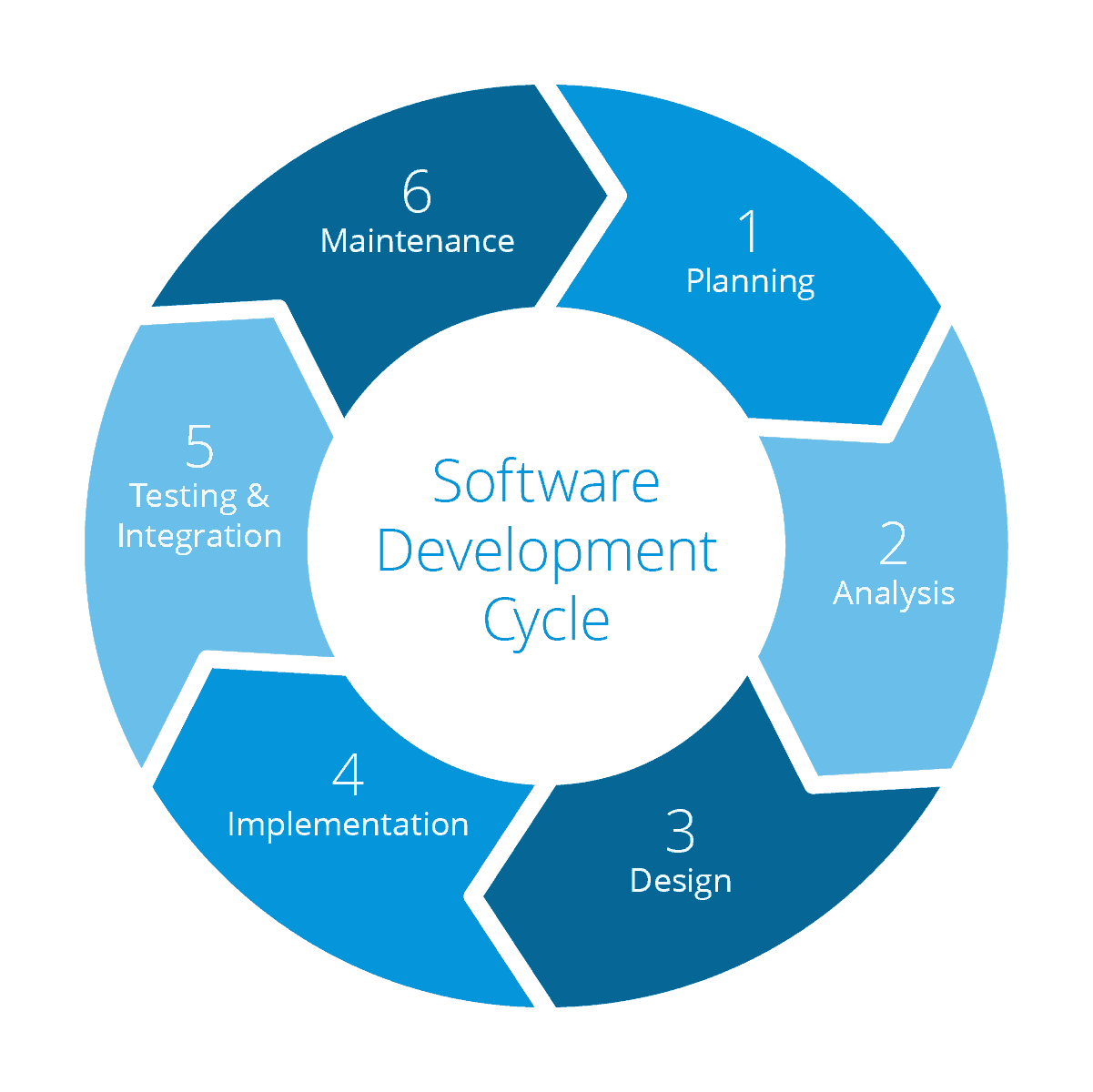
The design methodology is the logical and systematic method for proceeding with the design process [35]. For this project due to the way development should progress, the agile software development methodology was adopted.

Figure 3.1 Software development cycle [43]

Agile, seen in fig 3.1, involves breaking a project up into small sections and involving constant collaboration with the client making improvements each time. This fits well with the project due to having regular meeting with my supervisor who is acting as the client during this development and being able to make changes based on his feedback to the development.

Also, it aligns well with the incremental nature of the project due to each component being almost completely independent from each other as the puzzle generation, and each algorithm is implemented separately. This also means that the testing can be done on each component after it is implemented which is one of the bases of agile.

## High Level Design

The first step for designing the system is to identify the main components that need to be implemented and how they connect together. The key requirement that the high-level design needs to cover, seen in fig 3.2, is the test bed being the central hub for all the other components to interface with. This allows the tool to be dynamic based on the algorithms that are selected to be compared by the user and the number of puzzles to be created. The algorithms can then be called one at a time through the algorithm component that can interface with all the possible algorithms for comparison. Another important connection is the GUI as it interfaces only with the test bed making calls to the rest of the system based on the inputs from the user.

Figure 3.2 High Level Design of System

A brief description of the high-level components of the system:

GUI: This is the front end of the tool that allows the user to select algorithms for comparison and number of puzzles to be generated.

Test Bed: It acts as the core of the system linking the lower level puzzle generation and algorithm implementations to the GUI

Puzzle Generation: The component that generates the puzzles based on the requests from the test bed.

Algorithms: This connects the test bed to the actual individual implementations of the algorithms.

## Test Bed Design

### Design of Graphical User Interface

The user interface design is a vital part of the design and is based on the idea that the interactions between the user and the system need to be easy, meaning that the user should be able to navigate the user interface without the need for a tutorial or guide.

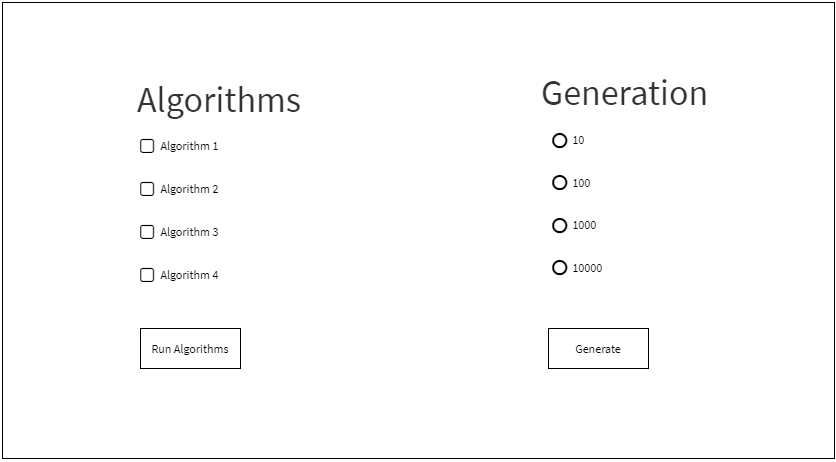
The user interface will consist of a main page with all the necessary inputs for the user to make. It will consist of 2 main aspects, with one side allowing the user to pick the algorithms that are going to be compared and the other side letting the user pick the number of sudoku boards to be generated.

Figure 3.3 GUI design for System

The algorithm selection on the left uses check buttons so the user can select from one of the algorithms to all of the algorithm depending on the comparison that is wanting the be made. After the desired algorithms have been selected the user can press the “Run Algorithms” button to run the selected algorithms over the test data puzzles that have been generated. The puzzle generation lets the user select a number of puzzles to be generated, then when generate is clicked the program will generate that many puzzles and output a message to the user when it is finished.

This allows for a user of competent knowledge of what the system does to interact with the tool without any walkthrough of how the interface works and allows for the user to start using the system quickly.

When the algorithms are run, the user interface will output the results in a graphical way in a separate window for the user to analyse. It should show all the raw data for the use as well as some statistics which are more readable.

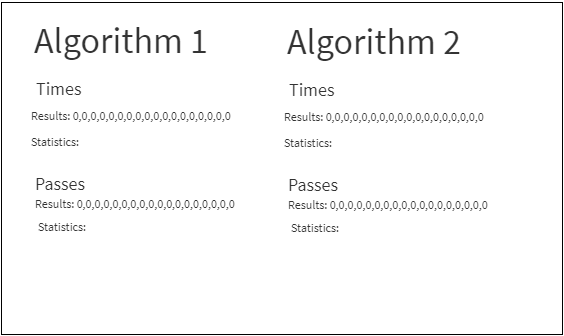


Figure 3.4 Mock of output GUI

Results for each algorithm that it tested is shown here, fig 3.4, in the results section for each of the 2 ways that we are measuring the algorithms. This is then followed by some statistics which could consist of the mean or the highest and lowest values which will give the user a simpler indicator of the efficiency of each algorithm.

### Design of Algorithm and Puzzle communication

The key feature of the test bed is to allow for the user interface to execute features that are implemented further down the hierarchy. It will take inputs from the GUI and use these to make requests to other areas of the system to get outputs that can then be output to the user interface.

This part of the test bed will also be the main way for the algorithms and the puzzles to communicate and be linked with each other. This is a challenge as the way they are linked can be done in different ways. The way that has been decided in the design is to have the ability to call a single algorithm into the test bed space, this is then held as the test bed extracts one puzzle at a time from the puzzle side of the system and passes it to the algorithm. This means that as little is taken by the test bed at a time and the rest is stored further down the hierarchy.

The final functionality of the test bed is the analysis of the data which will occur after all algorithms have been executed and their results returned. The design challenge with this is how much of this can be done in the GUI as the results are displayed graphically and therefore the analysis might be done within the graphical section rather than the algorithm communication.

## Design of Puzzle Generation

Puzzle Generation is responsible for generating the test data that is used to test the algorithms and is therefore responsible for the accuracy of the results that come out of the algorithms. The main challenges in this section is how the puzzles are being stored for the test bed to retrieve them and how the test bed is interfacing with the puzzle generation in the first place.

To solve the first challenge, we first need to look at what the different options are for storing the puzzles. One option is to generate one puzzle at a time and return it to the test bed, which can then run the algorithms on the Sudoku before requesting another puzzle. This has the advantage of the puzzles always being different every time the system is run. However, it has the downside of most likely taking an unreasonable amount of time to test against the simplest set of algorithms due to it having to run puzzle generation after each iteration of the algorithms.

The second option is to generate all the puzzles ahead of time before the algorithms have been run. Then store all the puzzles in a single file, ready to be iterated through by the test bed for each algorithm. Although this will overall take the same time as the first option, it allows for the test data be potentially be generated once and be used multiple times by the algorithms to gain more concrete results which would result in better data from the algorithms.

Puzzle File

Puzzle Generator

Test Bed

Test Bed

Figure 3.5 Design of interaction between test bed and puzzle generation

Here we can see how this would work in a high level, with test bed making calls to the puzzle generator based on the number of puzzles requested by the GUI. This then generates the Sudoku puzzles and stores them in the puzzle file formatted so each puzzle is on a separate line. When the test bed needs the test data to run the algorithms, it fetches the puzzles from the puzzle file while not interacting with the puzzle generator due to it not being needed at this point.

## Design of Algorithms

The algorithm implementations are all linked by an algorithm base that allows for each algorithm to be called with a single puzzle to be solved and the results stored back in the algorithm base, before being returned to the test bed.

The flow charts are all made with diagrams.net [36] and are all based off designs for the algorithms found in past research.

### Backtracking

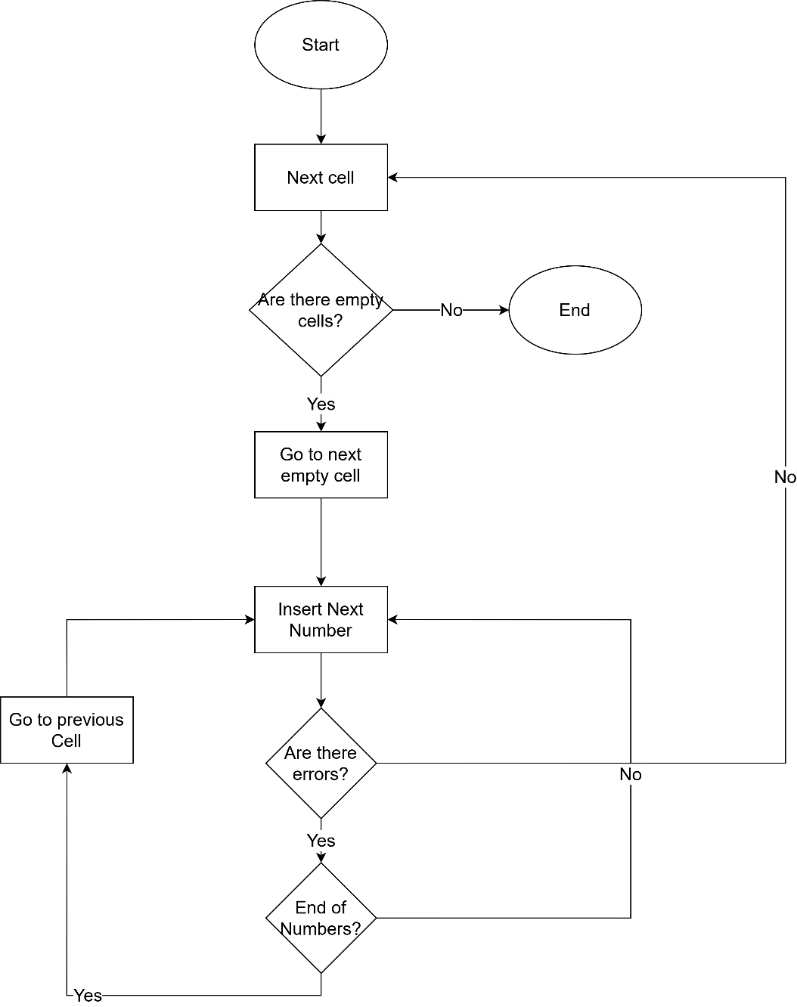
After backtracking is passed a puzzle to solve, the algorithm starts at the top-right corner of the puzzle and loops through each empty cell making sure the numbers it adds do not cause errors. If this happens it backtrackings to the previous cell and keeps trying new numbers.

