Comparison of Sudoku Solving Algorithms

Calum Harvey (Student ID: 170349985)

Computer Science

Supervisor: Dr Jason Steggles

Word Count: 15,000

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

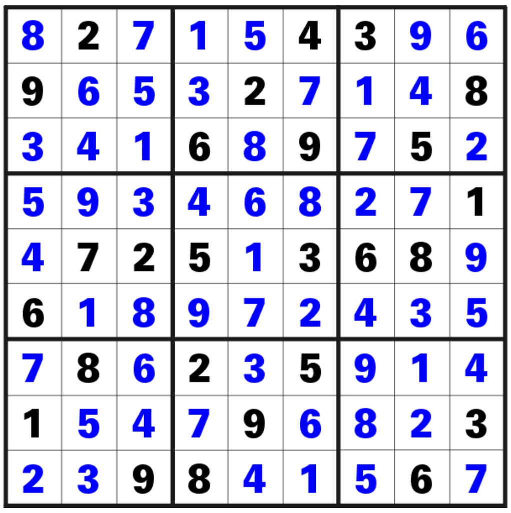
I would like to thank Dr Jason Steggles for supporting me throughout the dissertation and helping me to do this project in an area that interests me. For providing structure to the project to allow for it to be completed on time.

Table of Contents

1 Introduction

1.1 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 nine 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 three 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].



8	2	7	1	5	4	3	9	6
9	6	5	3	2	7	1	4	8
3	4	1	6	8	9	7	5	2
5	9	3	4	6	8	2	7	1
4	7	2	5	1	3	6	8	9
6	1	8	9	7	2	4	3	5
7	8	6	2	3	5	9	1	4
1	5	4	7	9	6	8	2	3
2	3	9	8	4	1	5	6	7

Figure 1.1 Completed Sudoku puzzle [45]

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. Another approach is using brute-force backtracking, but the efficiency of the algorithm decreases as the number of empty cells increases as the Sudoku problem gets harder.

A possible 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.

1.2 Motivation

The problem with0. 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 arrive at 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 approaches can then be compared in terms of their speed and number of iterations taken to reach the optimum solution. Further work can also be done to be compare a number of different complexities of puzzles across a wide set of problems to ensure the comparison is as accurate as possible.

1.3 Aim

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

1.4 Objectives

1. Explore current methods of Sudoku solving and select three
2. Explore state of the art Sudoku solving tools and evaluate selection
3. Identify the functional requirements of the system
4. Develop test bed to allow comparison of algorithms
5. Establish test data for comparison of algorithms
6. Implement algorithms into the test bed
7. Evaluate implemented Sudoku algorithms at multiple complexities of puzzle

2 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 us to justify the design decisions that are made later in the development.

2.1 Sudoku Puzzles

When looking at the way an algorithm solves a puzzle, 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 puzzles 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 three criteria that a valid solution must meet as [5]:

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

2.2 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 maximise effectiveness 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.

2.2.1 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 tested against a limited number of puzzles as a vast number of these puzzles are not required [7].

Choosing pre-generated puzzles ensures 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. It is possible to use pre-generated puzzles, but this would involve loading these puzzles from a source website which could become as complex as generating puzzles.

2.2.2 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 created which allows you to test algorithms against a very specific difficulty and 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 allowing a larger controlled sample to be generated than manually entered 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 to easily be passed to algorithms.

This is why research in the field of Sudoku solving incorporate the creation of the puzzles as it allows for vastly improved results. This is shown [8] where the solving of Sudoku puzzles using genetic algorithms are examined incorporating a test data algorithm.

Research into existing computer-based puzzle generators shows that there are two 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 complexity

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 complexity

Both these methods are similar in the way they created the final puzzle - they both require a solving algorithm to determine if there is unique solution for the current puzzle. There is a difference in what the solving algorithm needs to accomplish during 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 creates a solution as solving an empty puzzle, if implemented correctly, will give a valid complete puzzle.

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 since it tries every possibility. There is also research into using Genetic algorithms for Sudoku generation [8], due to the randomness of stochastic algorithms allows for multiple solutions in the bottom-up version to be found fast. It also enables the generation in top-down to be done with potentially more randomness than if a brute-force variation was used.