Figure 3.6 Flow chart of Backtracking

The design challenges for this algorithm are how the backtracking will work and how it will progress and whether the exit condition defined in fig 3.6 is correct. Since this is a simplistic algorithm most of the design is straight forward with the backtracking being the hardest concept. The algorithm needs to be able to hold the current state of all the previous cells in the chance that the algorithm backtracking to that cell and needs to continue from where it left off.

The easiest method of achieving this is to use recursion, with a different call for each cell, as it allows for the state of a cell to be put on the stack. This can then be popped off the stack if the algorithm comes back to that cell. This allows progression through the algorithm as cells are popped and pushed to the stack.

The exit condition of the recursion is there being no empty cells as once the algorithm iterates through all the cells and tries the cell after the last one, it will meet the exit condition and since the puzzle is solved up until then it can be returned as solved.

### Hill Climb

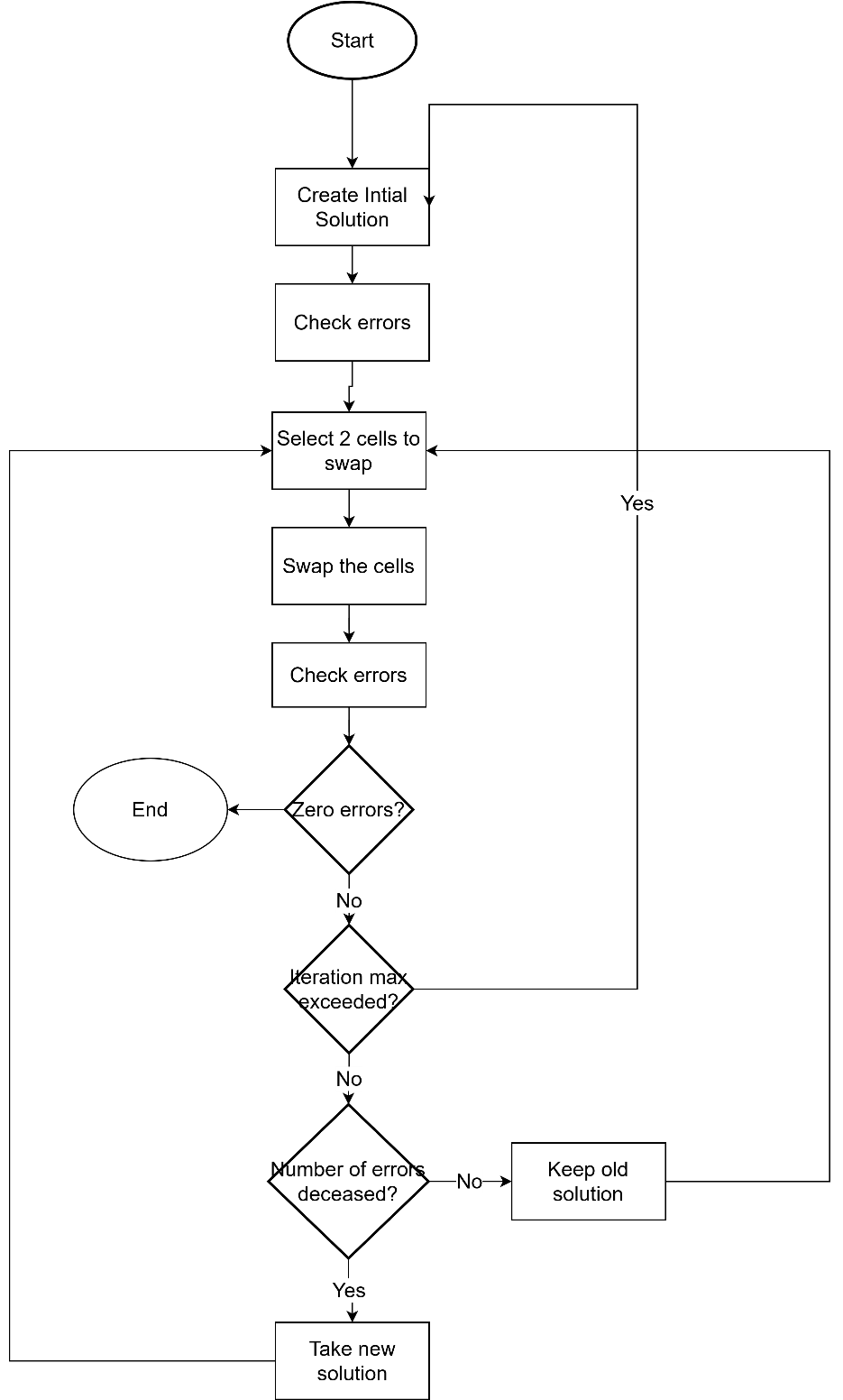
Hill Climb involves moving through different solutions for solving the puzzle, each time checking for errors and only moving to a new potential solution if it has less errors than the previous solution.

Figure 3.7 Flow chart of Hill Climb

Initial solution creation in this algorithm is important as how the errors are calculated is dependent on this. When it is initialised, some of the constraints of a valid Sudoku solution must be met. The design decision made is to meet the 3x3 sub-box constraint and fill each of the 9 boxes with all the values from 1-9. Although this is the hardest initialisation method, it allows errors to only be calculated by using the rows and columns of the potential solution which is more efficient.

This is an important decision in the design as the initial solution is created infrequently whereas the error checking is done every iteration of the puzzle and therefore much more often than initialisation.

Error checking needs to be designed well as it will be used throughout multiple solving algorithms. The easiest way is to find which numbers are missing from each row and column then collect all these missing numbers and count them to give the number of errors found in the solution.

Having a max iteration number allows the algorithm to get out of local minima even though it is the most basic way. Since Sudoku has only one solution and these algorithms are used to find an optimum solution, there is only one correct solution. If the algorithm starts going down a dead end path, then it will not reach a solution and the solution will need to be reset to allow the algorithm to try again.

### Simulated Annealing

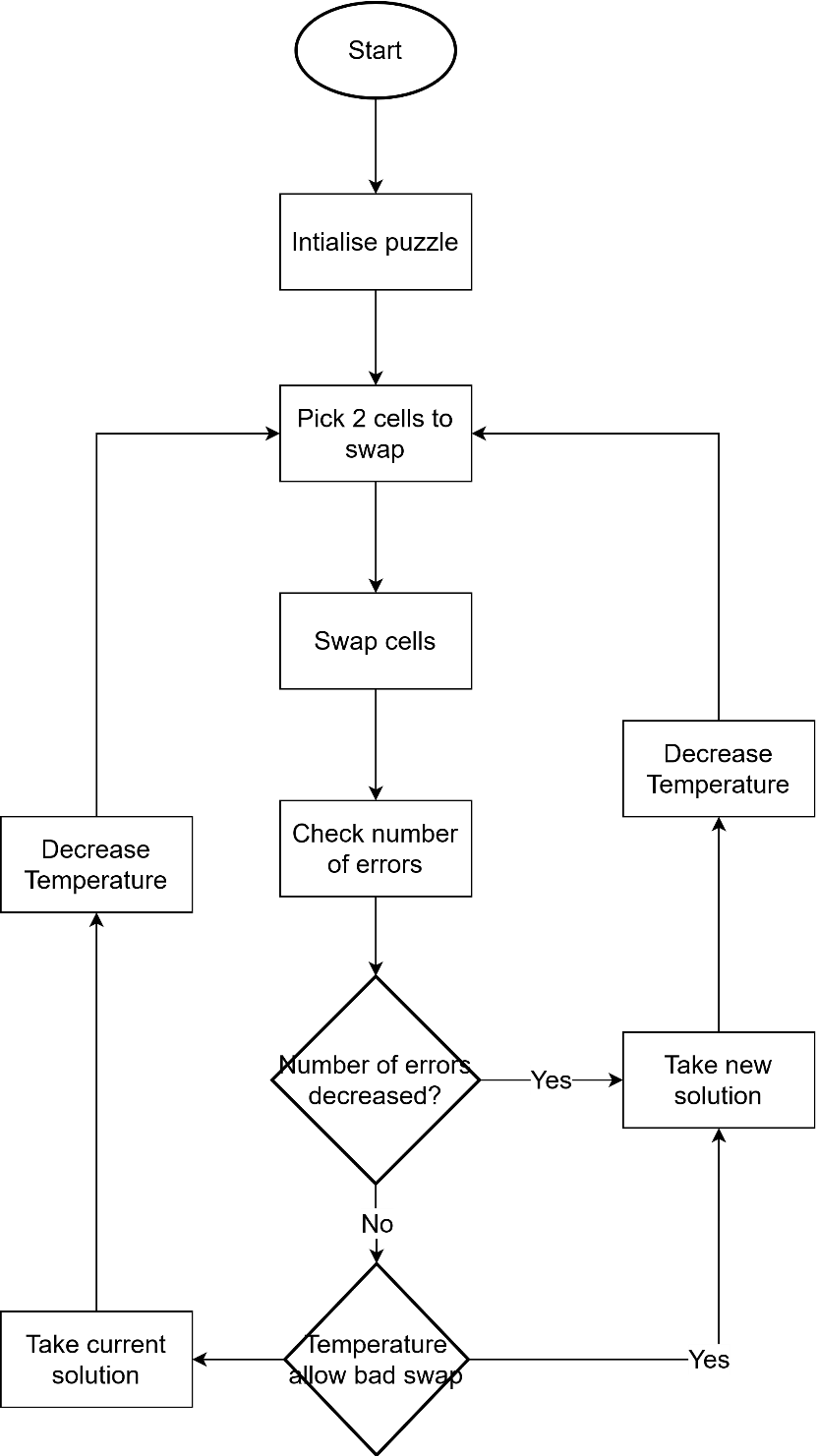
Simulated Annealing is an algorithm that is an optimization of Hill Climb but instead of only moving to a better solution based on a probability which comes from a temperature.

Figure 3.8 Flow chart of Simulated Annealing

Simulated Annealing is an optimization of Hill Climb and offers a lot of the same functionality with more improvement on getting out of local minima. The improvement involves using a temperature to decide whether what solution to take into the next iteration.

Temperature is used to measure the chance of the algorithm taking a worse solution into the next iteration. This is important as it allows the algorithm to avoid local minima early in the algorithm by moving back to solutions with more errors. The design challenges of this are how it is added to work effectively and do the job of optimizing Hill Climb.

Since the puzzle is only initialised once, the temperature needs to be correct to allow for sufficient exploration of the solution space but also enough to converge on an actual correct solution to the puzzle and this will be a challenge.

### Genetic Algorithm

Genetic Algorithms generate a population of many potential solutions, the fitness of each one is calculated, and it takes the best solutions and uses them to create a new population. This keeps iterating until a solution is found.