The desired complexity of the Sudoku when being algorithmically generated is dependent on the number of filled cells. 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 as [11] this is purely based on techniques a human uses to solve puzzles. Puzzles that contains less filled cells can need the use of more complex strategies to solve the puzzle and although there are a number of exceptions to this rule, the majority of puzzles follow the correlation between number of cells filled and difficulty to solve.

2.3 Algorithms

The algorithms most commonly examined 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 interpretation of the stochastic approach.

Selecting the right algorithms is important as there needs to be a variation to allow for a good comparison and for our objectives to more easily be satisfied. Backtracking is the brute-force algorithm and therefore is a good base benchmark. The stochastic algorithms are used to compare against each other to try and beat backtracking in time taken and efficiency.

2.3.1 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 the correct answer to the puzzle [12].

The algorithm iterates through numbers 1-9 in each cell, checking if placing each number in turn will break one of the three constraints of Sudoku. When a valid number is found, it is added to the cell and will move 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 [13] against other, more complex algorithms as it gives a very good base as the most straight forward type of algorithm.

2.3.2 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 exit from local minima.

There are three steps in generic Hill Climb:

- Generate a possible solution
- Evaluate the possible solution against expected solution
- Move to possible solution only if it better than previous solution

This can be adapted for Sudoku solving with minimal limitation.

Solving Sudoku requires an optimization 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 three 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 three Sudoku constraints – each box, column and row must have the numbers 1-9. In this paper rows are being used. 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 always successfully solve a Sudoku. 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 that will, after the algorithm executed for a number of cycles without improving, will restart and re-create the start state.

2.3.3 Simulated Annealing

Simulated Annealing (SA) is an optimization technique based on annealing in metallurgy which involves heating and cooling metals and is used for finding optimum solutions [17]. For each move, a neighbouring state is found by making a small random change to 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 improvement on the current state, then the algorithm changes to the new state. If the new state is worse, then the state only changes given an acceptance probability condition is met, otherwise it 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 more worse moves to be made but with more iterations the temperature decreases, meaning there is a lower chance for a bad move to be made.

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 Sudoku constraints 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 two 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 enumerating the number of values missing. The total cost of the state is the sum of all row and column values. An optimisation is only recalculating at most two rows and two columns after each new move as only the swapped values cause total cost changes.

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.

2.3.4 Genetic Algorithm

Genetic algorithms (GA) [20] are a family of optimisations inspired by survival of the fittest and evolution. It involves creating a population and finding the fittest members of the population. These are then taken and used to create a new population that is closer to the potential solution.

There are three principle stages of a GA:

- Population Initialisation: create a population containing a selection of random individuals.
- Fitness Calculation: test each individual in the population against a fitness function to determine which are closer to a potential solution
- Selection: choose individuals from the population based on their fitness. Then, two individuals are combined to form new individuals which is called crossover, and a individuals is changed on its own. That is called mutation.

Adaption of the algorithm for Sudoku solving requires some changes. There are different ways that the initialisation can be achieved, similar to Simulated Annealing [21] it can be implemented by imposing a constraint 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 three constraints of a Sudoku.

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

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

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 must be upheld during the crossover and mutation stages. 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.

2.4 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.

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

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 Backtracking or constraint programming.

Here [25] there is a comparison made between Brute-force, Backtracking and Dancing Links which are all derivates of brute-force, making it a good comparison. This, however, does not incorporate smarter algorithms into the comparison, focusing on the basic brute-force and its optimisations. This misses out on some important comparisons with more intricate algorithms.

This website [26] gives an empty 9x9 grid that can be filled with a puzzle problem. It will then 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 to solve the puzzle.

2.5 Implementation Technologies

2.5.1 Python

Python [27] is an interpreted, object-oriented, high-level programming language that was conceived in the late 1980s. It is used by many large organisations such as Wikipedia and Google for web applications. However, it has a major use in data analysis and machine learning, offering many important libraries for implementation.

The comparison between algorithms can be simplified by using the already existing libraries and it allows for easy visualisation of the results which can be very useful for algorithm comparison. Good libraries also exist for both Genetic and Simulated Annealing that allow the implementation of the algorithms to be simplified.

2.5.2 Visual Studio Code

Visual Studio Code [28] is an Integrated development environment (IDE) used in software development for a range of programming languages. It integrates with Github [29] to allow for pushes to be made within the application itself. An advantage is an extension for Python [30] that allows for easier debugging of Python code, which helps with development of software in Python. The tool is well supported and is useful when trying to learn a new language

2.5.3 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 arrays [32].

This is useful as Sudoku puzzles are typically represented in data arrays as inputs to algorithms. Using NumPy simplifies the way that the 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 data output from these algorithms is displayed as a set of numbers. Using Matplotlib, this data from different algorithms can be plotted and displayed, making the comparison of result data quick and simple.

2.5.4 Tkinter

Tkinter [34] is the most common Python standard Graphical User Interface (GUI) package. 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 this type of implementation as it allows the user to interact with the application more easily and output data being displayed in a graphical format.

3 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.

3.1 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.

3.2 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.

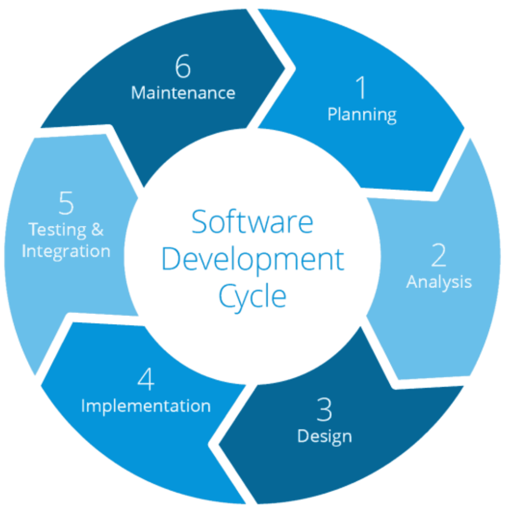


Figure 3.1 Software development cycle [46]

Agile, 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 cycle since regular meetings are held with my supervisor who acted as a client and provides feedback. This allows for changes and updates to be made on the project progression.

This approach aligns well with the incremental nature of the project, allowing each component to be developed independently as the puzzle generation, and each algorithm is implemented separately. A framework was created which allowed the creation of a Minimal Value Product with puzzle generation. This could then be tested and added to with each iteration.

3.3 High Level Design

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

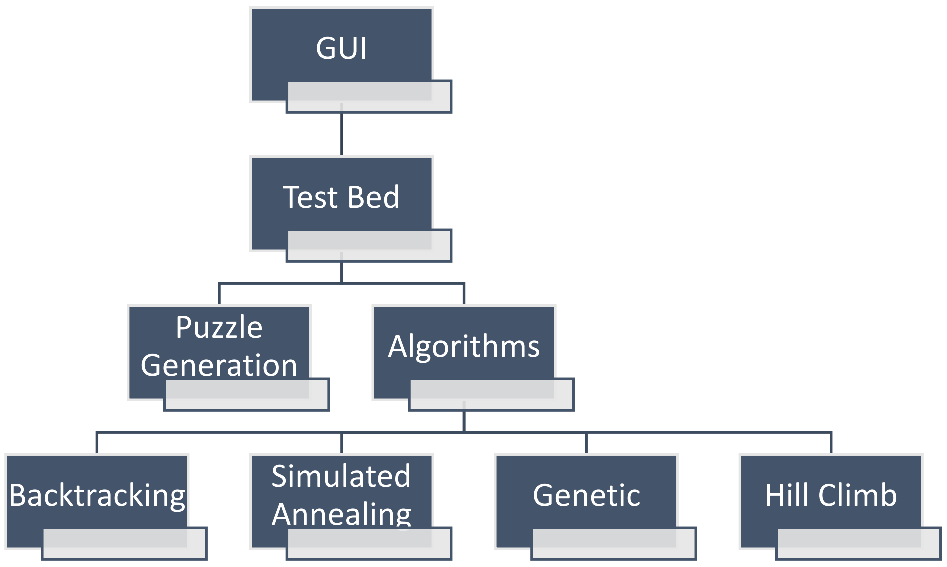


Figure 3.2 High Level Design of System

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 base of the system linking the lower level puzzle generation and algorithm implementations to the GUI

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

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

3.4 Test Bed Design

3.4.1 Design of Graphical User Interface

The user interface design is a vital part of the overall 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 two main aspects, with one side allowing the user to pick the algorithms for comparison and the other side letting the user pick the number of Sudoku puzzles to be generated.