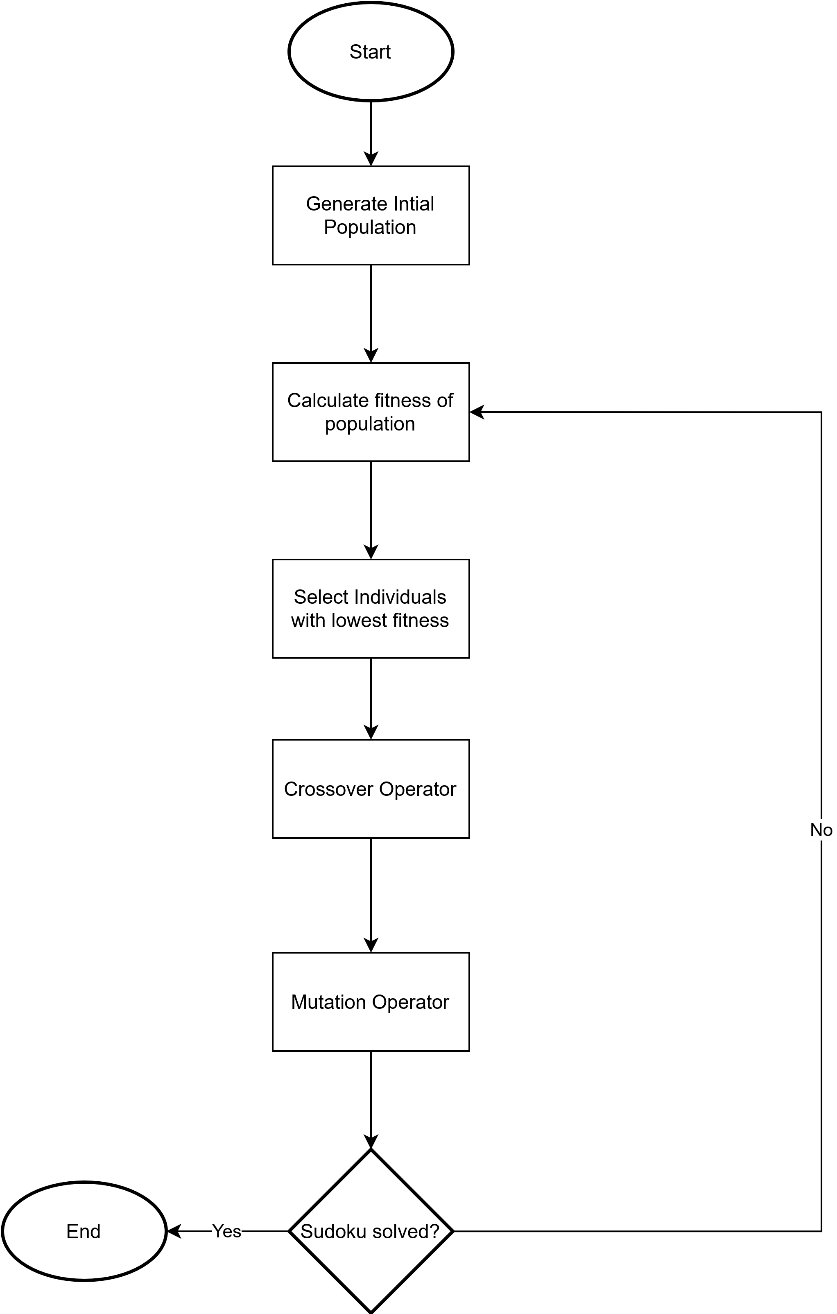


Figure 3.9 Flow chart of Genetic Algorithm

Genetic algorithms contain many different challenging aspects to it, because of this many different decisions needed to be made. The first is how the implementation will be structured due to the need for a population. Using objects here for each solution allows for the storing of fitness’s as well as the population being a group of objects makes crossover and mutation easier.

Crossover and mutation are important parts of the algorithm as they allow for changes to be made to the solution to move closer to a complete solution. Crossover takes 2 individuals and crosses them over to create 2 new child solutions that are used for the new population. Mutation takes a child from after crossover and makes a mutation change to it e.g. changing a single cell in the same way that children are slightly different from their parents. This mutated child is then used for the next population which will calculate fitness’s again unless one of the individuals has a solved Sudoku, which is the terminate condition.

The design for crossover will involve splitting the two solutions in half and swapping them dependent on how they have been initialised to ensure that the constraints that were met during it will stay valid. This means using either the row or column constraint when initialising which would allow for splitting of the solutions.

Since mutation is designed to be done within a single solution, it is possible to do this the same way as in Hill Climb and Simulated Annealing with little change except the constraint being used.

The last challenge will be in error checking, with a different constraint being used the error checking will be hard to accomplish and will require a different method to the previous algorithms. The design of this will be similar but with needing to check sub-boxes which will be new.

# Implementation

This chapter shows how the implementation of the system that follows the designs in the previous chapter. It will describe in detail the different parts of the design and how they were implemented and how this meets all the defined requirements. The first component to be implemented is the Sudoku Generation which will be creating the test data, followed by the different algorithms for solving the puzzles and finally, the Test Bed to bring all the other components together.

## Sudoku Generation

The main idea for this is to provide a way for test data to be generated in some way to allow the algorithms to be tested using the puzzles. There are two ways as will be seen in versions 1 and 2, the reason for the two versions is this is an important part of the testing of the algorithms and getting an more simple version of this working early allows for the rest of the project to run more smoothly. Version 1 involves retrieving pre-existing puzzles from the internet and passing them to the algorithms whereas version 2 creates the puzzles within the project and does not need to use outside sources.

### Version 1

Due to the iterative way that the project had been designed, this was a simplistic design that used a single class with most of the work done manually. Using [37] each refresh gives the user a new puzzle which can then be copied into a text file with the empty cells replaced with zeros. This Fetch class, when called can then get a line from the text file and return it to be tested on an algorithm.

This version was found to create problems within the system due to puzzles being manually added and this taking time. The amount of time this would take would not result in enough test data to ensure that the results from the algorithms were sufficiently accurate, meaning a new version needed to be created that could generate almost infinite test data automatically.

Also having the same puzzles being solved by the algorithms will impact the results produced by the algorithms as, even though almost all the algorithms are non-deterministic, backtracking will always have the same results from the same test data.

### Version 2

In this iteration of Sudoku Generation, the puzzles are generated by the algorithm and then stored in a text file that the Test Bed can read out of when algorithms need to be run. This is all done by a single class with a main function to loop the generation for the number of puzzles requested by the Test Bed.

The initial puzzle filling algorithm is seen in fig 4.1. It uses Backtracking which is one of the already implemented algorithms due to its ease of use in easily creating randomness. Each new cell that tries to be filled gets a new random order of the numbers 1-9 to ensure extra randomness for the solution. Outputted is a completely random valid puzzle out of the over 6 sextillion [38] possible combinations

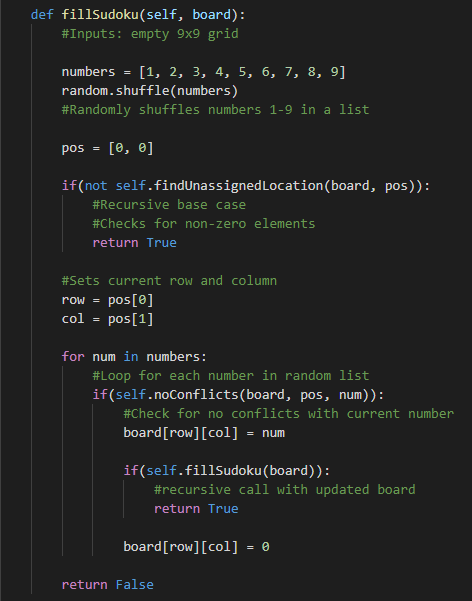


Figure 4.1 Code snippet for filling Sudoku board

Cells are selected from the puzzle and removed; the puzzle then has to be checked for having multiple solutions as a valid Sudoku puzzle must only have one answer. This again uses Backtracking as a brute-force algorithm needs to be used here to uniformly search the solution space.



Figure 4.2 Code snippet for calls for solving puzzle in puzzle generation

Figure 4.3 Code snippet for order of numbers used in puzzle generation

As seen in fig 4.1, Backtracking uses a list containing the numbers 1-9 to solve, this determines the order that the numbers are tried in each cell. When searching for multiple solutions, it is important to search the whole solution space for two solutions since as long as there is more than one then the puzzle is not valid.

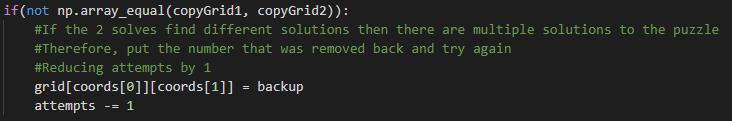
In fig 4.3, we see that two lists are defined, ascending and descending numbers 1-9. This, in fig 4.2, is passed as a parameter to a solve function as well as the puzzle. In order to search the whole solution space, we use ascending order to start at the top and work down; and we use descending to start at the bottom and work up.

Figure 4.4 Code snippet for comparing results of ascending and descending solves

They uniformly search the space for solutions which are then returned and compared a seen in fig 4.3. If the results from both solves are the same, then both ascending and descending have reached each other somewhere in the middle and therefore we can conclude that there is only one solution

Initially, it was planned to have the ability to create different difficulties of puzzle and this would be done by increasing and decreasing the value given to “attempts” with more attempts allowing for more chances for numbers to be removed. However, changing the “attempts” variable did not change the difficulties of the puzzle and therefore it was decided to test using a single difficulty of puzzle that could be generated by the puzzle generation algorithm.

## Algorithms

Algorithm implementation is the main section of the system as it creates the data that allows the comparisons to be made and is the reason for the other sections of the system to be created. Algorithms were implemented in a specific order from easiest to hardest, this is due to the possible time constraint on the project and therefore it was important to have the easier algorithms completed to allow for more time to implement the more complex ones. This also meant that if development time were running out and the harder algorithms could not be implemented successfully, a comparison with the other algorithms could still be made.

NumPy arrays are used for each algorithm as they are more efficient for creating and accessing 2d arrays which makes the algorithms run faster and more efficiently.

The algorithms build on each other with Backtracking being brute-force and Hill Climb being the easiest type of Stochastic algorithm, Simulated Annealing is a Hill Climb optimization and Genetic being an improvement on Simulated Annealing.

Each of the algorithms contains a “runAlgorithm” function that is called by the Test Bed when the algorithm needs to be run. In this function the time before and after the algorithm is run are taken and in each of the algorithms there is an iterations counter that is returned with the time taken after the algorithm has returned.

### Backtracking

Backtracking is the brute-force method for solving Sudoku as it searches through every potential solution until it reaches one that is valid, but it does it in a smarter way than generic brute-force. The utilisation of recursion within the algorithm allows moving back to previous cells with their state’s saved easier.

The algorithm has a simplistic design and therefore the implementation is mostly straightforward with a couple of aspects that are challenging. The first challenge involved detecting errors as a important part of this algorithm is ensuring each number entered does not cause conflicts within the puzzle.

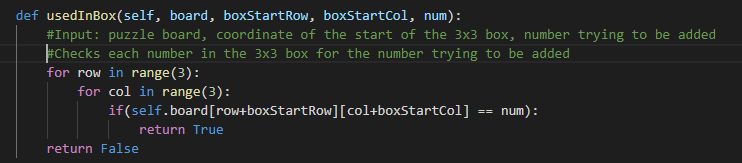
Checking conflicts within sub-boxes involves using the current row and column to find the top-right corner of the sub-box as seen in fig 4.5. Then in fig 4.6 the function extends two cells right and down and checks all these cells for the number being checked.

Figure 4.5 Code snippet for getting sub-box of cell

Figure 4.6 Code snippet for iterating through sub-box

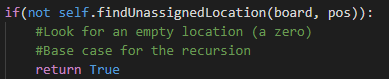
The backtracking using recursion works by recursively calling the solve function for each new cell that is being worked on. This results in the algorithm stack containing a waiting process that can be continued with if it is popped off the stack, meaning all the progress on the individual cells can be left was it was.

Figure 4.7 Code snippet of Backtracking recursion base case

The recursion ends after the algorithm finishes the last cell and no empty cells are left fig 4.7, allowing all the states left on the stack to be popped off using the return true in fig 4.8 to communicate to each process on the stack that has been solved. This creates a waterfall effect from the last cell returning true to the solved puzzle being output.

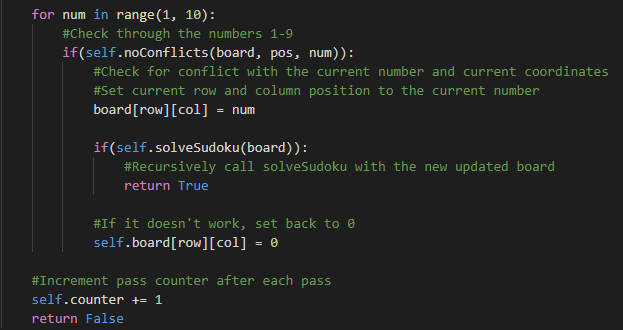


Figure 4.8 Code snippet for backtracking fixed loop

### Hill Climb

Hill Climb involves incrementally moving towards the correct solution each time moving closer to the optimum solution like climbing a hill to reach the top. It is more efficient than brute-force for finding the optimum solution for a problem as it only increments towards the solution and does not check the whole solution space.

Initialisation of the puzzle is important as it important for making changes to move towards a solution and for calculating energy, It also is used in all stochastic algorithm which all the other algorithms implemented are.

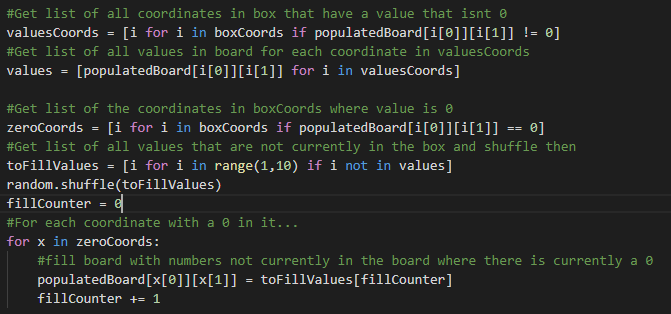
Initialisation is implemented by making every 3x3 sub-box contain the numbers 1-9. For each box, the fixed numbers are counted to determine what numbers are missing. These are then added to all the zero value slots at random, allowing all the numbers to exist in each sub-box, this is seen in fig 4.9.