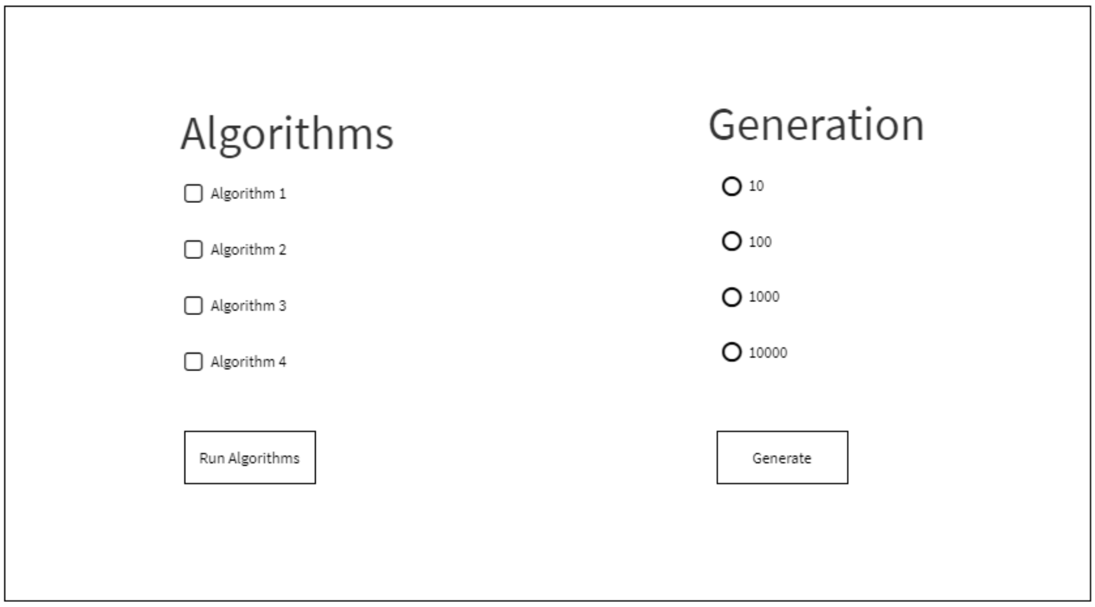


Figure 3.3 GUI design for System

The algorithm selection on the left uses check buttons allowing the user to select from one to all of the algorithm depending on the comparison that is wanting to 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 the number of puzzles to generate, then when generate is clicked the program will generate the selected number of puzzles and indicate completion to the user.

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 instantly.

When the algorithms are run, the user interface will output the results in a graphical form in a separate window for the user to analyse.

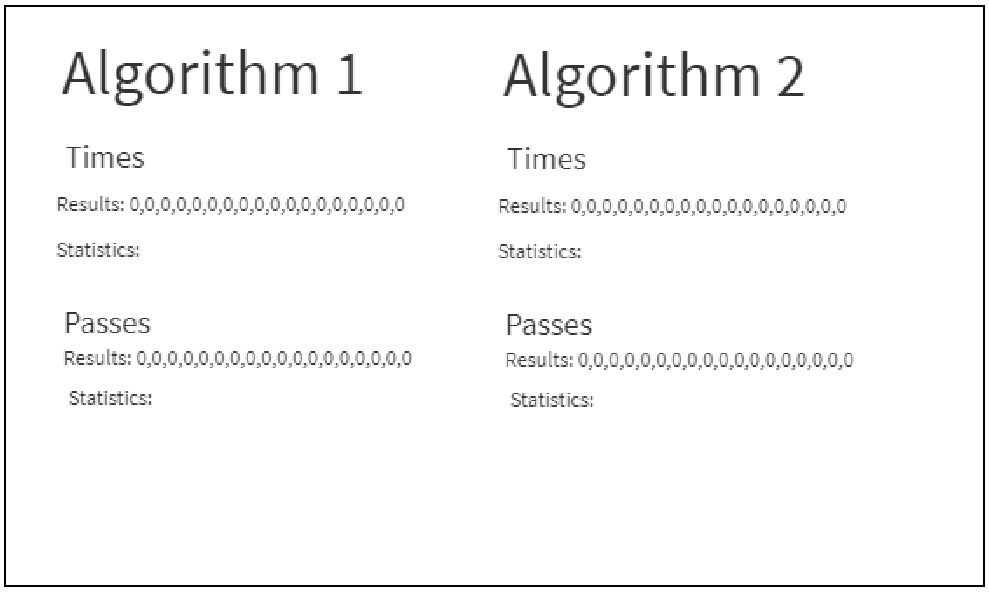

Algorithm 1
Times
Results: 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
Statistics:
Passes
Results: 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
Statistics:
Algorithm 2
Times
Results: 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
Statistics:
Passes
Results: 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
Statistics:

Figure 3.4 Mock of output GUI

Results for each algorithm run is shown, fig 3.4, in the results section with the two metrics we are using to measure the algorithms. This is then followed by some statistics which could consist of the mean or the highest and lowest resulting values which will give the user a simpler indicator of the efficiency of each algorithm.