Figure 4.9 Code snippet for puzzle initialisation

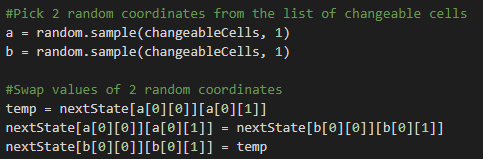
Climbing towards an optimal solution is done through a pair of functions, one to make a change between the old and new solution and the other for calculating the errors. As seen in fig 4.10, for moving between solutions two random non-fixed cells are picked at random and subsequently swapped to create a new solution.

Figure 4.10 Code snippet of swapping of cells in Hill Climb

Error calculation works by using sets to count the number of unique values in each row and column, as sets cannot have duplicate values. Each row and column is then taken and the difference between the length of the set and 9 is added to the score, as if the length of the set is 9 then all the numbers are correct in that row and the value added to the score is 0. The score is then returned to the calling function to give the number of errors present in the puzzle.

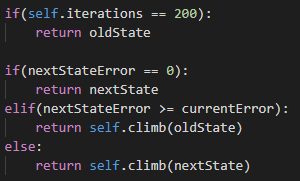
The errors of the old solution and the new solution are calculated by calling the energy function and then compared again a set of statements, fig 4.11. If the maximum number of iterations has been reached, which is set to 200, or the number of errors is 0 then the puzzle is returned, and it handled by the first function. The second part is based on the number of errors in both solutions and depending on that, the more optimum solution is taken into the next iteration.

Figure 4.11 Code snippet for moving to new states in Hill Climb

Iterations allows this algorithm to deal with finding local minima as there is a high chance when solving a Sudoku puzzle that the algorithm will go down the wrong path towards solving it. To fix this, the algorithm must loop back to re-initialise the puzzle and start again in order to find the correct path.

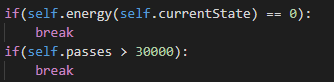
Due to the algorithm finding local minima, the algorithm could run indefinitely without finding the correct path towards the solution due to the randomness of moving. Therefore, seen in fig 4.12, there is a timeout based on the number of iterations made through the algorithm that stops it from running infinitely.

Figure 4.12 Code snippet of timeout condition in Hill Climb

### Simulated Annealing

Simulated Annealing is an optimization of Hill Climb and therefore inherits a lot of the same traits as the previous algorithm. The initialisation of the solution stays the same with each of the sub-boxes containing the numbers 1-9. The difference is that is initialisation only occurs once at the start of the algorithm not multiple times as it does in Hill Climb.

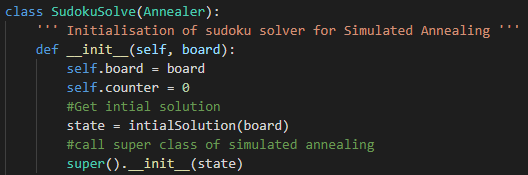
When implementing the Simulated Annealing algorithm, there is a Python library [39] that takes 2 functions and a set of parameters and performs the Simulated Annealing on an array. This is one of the reasons Python was chosen for the development as writing the code for reducing the temperature would take potentially more time than was available. This library is inherited by a class, seen in fig 4.15, and the 2 functions called move and energy are written in this class, which are almost identical to that in Hill Climb.

Figure 4.13 code snippet of initialisation of SudokuSolver class for Simulated Annealing

When running the annealing the library requires a set of parameters to control temperature changes. These is they are set wrong will result in the puzzle not being solved.

These parameters define 4 things:

* Starting temperature
* Ending temperature
* Number of iterations
* Number of updates

Temperature starting and ending are important as the temperature defines the chance that a bad solution is accepted by the algorithm. If the starting temperature is too high then bad solutions are accepted for too long and a real answer is never reached, if too low then the solution space to not wide enough to be able to find the correct solution to the puzzle. Ending temperature must also be correct to allow for enough convergence of solutions to each the optimum solution to the problem without stopping too early.

The number of iterations is the number of times for each temperature that there is a move-energy cycle using the solution acceptance rate for that current temperature. If this is too low, there will not be enough iterations to get closer to the solution and too high and the algorithm will take too long.

In solving Sudoku, there needs to be wide search space to allow for many different paths towards the correct solution to be taken due to Sudoku having a number of local minima - a potential solution turning out to be wrong only when the algorithm thinks it has almost found the solution. However, the ending temperature must be low as once the correct path is found the algorithm must converge very quickly towards it to allow for many attempts to be made to each the solution to the puzzle.

The number of iterations for solving Sudoku must also be large again due to there being many local minima. Also when the algorithm gets close to the solution and will only accept lower error solutions, there are few moves that will cause a better solution to be created meaning many attempts need to be made to ensure that these moves are found before the algorithm times out.

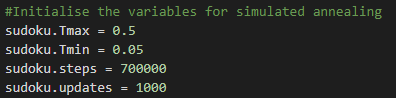


Figure 4.14 Code snippet of Simulated Annealing parameters

Figure 4.15 Code Snippet of Simulated Annealing parameters

As seen in fig 4.18, the starting and ending temperatures chosen are 0.5 and 0.05, giving an over 90% acceptance rate at the beginning to just above 0% at the end, allowing an adequate solution search space. The number of iterations chosen is 700,000 which is very large due to Sudoku puzzles having one solution rather than a global minimum which is what the algorithm is commonly used to look for. The updates variable refers to the number of updates made to the output that is shown when the algorithm is run which gives an idea of how close the algorithm is to completion.

### Genetic Algorithm

Genetic Algorithm (GA) is the hardest of the algorithms to implement due to its overall having more complex components to it and that even though there is a Python library containing an implementation of GA it cannot be effectively used for Sudoku solving. This is because it is focused on being used for a more general GA use which is for finding the global minimum of a problem.

Therefore, a full implementation is needed and although there are similar systems available for reference the time constraints did not allow for this algorithm to be completed successfully, although all the base logic was completed.

This algorithm implementation makes use of objects well with each individual being an object and therefore a population consists of a list of individual objects, making it easy for each individual to have its own fitness value. The population can then be sorted using the fitness’s of each individual to give the strongest candidates to mutate and crossover.

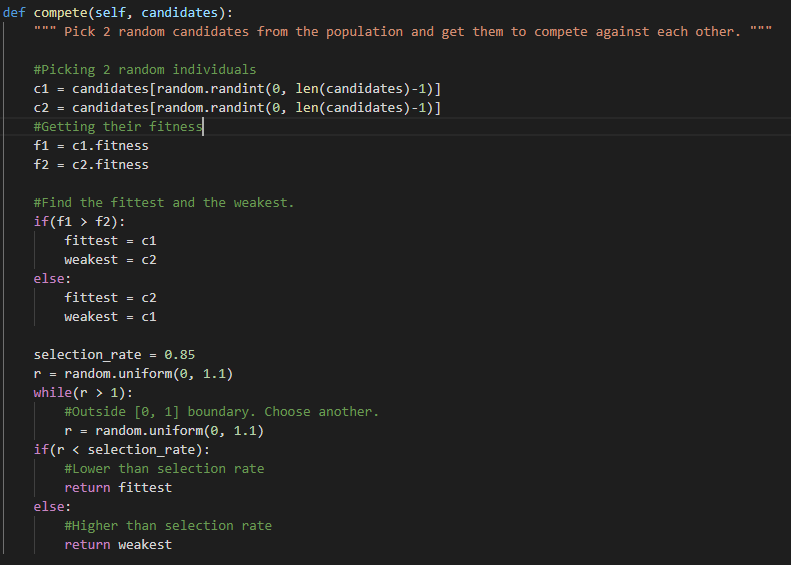
A number of processes are done to the population when the new population is being created. The first is in the tournament class that takes 2 individuals and using a selection rate either choses the fitter or weaker individual to ensure that the population as a wide solution space and does not converge on an answer to quickly.

Figure 4.16 Code snippet of compete function in Genetic Algorithm

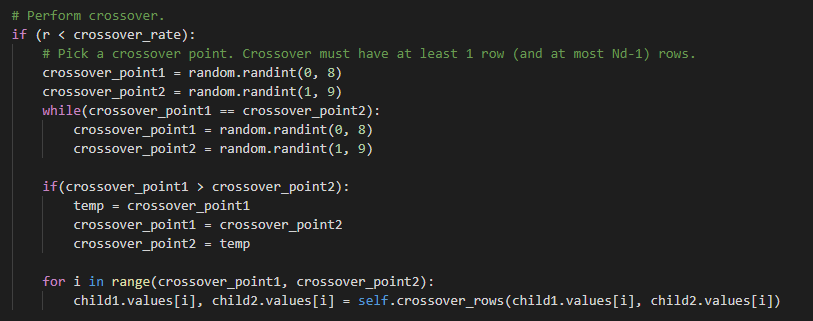
The algorithm uses the same method of initialisation of puzzles as the previous algorithms except it uses the row instead of the sub-box constraint for always keeping correct as it makes crossover easier to accomplish. Using this, crossover takes 2 parent individuals from the population and selects 2 rows in the puzzle. Then all the rows between the two points are swapped from one individual to the other to create 2 new child individuals.

Figure 4.17 Code snippet of crossover in Genetic Algorithm

After the crossover, mutation is done in a very similar way to the climbing in Hill Climb and move in Simulated Annealing except it uses rows instead of sub-boxes. A random row is selected and then within that row, 2 values are identified that are non-fixed and therefore can be moved. These are then swapped based on a mutation rate which determines how likely this swap is to execute. This is done for every individual in the population after each one has gone through crossover, the result of this is a new population and the process can then be repeated.

All this logic is in place however there are problem with the syntax of the algorithm that prevents it from running. With more time it is possible to have this implemented, however, that will be in future work.

## Test Bed

The test bed consists of 2 components within it with the aim of bringing all the rest of the system together to give a result that is effective and readable. The first component is the interacting with the Sudoku generation and the algorithms, getting puzzles from the files and passing them to algorithms. The second is the GUI and the analysis that is done of the data, giving the user an easy way to interact with the system as well as showing the data in an easy and understandable way.

### Puzzle and Algorithm Interaction

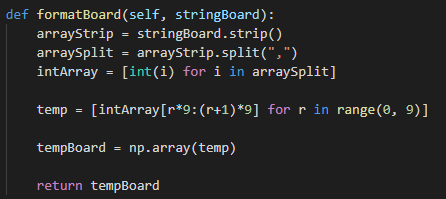
Puzzle Generation, as we saw previously in the chapter, puts the puzzles it generates into a text file that can then be read from, this is done here. First the file is opened, and a specific line is taken. This board is then formatted as it is stored in a one-dimensional list in the text file but needs to be a 2d NumPy array for the algorithms to run, first by splitting the list by the commas then by every 9 numbers and transformed into a NumPy array.

Figure 4.18 Code snippet of board formatting in Test Bed

This is then passed to the runAlgorithm function that is passed a number dependent on what algorithm is going to be run, then each puzzle from the text file is called one at a time and passed to the algorithm to be run and the results stored in arrays to be returned after all the puzzles have been iterated through.

### GUI and Analysis

The GUI and analysis are linked together due to the results of the analysis being output to the user via the GUI and therefore the analysis calculations are done within the GUI class in the implementation.

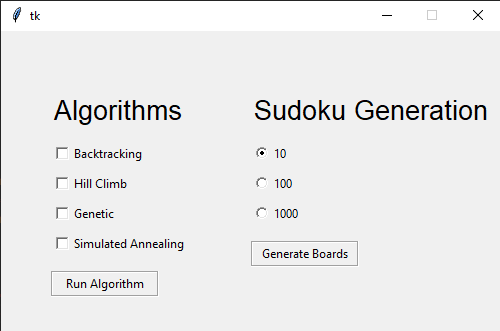
The GUI was implemented as per the design in the previous chapter due to it being very intuitive.

Figure 4.19 GUI for algorithm selection and puzzle generation

The algorithms on the left are checkboxes that enable the user to select any number of the algorithm that are to be compared and the right consists of which number of Sudoku puzzles to be generated using radio buttons with the current number puzzles selected. When algorithm generation is complete, “done” will appear below the button to inform the user that it has completed. Although this system can be used without the terminal, there is no way to update the GUI with the current progress of either the algorithms or the Sudoku Generation. Therefore, there are console outputs for a counter of the puzzles being generated as well as for the algorithms.

When Generate Boards is clicked, the function will get the current value that is selected in the GUI and this is passed to the Sudoku Generation component of the system. Run Algorithm calls a function that checks each of the checkboxes for whether they have been selected, this list of algorithms is then passed so that each algorithm can be called on all puzzles with the results all stored in an Analysis object.