3.4.2 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 displayed on 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 implemented in different ways. The way that has been decided in this 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.

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 is done in the GUI since the results are displayed graphically and therefore the analysis might be done within the graphical section rather than the algorithm processes.

3.5 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 are output from the algorithms. The main challenges in this section is how the puzzles are stored for the test bed to retrieve them and how interfacing with the puzzle generation overall is achieved.

To solve the first challenge, we first need to look at 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 excessive 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 for the comparisons.

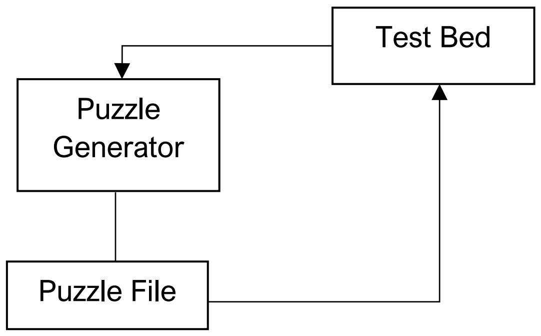


Figure 3.5 Design of interaction between test bed and puzzle generation

Fig 3.5 shows how this would work at a high level, with the 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.

3.6 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 returning to the test bed.

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

3.6.1 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 it reaches the last number, it backtracks to the previous cell and keeps trying new numbers until the criteria is met.

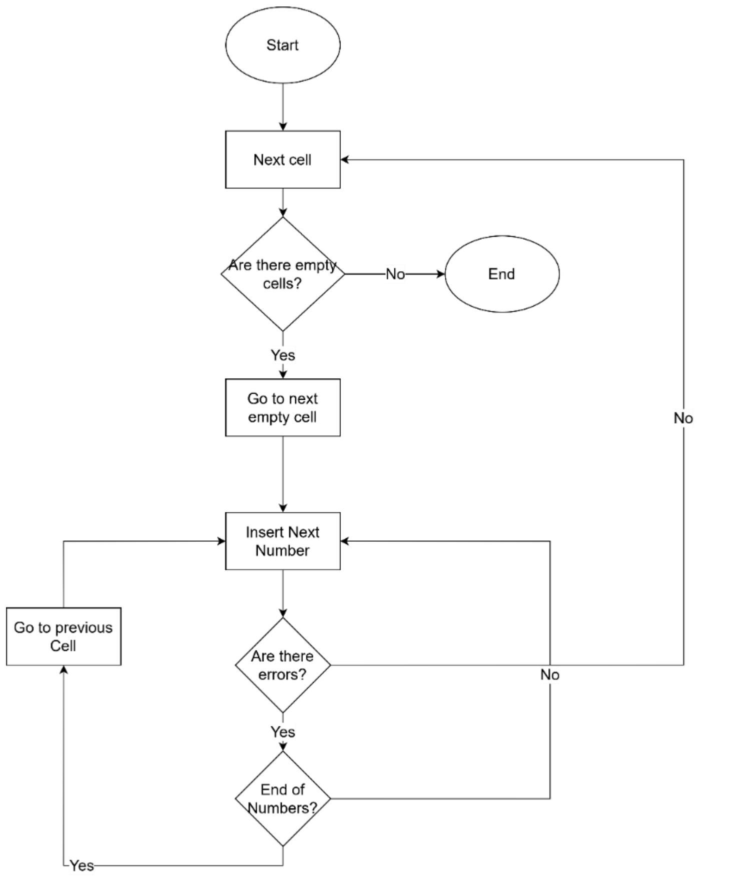


Figure 3.6 Flow chart of Backtracking

The design challenges for this algorithm are how the backtracking will function, 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 functionality being the hardest concept. The algorithm needs to be able to hold the current state of all the previous cells to allow the algorithm to backtrack to earlier cell states and continue the algorithm execution.