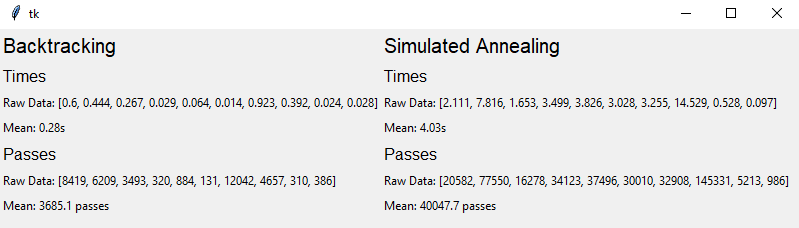
The results from the algorithms being run are displayed in 2 different ways both with graphics.

Figure 4.20 GUI for result display

The first is the raw data from the experiment, which comprises of the times and number of passes. These are in columns with a list of all the raw data shown as well as the mean for all the raw data, giving a clearer statistic for easier readability. The main is calculated using the Statistics Python library with the ability to provide other stats using the library.

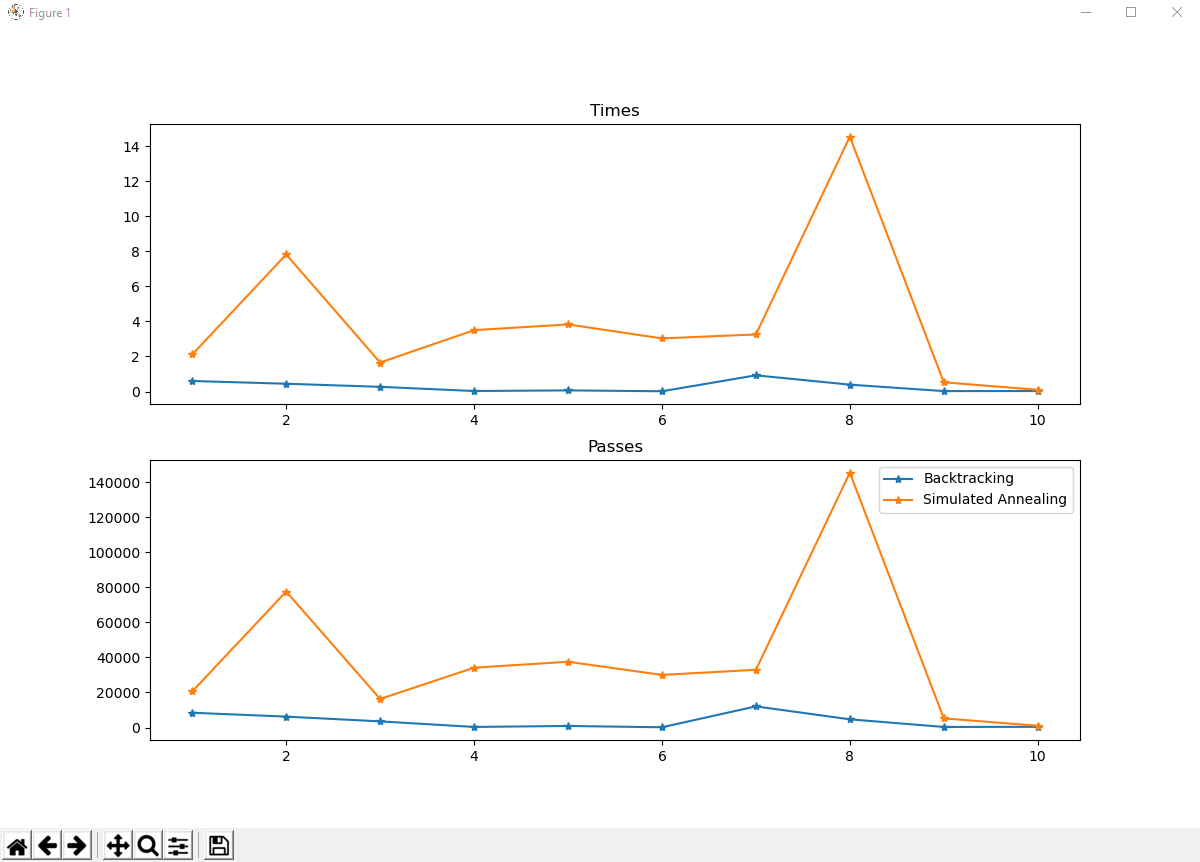
The second method of displaying results is through a graph.

Figure 4.21 GUI graph of results

Fig 4.21 shows the same raw data that is shown in the list, however, is a lot easier to understand which algorithm is the most efficient. This is done using the Python data visualisation library MatPlotLib which when passed an NumPy array of values and an x-axis scale can produce graphs such as the one above.

# Testing

Testing of the system is important as it allows us to make sure that the implementation that has been developed is working as intended and giving the correct and expected result. Exhaustive testing is done and even though no system is bug free, as many bugs as possible are identified. The structure for this testing is [40] which involves four stages: Unit, Integration, System and acceptance testing. We will focus on the Unit and System testing as they are the most important for the project, these tests are seen in full in Appendix A.

## Unit Testing

Unit testing is known as white-box testing and involves testing individual components of the software being tested, the purpose being to make sure each unit of the software is working. The components in this project are the methods in the different sections: algorithms, Sudoku generation and test bed.

Unit testing in Python can be done with a number of different libraries but the easiest is Unittest [40] which is integrated into standard Python. These tests use assert statements to check if actual output equals expected output. Unit testing was carried out continuously during the development by testing methods as they were implemented.

### Algorithm Testing

Unit testing for algorithms, Appendix A1, was used to ensure that methods within each algorithm was producing the expected outcomes. The testing is not able to fully test the algorithms due to the randomness involved in how the puzzles are solved and therefore it is hard to test this on a specific scenario and get the same result every time.

Between the algorithms, there is a large amount of overlap in the functions being used, e.g. puzzle initialisation, which means they only need to be tested once. Algorithm testing went in order of implementation so first was Backtracking then Hill Climb, Simulated Annealing and Genetic last.

Unit tests were also carried out on the main functions of the algorithms which acted as secondary tests for all the methods within the algorithm. Done by passing in a puzzle to solve and comparing the result to the excepted solved puzzle.

Figure 5.1 Output of algorithm unit tests

The output for the unit tests run on the algorithms are in figure 5.4, showing all the tests passed except for one failure and one error. The failed test involves the Hill Climb algorithm overall test as the algorithm itself does not always solve the puzzle before timing out due to the nature of the algorithm. This is expected as, if the algorithm times out it doesn’t return a solved puzzle and therefore fails. The error test is the Genetic algorithm test as the implementation is not complete and therefore doesn’t work and returns an error.

### Sudoku Generation and Test Bed Testing

Sudoku generation also has a large amount of randomness that inhibits the amount of unit testing that can be done on the component due to a different puzzle being created by it each time. There is also a large amount of overlap between Backtracking and generation due to the solving of the puzzles to find more than one solution is being done with Backtracking and therefore it does not need to be tested again as it has been covered in the algorithm tests.

The Test Bed also has limitations in unit testing due to the majority of the functionality in the test bed being embedded in the GUI. This means what system testing is the main method of Testing the Test Bed. These tests are located in Appendix A2.

Figure 5.2 Output of Sudoku Generation unit tests

Sudoku Generation test results are seen in fig 5.2, showing that all tests passed. This being a small number of tests is due, as discussed earlier, to the overlap with Backtracking and this component of the system having a small number of methods to be tested.



Figure 5.3 Output of Test Bed unit tests

The results for unit testing for the Test Bed is seen in fig 5.3. It only consists of retrieval of the board due to, as already mentioned, all the rest of the functions are embedded within the GUI and therefore can be tested in the system testing. The test involves opening the file, gets a specific line and formats it. It then calls the function and compares the results.

## System Testing

System testing is black box which means there is no knowledge of the inner code design, meaning was done using the GUI as this is what a user would see. Tests were written and carried out manually with results compared against expected outputs.

Within the GUI there are the algorithms and the Sudoku generation that need to be tested. The radiobuttons are tested by clicking each one making sure the selected number changes and the text file for storing the created puzzles can be checked for each number of puzzles tested. Sudoku generation tests are shown in Appendix.

The algorithm testing in the Test Bed is done in the same way as the Sudoku generation with the GUI tests on the check buttons as well as testing the calling of the algorithms with the correct boards.

For testing the results produced by the algorithms, the time taken challenging to test although outputs are given by the algorithms to the console and a rough estimate of the expected time taken can be used.

For iterations, putting the counter incrementor in the correct place for each algorithm should have been solved in the initial debugging phase during the development of each algorithm and the use of visual observation for the number it is producing can determine its accuracy.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Number** | **Input** | **Expected Output** | **Actual Output** | **Pass/Fail** |
| 27 | “Hill Climb” selected with 10 puzzles and “Run Algorithm” clicked | Numbers 0-30000 output to console 10 times, 2 windows open for data and graph | Numbers 0-30000 output to console 10 times, 2 windows open for data and graph | Pass |
| 29 | “Genetic” selected with 10 puzzles and “Run Algorithm” clicked | Numbers 0-9 output to console, 2 windows open for data and graph | Algorithm crashes and returns error message to console | Failed |

Table 5.1 Some tests for algorithm execution through GUI

Table 5.1 shows two important tests, test number 27 is the Hill Climb test which passes even though the it fails the unit test and doesn’t return the solved puzzle. This is because the Hill Climb algorithm still produces a time and iteration result and therefore can display these returns even though the algorithm itself did not solve the puzzle. Test number 29, the Genetic algorithm test, fails as the algorithm produces an error and therefore crashes before being able to produce a result.

## Interesting Bugs? – Testing Results

Given the testing that was done, there were a number of bugs that were found in the code. A few of these bugs are listed below:

* During testing large number of puzzle with Hill Climb and Simulated Annealing there was found to be an issue with swapping cells. There was a very unlikely case that caused the number of changeable cells in a box to be less than 2 and therefore when 2 where selected at random, the algorithm crashed. This was fixed by checking the number of changeable cells and if it was less than 2 then another box was picked.
* Testing of the GUI uncovered a potential issue that involved being able to click “run algorithms” without having any algorithms selected that resulted in an empty window of results and an empty graph.
* Sudoku generation requires copying the puzzle into new variables. Initially, using “=” did not copy across the whole structure due to the puzzle being an NumPy array. Therefore, “deepcopy” needed to be used which allows for an exact copy to be made even if there are objects within the thing being copied.

# Evaluation

This section discusses the results found from this project as well as an evaluation of the requirements that were outlined in chapter 3 for the implementation of the system.

## Results

When examining the results, it is important to consider that the stochastic algorithms are optimized as best they can for finding a specific solution however, their main purpose is for finding the optimum solution where there might be many.

There are two data points used for the results: the time taken and the number iterations, which both generate float and integer values respectively that can then be statistically analysed as well as plotted on graphs.

Using a list of results, statistics can be computed from this. The important statistics for each algorithm were the mean, variance and, minimum and maximum as it is possible that the best overall algorithm might have the best single result.

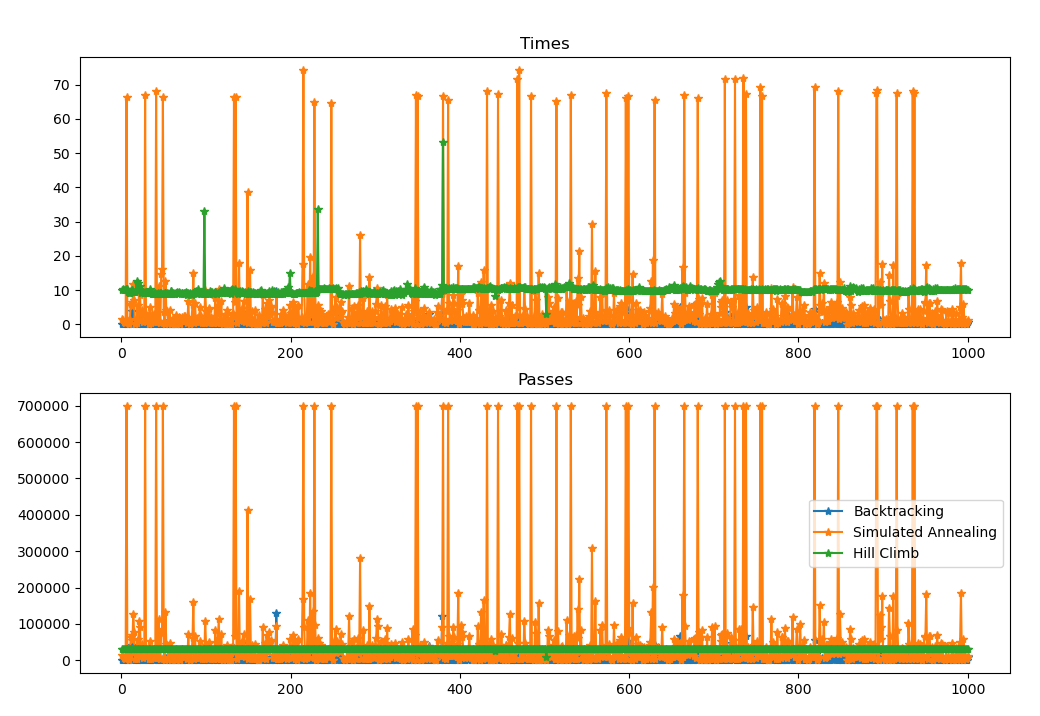
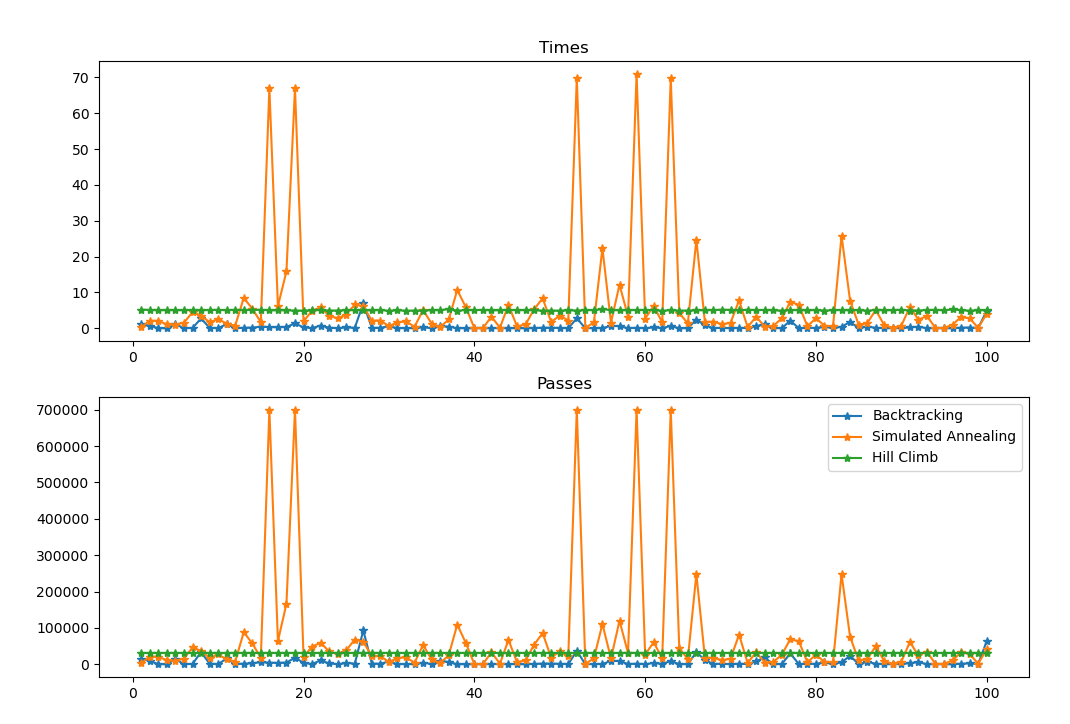
The most accurate results come from the largest dataset and therefore, the three algorithms were run over 1000 puzzles to give the results found in fig 6.1. However, these results are unclear due to there being too many points, meaning most of the data is unreadable although it does give insight into the performance of the algorithms.

Figure 6.1 Result graph for 1000 puzzles

The results we can see from this that Simulated Annealing has the biggest difference between the lowest and highest times and passes whereas the other two algorithms almost consistently keep around the same time and passes for any puzzle.

To create more detailed results, 100 puzzles were generated, and the same tests done. This offered slightly better understanding, seen in fig 6.2, as we can see Backtracking performing consistency better than the other two, and that Hill Climb is mostly a fixed horizontal line.

Figure 6.2 Result graph for 100 puzzles



To look deeper into these results the statistics that have been calculated come in useful as it allows for an overview off all the data without having to look through all the raw data output by the system.

As seen in fig 6.3, there are a number of statistics given for each algorithm. These are calculated using the raw data but for the output the raw data has been capped at the first 10 results.

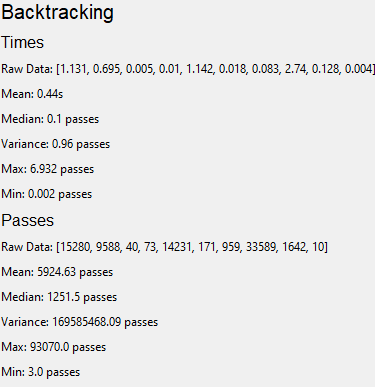


Figure 6.3 Raw output and statistics results for Backtracking algorithm

|  |  |  |  |
| --- | --- | --- | --- |
| Mean | Backtracking | Hill Climb | Simulated Annealing |
| Times (s) | 0.44 | 5.03 | 6.96 |
| Passes | 5,924 | 30,200 | 69,216 |

Table 6.1 Mean results from running algorithms

The mean of all the data shows us that Backtracking, both, has overall the quickest time for solving and the smallest number of iterations through the algorithm to solve the puzzle. This shows considerably better performance than both the other algorithms with a speed of just number half a second.

|  |  |  |  |
| --- | --- | --- | --- |
| Median | Backtracking | Hill Climb | Simulated Annealing |
| Times (s) | 0.1 | 5.03 | 2.28 |
| Passes | 1,251 | 30,200 | 23,460 |

Table 6.2 Median results from running algorithms

The median results show differerent results from the mean data, although it solidifies the discovery that Backtracking is the fastest and most efficient. The difference in results is that Simulated Annealing is faster and more efficient than Hill Climb in terms of medians.

|  |  |  |  |
| --- | --- | --- | --- |
| Max/Min | Backtracking | Hill Climb | Simulated Annealing |
| Times (s) | 6.9/0.002 | 5.3/4.8 | 70.9/0.028 |
| Passes | 93,070/3 | 30,200/30,200 | 700,000/278 |

Table 6.3 Max and Min results from running algorithms

The last important set of results is the minimum and maximum values in the data. It first still shows that Backtracking is the best algorithm but it shows that the minimum time from Simulated Annealing does almost match that of Backtracking athough the maxmimums are very different. It shows also that Hill Climb reaches the maximum number of iterations and times out. This suggests that the algorithm does not successfully solve the puzzles and instead stops running before then. Simulated Annealing also reaches this maximum iterations, however, as shown by the minimum it manages to solve the puzzles the majority of the time.

Overall, the results show that although the smarter algorithms that look for better solutions are more effective at doing their purpose of finding an optimum. In the case of Sudoku solving when there is a single correct solution that these algorithms are looking to find, the brute-force approach is the best in terms of time and iterations. This could be due to the relatively small solution space for a 9x9 puzzle compared to the types of data the more complicated algorithms are designed for.

This could change if the puzzles were made larger to be 4x4 or bigger to a level where the brute-force approach with require a larger number of calculations and iterations to search the now larger solution space.

## Project Requirements

**Requirement 1**: *The tool should provide an implementation of multiple sudoku solving algorithms.*

The final system contains the implementation of three working Sudoku solving algorithms to be compared. Although four algorithms were attempted, the aim was to implement three successfully and therefore even though genetic was attempted and not fully implemented, the original aim of the project was satisfied.

**Requirement 2**: *The test bed should allow puzzle generation and algorithm execution.*

The test bed offers the user a option of 2 functions do execute: calling the puzzle generation algorithm with a specific number and to pick any number of algorithms and run them, using a file of generated puzzles.

**Requirement 3**: *It should provide the ability to generate sudoku puzzles.*

The final system gives the user a button that allows for the puzzle generation algorithm to be called, allowing for puzzles to be created and stored in a file that can later be read by the Test Bed when running algorithms.

Improvements could be made the puzzle generation as it is very inefficient, and the time taken for generating a board varies widely. Generating anything more than 1,000 puzzles takes a considerable amount of time and this subsequently impacts the overall data. With more time, more investigate could be done into other methods of puzzle generation that offer better efficiency.

**Requirement 4**: *The user should be able to choose the number of puzzles to be generated.*

The user can select from either 10,100 or 1000 puzzles to be generated which changes the value that is passed to the puzzle generation algorithm when the puzzle generation button is clicked by the user.

**Requirement 5**: *It should allow for algorithms to be compared.*

For algorithm comparison, all the raw data collected from the running of the algorithms is output in a GUI window to be looked at and the best algorithm found. To support this, graphs plotting the same raw data results are displayed. This creates a more readable way to understand the information.

**Requirement 6**: *The algorithms that are to be compared should be able to be selected by the user.*

The system offers the user checkboxes for each of the algorithms, this allows the selection of any number of algorithms to be compared. With each algorithm run in order from top to bottom of the list depending on how many are selected from 1 to all of them.

**Requirement 7**: *A GUI should be provided that allows Sudoku generation and algorithm comparison.*

The GUI provides the described functionality that both puzzles can be generated, and algorithms can be run. This can be navigated without the need for a tutorial due to it the clarity of the heading and the labels for each input. This tool is also not aimed towards a regular person who has no knowledge of algorithms as it is specialised for a specific purpose of comparing Sudoku solving algorithms. Therefore, it does not need to be accessible to the average user.

The GUI is provided here to allow for an easier interface for using the system not as a way for any user to use the system as that is not its purpose. This means that it is not overly user friendly and only is used to provide the purpose that would otherwise be done using the command line and text inputs.

**Requirement 8**: *Results of comparison using time taken and iterations should be output to the user in a readable manner.*

Each time an algorithm is run, the time taken, and the number of iterations is stored. The first output is the raw data which is not a readable way to view the data. The mean of all the data is given so the user can see in a readable way how the algorithms compare.

The more readable way is through the graph as the user can see, for each puzzle, how the algorithms directly compared to each other in terms of the time taken and the number of iterations. Each algorithm is given a different coloured line which helps to distinguish them from each other and allows the user to clearly see the comparison.

## User Study Plan

A part of the evaluation of the system involves evaluating the user interface portion of the tool. This was important as the plan for the GUI was to not provide a user guide on the features of the interface and therefore a user test had to be done to discover whether this had been done correctly.

A questionnaire containing six questions was created using the University’s form builder [40]. All the questions are asking the user about the ease of use both of the main window as well as the results. The first three questions involve how easy the tool is to use and navigating the main page with the next two questions about the readability of the results produced and the last question an overall review of the system.

The plan for filling out this questionnaire was asking people to use the system on my own laptop in person and after getting them to fill in the questionnaire online. This is due to the system being an application that cannot be easily sent to people to try without meeting in person with a USB stick. Therefore, it was easier for me to facilitate the using of the tool for the testers.

However, due to the current situation and being unable to return to University or meet people to allow them to do the testing, the user studying testing is unable to be carried out.

# Conclusion

In this chapter, a reflection on the overall dissertation will be made. This will include reviewing the objectives outlined in chapter 1 as well as a discussion on the personal development I have made throughout the project and some of the challenges that I faced. Finally, future work will be discussed showing ways the system could be improved upon in the future.

## Overview

This project developed a tool that provided the ability for the execution and comparison of algorithms for solving Sudoku using puzzles generated by the system. A set of aims and objectives were defined to facilitate the development of this system that will help solve the problem initially defined.

Three sections were developed within this system. The first is the algorithm implementations which were based on designs in chapter 3, each one was implemented independently as they needed to be able to be run individually and all the organisation of this was done by the Test Bed. The second part was the Sudoku generation which was kept completely separate from the algorithms to keep modularity. This section generated a number of puzzles based on the number given to the algorithm and output that number of incomplete puzzles to a text file. The first section is the Test Bed which is the way the puzzle generation communicated with the algorithm and how the user communicated with the rest of the system. All inputs were done through here and then other calls were made dependent on the inputs.

These sections were implemented and tested one at a time as the system relies on all these components working and therefore testing incrementally was essential. Testing uncovered a number of bugs through this unit testing which meant when system testing was initiated, the number of lower level bugs was significantly reduced. Due to testing being done solely by the developer, blackbox testing was unable to be fully utilised and not all test cases were covered.

## Aims and Objectives

The overall aim of the project was defined in chapter 2 as ‘To develop a system that allows for the investigation and comparison of main sudoku solving algorithms at a range of difficulty of puzzle’. This aim was then split into a list of objectives that would allow us to satisfy the aim.

**Objective 1:** Explore current methods of sudoku solving and select three

This research was undertaken and the results of this can be found in section 2.3 with consideration taken for the time constraint that is on this project when selecting algorithms. Although four algorithms were selected, this is due to Genetic being a much harder algorithm to implement and therefore a completed implementation was not guaranteed.

**Objective 2:** Explore state of the art sudoku solving tools and evaluate selection

During background research, different existing systems were examined, and the evaluation is given in section 2.4. The evaluation explored five different tools both in papers of related research and other mediums of comparisons, examining the differences and how we can improve on their systems.

I feel that the project did not achieve the personal goal of improving on some of the systems researched. The systems implemented more complex algorithms that were potentially not feasible in the time frame. However, I feel I could have pushed myself harder to understand the more complex algorithms at similar levels to Genetic if I had started the research earlier.

**Objective 3:** Identify the functional requirements of the system

The requirements for the project were the most important part of the design before implementation. With both the design and implementation based on the set of requirements defined in section 3.1. These requirements were then verified in section 6.2 that all of them have been met by the final implementation of the system.

**Objective 4:** Develop test bed to allow comparison of algorithms

The development of the Test Bed can be seen in section 4.3 which details how it was implemented. The Test Bed does allow for comparison of algorithms through the collection of data, time taken and iterations, from each time an algorithm runs and the display of this data in a GUI form both as raw data and visualised as graphs.

Although this is the second objective, the test bed was incrementally developed alongside the other components of the system as more functionality was able to be added. During design and development, we found that the test bed would have more functionality that just the comparison of algorithms as it would act as a base for the other sections of the implementation to return their results to and bring the puzzles and algorithms together.

**Objective 5:** Establish test data for comparison of algorithms

The establishing of test data was important as it not only allows for the algorithm comparison but also the testing of the algorithm depends on solvable sudoku puzzles that are passed to algorithms. This was implemented in two iterations as shown in section 4.3, the first being static data that is found from a third party and stored and the second being generation of puzzles from within the system. Both were implemented with the second iteration providing the ability to have more test data and therefore used for each algorithm being compared.

**Objective 6:** Implement algorithms into the test bed

Algorithm implementation was done incrementally from simplest of most complex algorithm. The design is seen in section 3.7 which defines how each algorithm should be implemented, with implementation in section 4.2.

Since objective 1 explains the fourth algorithm is time dependant, there are only three working implementations of algorithms in the system with the Genetic algorithm not being error free although the logic for the algorithm is there. The original objective for selecting three algorithms and implementing them has been met, however, due to us selecting an additional algorithm, not all the algorithms that should have been implemented correctly were.

**Objective 7:** Evaluate implemented sudoku algorithms at multiple complexities of puzzle

Evaluation of the algorithms was done using two data points, the time taken for the algorithm to solve the puzzle and the number of times the algorithm iterated through itself. This data was collected every time an algorithm ran, and it was stored until the data was being displayed. The evaluation of the data in section 6.1 shows that the displayed raw data and graphs allows for a good comparison to be made and for all the algorithms to be evaluated for efficiency and speed.

For comparisons with multiple complexities of puzzle, the objective has not been met as the puzzle generation algorithm is only sufficient to create a single difficulty of puzzle. We found that even though this was a disadvantage of generating puzzles, it still allows for better test data than if puzzles were fixed every time the algorithms were run.

## Personal Development

Throughout the process of writing this dissertation, many new skills have been learnt:

* **Research skills:** Throughout the research for the project I had to learn how to effectively read and take in academic papers. This was a new experience for me as, although I had read papers, the extent that I had to read and finding useful information was a valuable skill I had to learn.
* **Knowledge of algorithms:** Having come into this project with almost no knowledge of stochastic or machine learning algorithms, what I have learn during both the research and development of the algorithms has grown my knowledge.
* **Python:** Although I had used basic Python while at school, using the language in more of a way that it was intended through data manipulation and algorithms that can overall be applied to data analysis increased my knowledge tremendously. This is something that interests me so having this opportunity to learn it in a productive setting thoroughly helped me.

* **Data Analysis and Visualisation:** The Python libraries for data analysis and visualisation are very powerful and widely used and even though I did not use them fully, being able to understand how they work is a skill I am glad I have gained.
* **Development Cycles:** Working on the development in an iterative method with testing done after each component was completed was something I had yet to experience and considering its importance in commercial software development, the importance of this skill is huge.
* **System Design:** Designing a system from scratch was something I had experience with but not in a big scale project like this, so it reinforced the importance of both low and high level design early in the system implementation.

## Challenges

* **Research:** During the beginning of initial research, starting with a very limited knowledge base and trying to understand these difficulty concepts was hard. Since academical papers expect a certain level of pre-existing knowledge of the topic and finding the motivation to do background research on the algorithms especially was hard to begin with.
* **System Size:** Dealing with the different algorithms as well as the puzzle generation and test bed all as different components was hard to link together. Understanding how the different parts communicated with each other while writing the design section was challenging for me. Also, the amount of code that was in the implementation made readability an issue before I started commenting and leaving appropriate white space.
* **Python Libraries:** Given the data being collected and the analysis being done I had to use a lot of different libraries for data that I had never used before. This was hard as it is a section of Python that I have never explored before and so it was all new to me. Using exclusively NumPy arrays for this project was also interesting and I learnt a lot through the experience of using them, even though it was something new.
* **Genetic Algorithm:** The complexity of the genetic algorithm compared to the other three implemented algorithms posed a real challenge to me as it required me to learn and understand a lot of new concepts. The idea of a population and the basis of the algorithm on genetics was a new area for me and took longer than the time of the development to fully grasp.

## Future Work

The future of this system could go many ways whether its towards usability and teaching or more algorithm and comparison focused. Outlined are a few possibilities:

* Finishing the implementation of Genetic algorithm that was unable to be completed due to the time constraint. Since the logic has been implemented, this would involve testing and fixing bugs in the code to allow the algorithm to work.
* Building on the puzzle generation, the addition of being able to create different difficulties of puzzle would help the tool overall as it would allow more detailed comparisons to be made as it is possible that different algorithms perform differently dependent on puzzle difficulty.
* Adding more solving algorithms to the tools would be possible, allowing for better comparison with more complex algorithms. The tool is easily set up for supporting more algorithms as it is easy to modify code to accommodate them.

Some more challenging features that could be implemented in the future:

* Adding additional support for real-time animation of the algorithms as they are run. This would be a good feature for understanding how the algorithms work and how they reach the solution, however, it would take away from performance and transform the tool into more of a teaching tool.
* Adding a web based client for allowing the system to be used anywhere as long as there is an internet connection. Allowing for a user to not require the system to be downloaded which would save them time.

References

|  |  |
| --- | --- |
| [1] | “Sudoku,” Wikipedia, 2020 March 23. [Online]. Available: https://en.wikipedia.org/wiki/Sudoku. [Accessed 28 March 2020]. |
| [2] | W. Rayment, “The History of Sudoku,” Sudoku.com, 2018. [Online]. Available: https://sudoku.com/how-to-play/the-history-of-sudoku/. [Accessed 14 March 2020]. |
| [3] | “Sudoku Rules for Complete Beginners,” Sudoku, [Online]. Available: https://sudoku.com/how-to-play/sudoku-rules-for-complete-beginners/. [Accessed 28 March 2020]. |
| [4] | “Sudoku Generator Algorithm,” 101 Computing, 21 March 2019. [Online]. Available: https://www.101computing.net/sudoku-generator-algorithm/. [Accessed 28 March 2020]. |
| [5] | Big Fish Games, “How to Solve Sudoku Puzzles Quickly and Reliably,” Big Fish Games, 9 August 2011. [Online]. Available: https://www.bigfishgames.com/blog/how-to-solve-sudoku-puzzles-quickly-and-reliably/. [Accessed 14 March 2020]. |
| [6] | “qqWing,” [Online]. Available: https://qqwing.com/. [Accessed 25 February 2020]. |
| [7] | H.-f. Leung and C.-h. Lam, “Progressive Stochastic Search for Solving Constraint Satisfaction Problems,” in *Tools with Artificial Intelligence*, 2003. |
| [8] | T. Mantere and J. Kolijonen, “Solving, rating and generating Sudoku puzzles with GA,” in *Evolutionary Computation*, 2007. |
| [9] | C. Chang, Z. Fan and Y. Sun, “A Difficulty Metric and Puzzle Generator for Sudoku,” *The UMAP Journal,* vol. 29, no. 3, pp. 305-326, 2008. |
| [10] | J. Meng and X. Lu, “The Design of the Algorithm of Creating Sudoku Puzzle,” in *Advances in Swarm Intelligence*, Chongqing, China, 2011. |
| [11] | M. Hunt, C. Pong and G. Tucker, “Difficulty-Driven Sudoku Puzzle,” *The UMAP Journal,* vol. 29, no. 3, pp. 343 - 362, 2008. |
| [12] | S. Zibbu, “Sudoku and Backtracking,” Hackernoon, 1 June 2018. [Online]. Available: https://hackernoon.com/sudoku-and-backtracking-6613d33229af. [Accessed 14 March 2020]. |
| [13] | “Solving Sudoku boards using stochastic methods and genetic algorithms,” 1 December 2018. [Online]. Available: https://github.com/sraaphorst/sudoku\_stochastic. [Accessed 5 March 2020]. |
| [14] | Upasana, “An Introduction to Hill Climbing Algorithm,” Edureka, 10 December 2019. [Online]. Available: https://www.edureka.co/blog/hill-climbing-algorithm-ai/. [Accessed 1 April 2020]. |
| [15] | “Introduction to Hill Climbing | Artificial Intelligence,” GeeksforGeeks, [Online]. Available: https://www.geeksforgeeks.org/introduction-hill-climbing-artificial-intelligence/. [Accessed 1 April 2020]. |
| [16] | M. Schermerhorn, “A Sudoku Solver,” [Online]. Available: https://www.cs.rochester.edu/u/brown/242/assts/termprojs/Sudoku09.pdf. [Accessed 1 April 2020]. |
| [17] | D. Bertsimas and J. Tsitsiklis, “Simulated Annealing,” *Statistical Science,* vol. 8, no. 1, pp. 10-15, 1993. |
| [18] | R. Lewis, “Metaheuristics can Solve Sudoku Puzzles.,” Edinburgh, 2007. |
| [19] | M. Perez and T. Marwala, “STOCHASTIC OPTIMIZATION APPROACHES FOR SOLVING SUDOKU,” Computing Research Repository, 2008. |
| [20] | D. Whitley, “A genetic algorithm tutorial,” *Stat&tics and Computing ,* vol. 4, pp. 65-85 , 1994. |
| [21] | N. Thirer, “About the FPGA implementation of a genetic algorithm for solving Sudoku puzzles,” in *2012 IEEE 27th Convention of Electrical and Electronics Engineers in Israel*, Eilat, 2012. |
| [22] | S. Houthaak, “Solving a Sudoku with a Genetic Algorithm,” 3 December 2017. [Online]. Available: https://studiohouthaak.nl/solving-a-sudoku-with-a-genetic-algorithm/. [Accessed 9 March 2020]. |
| [23] | Q. D. Xiu and D. L. Yong, “A novel hybrid genetic algorithm for solving Sudoku,” *Optimization Letters,* vol. 7, p. 241–257, 2011. |
| [24] | S. Ekne and K. Gylleus, “Analysis and comparison of solving algorithms for sudoku,” KTH ROYAL INSTITUTE OF TECHNOLOGY, Stockholm, 2015. |
| [25] | M. Thenmozhi, P. Jain, S. Anand R and S. Ram B, “Analysis of Sudoku Solving Algorithms,” *International Journal of Engineering and Technology,* vol. 9, no. 3, pp. 1745 - 1749, 2017. |
| [26] | “Online Sudoku Solver and Helper,” Sudoku Solution, [Online]. Available: https://www.sudoku-solutions.com/. [Accessed 14 March 2020]. |
| [27] | “Python,” Python, [Online]. Available: https://www.python.org/. [Accessed 14 March 2020]. |
| [28] | “Visual Studio Code,” Microsoft, [Online]. Available: https://code.visualstudio.com/. [Accessed 14 March 2020]. |
| [29] | “Github,” Github, [Online]. Available: https://github.com/. [Accessed 14 March 2020]. |
| [30] | “Python,” Microsoft, 2020 February 21. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=ms-python.python. [Accessed 14 March 2020]. |
| [31] | “NumPy,” NumPy Developers, 2020. [Online]. Available: https://numpy.org/. [Accessed 14 March 2020]. |
| [32] | “NumPy,” Wikipedia, 2020 January 31. [Online]. Available: https://en.wikipedia.org/wiki/NumPy. [Accessed 14 March 2020]. |
| [33] | “Matplotlib,” The Matplotlib development team, 4 March 2020. [Online]. Available: https://matplotlib.org/. [Accessed 14 March 2020]. |
| [34] | “Tkinter,” Python, 06 December 2019. [Online]. Available: https://wiki.python.org/moin/TkInter. [Accessed 2020 March 14 ]. |
| [35] | “The Software Design Methodology,” [Online]. Available: https://userpages.umbc.edu/~khoo/survey1.html. [Accessed 2 April 2020]. |
| [36] | [Online]. Available: https://app.diagrams.net/. [Accessed 28 April 2020]. |
| [37] | “Sudoku,” [Online]. Available: https://sudoku.com/. [Accessed 12 April 2020]. |
| [38] | [Online]. Available: https://en.wikipedia.org/wiki/Mathematics\_of\_Sudoku. [Accessed 28 April 2020]. |
| [39] | perrygeo, “Python module for Simulated Annealing optimization,” GitHub, [Online]. Available: https://github.com/perrygeo/simanneal. [Accessed 29 April 2020]. |
| [40] | D. Marshall, “Steepest Ascent Hill Climbing,” [Online]. Available: http://users.cs.cf.ac.uk/Dave.Marshall/AI1/steep.html. [Accessed 1 April 2020]. |
| [41] | M. Azmi Al-Betar, M. A. Awadallah, A. L. Bolaji and B. O. Alijla, “ß-Hill Climbing algorithm for sudoku game,” in *2017 Palestinian International Conference on Information and Communication Technology (PICICT)*, Palestinian , 2017. |
| [42] | [Online]. Available: https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.andrew.cmu.edu%2Fuser%2Fkkuan%2FfinalWriteup.html&psig=AOvVaw0pBm2sArXCLkBrq7I5jdia&ust=1583347736731000&source=images&cd=vfe&ved=0CAIQjRxqFwoTCLDY-Yr9\_ucCFQAAAAAdAAAAABAD. [Accessed 23 April 2020]. |
| [43] | [Online]. Available: https://images.mendix.com/wp-content/uploads/Artboard-1@2x-701x700.png. [Accessed 23 April 2020]. |

Table of Figures

[Figure 1.1 Completed Sudoku puzzle [42] 7](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418746)

[Figure 3.1 Software development cycle [43] 18](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418747)

[Figure 3.2 High Level Design of System 19](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418748)

[Figure 3.3 GUI design for System 20](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418749)

[Figure 3.4 Mock of output GUI 21](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418750)

[Figure 3.5 Design of interaction between test bed and puzzle generation 22](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418751)

[Figure 3.6 Flow chart of Backtracking 23](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418752)

[Figure 3.7 Flow chart of Hill Climb 25](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418753)

[Figure 3.8 Flow chart of Simulated Annealing 27](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418754)

[Figure 3.9 Flow chart of Genetic Algorithm 28](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418755)

[Figure 4.1 Code snippet for filling Sudoku board 31](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418756)

[Figure 4.2 Code snippet for calls for solving puzzle in puzzle generation 32](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418757)

[Figure 4.3 Code snippet for order of numbers used in puzzle generation 32](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418758)

[Figure 4.4 Code snippet for comparing results of ascending and descending solves 32](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418759)

[Figure 4.5 Code snippet for getting sub-box of cell 33](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418760)

[Figure 4.6 Code snippet for iterating through sub-box 33](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418761)

[Figure 4.7 Code snippet of Backtracking recursion base case 34](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418762)

[Figure 4.8 Code snippet for backtracking fixed loop 34](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418763)

[Figure 4.9 Code snippet for puzzle initialisation 35](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418764)

[Figure 4.10 Code snippet of swapping of cells in Hill Climb 35](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418765)

[Figure 4.11 Code snippet for moving to new states in Hill Climb 36](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418766)

[Figure 4.12 Code snippet of timeout condition in Hill Climb 36](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418767)

[Figure 4.13 code snippet of initialisation of SudokuSolver class for Simulated Annealing 37](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418768)

[Figure 4.14 Code snippet of Simulated Annealing parameters 38](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418769)

[Figure 4.15 Code Snippet of Simulated Annealing parameters 38](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418770)

[Figure 4.16 Code snippet of compete function in Genetic Algorithm 39](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418771)

[Figure 4.17 Code snippet of crossover in Genetic Algorithm 40](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418772)

[Figure 4.18 Code snippet of board formatting in Test Bed 41](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418773)

[Figure 4.19 GUI for algorithm selection and puzzle generation 42](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418774)

[Figure 4.20 GUI for result display 43](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418775)

[Figure 4.21 GUI graph of results 43](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418776)

[Figure 5.4 Output of algorithm unit tests 44](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418777)

[Figure 5.5 Output of Sudoku Generation unit tests 45](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418778)

[Figure 5.6 Output of Test Bed unit tests 46](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418779)

[Figure 6.1 Result graph for 1000 puzzles 49](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418780)

[Figure 6.2 Result graph for 100 puzzles 50](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418781)

[Figure 6.3 Raw output and statistics results for Backtracking algorithm 51](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc39418782)

Table of Tables

[Table 5.1 Some tests for algorithm execution through GUI 46](#_Toc39430076)

[Table 6.1 Mean results from running algorithms 50](#_Toc39430077)

[Table 6.2 Median results from running algorithms 51](#_Toc39430078)

[Table 6.3 Max and Min results from running algorithms 51](#_Toc39430079)

Appendices

Appendix A: Testing Done

A1: Algorithm Unit Tests

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Number** | **Input** | **Expected Output** | **Actual Output** | **Pass/Fail** |
| 1 | Almost complete puzzle with 1 “0” located at position 4,4 | Find Unassigned Location returns the position 4,4 as it contains a “0” | Find Unassigned Location returns the position 4,4 as it contains a “0” | Pass |
| 2 | Pass incomplete puzzle with a “3” in row 6 to used in row function with row 6 and number 3 | Asserts true that the number 3 is already used in row 6 | Asserts true that the number 3 is already used in row 6 | Pass |
| 3 | Pass incomplete puzzle with a “1” in column 2 to used in column function with column 2 and number 1 | Asserts true that the number 1 is already used in column 2 | Asserts true that the number 1 is already used in column 2 | Pass |
| 4 | Pass incomplete puzzle with a “7” in the box starting with the coordinates 3,3 to used in box function with coordinates 3,3 and number 7 | Asserts true that the number 7 is already used in box with starting coordinates 3,3 | Asserts true that the number 7 is already used in box with starting coordinates 3,3 | Pass |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | Call get box using the number 3 with a list of coordinates from that box | The coordinates returned by the get box function will be equal to the expected data | The coordinates returned by the get box function will be equal to the expected data | Pass |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 6 | Initialise the incomplete puzzle using the function and get the values in a single box | The numbers in the box should be the numbers 1-9 only occurring once each | The numbers in the box should be the numbers 1-9 only occurring once each | Pass |
| 7 | Pass incomplete board to Backtracking | Puzzle returned is equal to the completed puzzle solved by a third party | Puzzle returned is equal to the completed puzzle solved by a third party | Pass |
| 8 | Pass incomplete board to Hill Climb | Puzzle returned is equal to the completed puzzle solved by a third party | Failure as puzzle returned does not equal expected puzzle | Fail |
| 9 | Pass incomplete board to Simulated Annealing | Puzzle returned is equal to the completed puzzle solved by a third party | Puzzle returned is equal to the completed puzzle solved by a third party | Pass |
| 10 | Pass incomplete board to Genetic | Puzzle returned is equal to the completed puzzle solved by a third party | Algorithm crashes before returning a solved puzzle | Fail |

A2: Generation and Test Bed Unit Tests

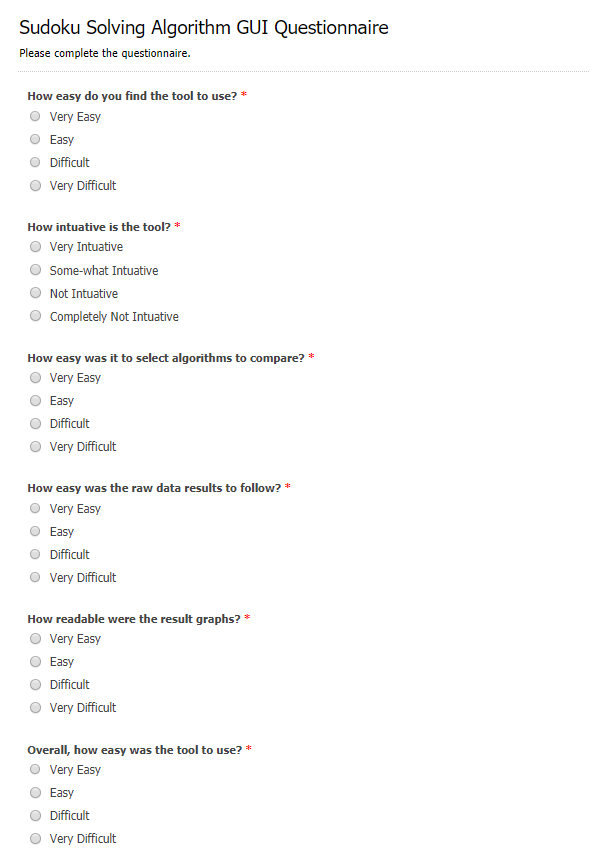
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Number** | **Input** | **Expected Output** | **Actual Output** | **Pass/Fail** |
| 11 | Get board from file with line number 3 from function | Return from function equals the test opening the file and getting the data itself | Return from function equals the test opening the file and getting the data itself | Pass |
| 12 | Pass an empty board to the fill sudoku function | Returned board should have no “0” numbers within it. | Returned board should have no “0” numbers within it. | Pass |
| 13 | Pass a puzzle with non-empty cells to find nonempty cells function | Return from function should not equal “0” | Return from function should not equal “0” | Pass |
| 14 | Pass completed puzzle to main which removes numbers to make it solvable | Comparing puzzle returning to incomplete puzzle should have the same number of “0” cells | Comparing puzzle returning to incomplete puzzle should have the same number of “0” cells | Pass |

A3: GUI Puzzle Generation Tests

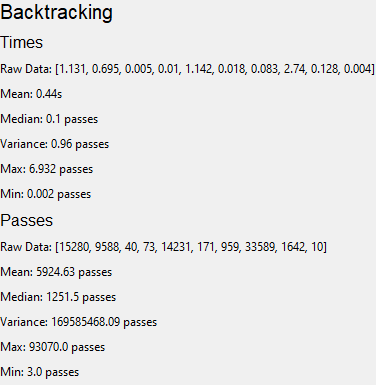
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Number** | **Input** | **Expected Output** | **Actual Output** | **Pass/Fail** |
| 15 | Click “10” puzzle radio button | “10” puzzle button is selected | “10” puzzle button is selected | Pass |
| 16 | Click “100” puzzle radio button | “100” puzzle button is selected | “100” puzzle button is selected | Pass |
| 17 | Click “1000” puzzle radio button | “1000” puzzle button is selected | “1000” puzzle button is selected | Pass |
| 18 | Click “10” puzzle radio button and “generate boards” clicked | Outputs numbers 1-10 in console and 10 lines are filled in puzzles.txt | Outputs numbers 1-10 in console and 10 lines are filled in puzzles.txt | Pass |
| 19 | Click “100” puzzle radio button and “generate boards” clicked | Outputs numbers 1-100 in console and 100 lines are filled in puzzles.txt | Outputs numbers 1-100 in console and 100 lines are filled in puzzles.txt | Pass |
| 20 | Click “1000” puzzle radio button and “generate boards” clicked | Outputs numbers 1-1000 in console and 1000 lines are filled in puzzles.txt | Outputs numbers 1-1000 in console and 1000 lines are filled in puzzles.txt | Pass |

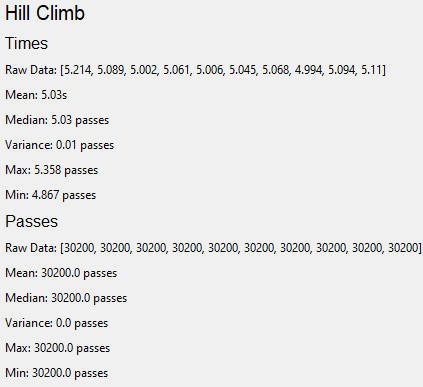
A4: GUI Algorithm Run Tests

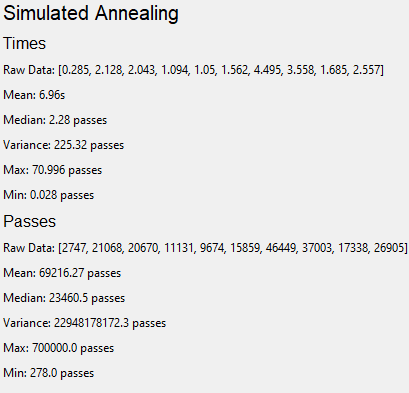
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Number** | **Input** | **Expected Output** | **Actual Output** | **Pass/Fail** |
| 21 | Click “Backtracking” check box | “Backtracking” check box selected | “Backtracking” check box selected | Pass |
| 22 | Click “Hill Climb” check box | “Hill Climb” check box selected | “Hill Climb” check box selected | Pass |
| 23 | Click “Simulated Annealing” check box | “Simulated Annealing” check box selected | “Simulated Annealing” check box selected | Pass |
| 24 | Click “Genetic” check box | “Genetic” check box selected | “Genetic” check box selected | Pass |
| 25 | Click “Run Algorithms” with no check boxes selected | Error message saying no algorithms were selected | Error message saying no algorithms were selected | Pass |
| 26 | “Backtracking” selected with 10 puzzles and “Run Algorithm” clicked | Numbers 0-9 output to console, 2 windows open for data and graph | Numbers 0-9 output to console, 2 windows open for data and graph | Pass |
| 27 | “Hill Climb” selected with 10 puzzles and “Run Algorithm” clicked | Numbers 0-30000 output to console 10 times, 2 windows open for data and graph | Numbers 0-30000 output to console 10 times, 2 windows open for data and graph | Pass |
| 28 | “Simulated Annealing” selected with 10 puzzles and “Run Algorithm” clicked | Simulated Annealing display output to console, 2 windows open for data and graph | Simulated Annealing display output to console, 2 windows open for data and graph | Pass |
| 29 | “Genetic” selected with 10 puzzles and “Run Algorithm” clicked | Numbers 0-9 output to console, 2 windows open for data and graph | Algorithm crashes and returns error message to console | Fail |

Appendix B: Questionnaire for User Study

Appendix C: Full Results

C1: Backtracking

C2: Hill Climb

C3: Simulated Annealing