The easiest method of achieving this is to use recursion, with a separate call for each cell, as it allows for the state of a cell to be put on the stack as they are processed. This can then be popped off the stack if the algorithm backtracks to that cell, allowing the number checking to continue from where it left off. This allows progression through the algorithm as cells are popped and pushed to the stack.

The exit condition of the recursion happens when no empty cells remain. Once the algorithm iterates through all the cells and tries the cell after the last one, it will meet the exit condition and can be flagged as solved to all states on the stack.

3.6.2 Hill Climb

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

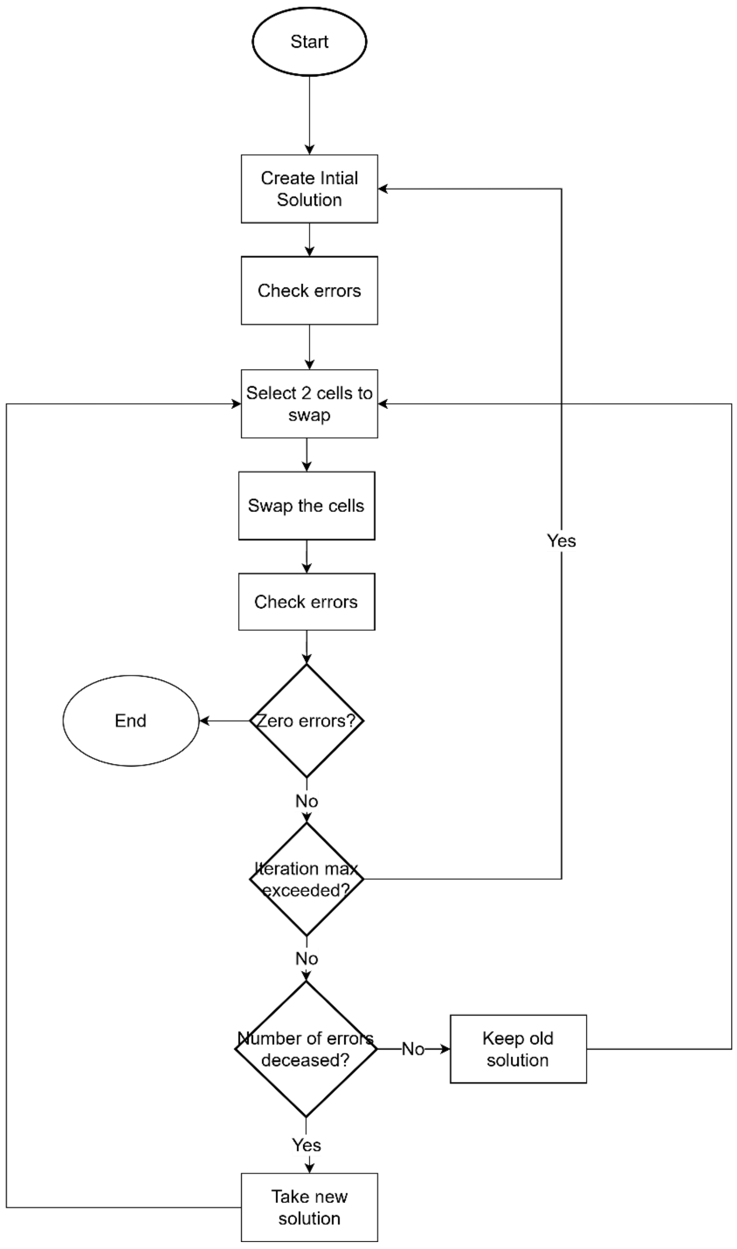


Figure 3.7 Flow chart of Hill Climb

Initial solution creation in this algorithm is important as error calculation is dependent on it. When the puzzle 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 overall, more efficient.

Error checking needs to be designed well as it will be used throughout multiple stochastic solving algorithms. The easiest method 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.

A max iterations is defined to allow the algorithm to force exit from local minima. Since Sudoku has only one solution and these algorithms are used to find an optimum solution, a problem can occur. If the algorithm progresses down a dead end path, then it will not reach a solution and the algorithm process will need to be reset to allow the algorithm to try again.

3.6.3 Simulated Annealing

Simulated Annealing is an algorithm that is an optimization of Hill Climb as it moves to a better solution based on a probability which comes from a temperature.

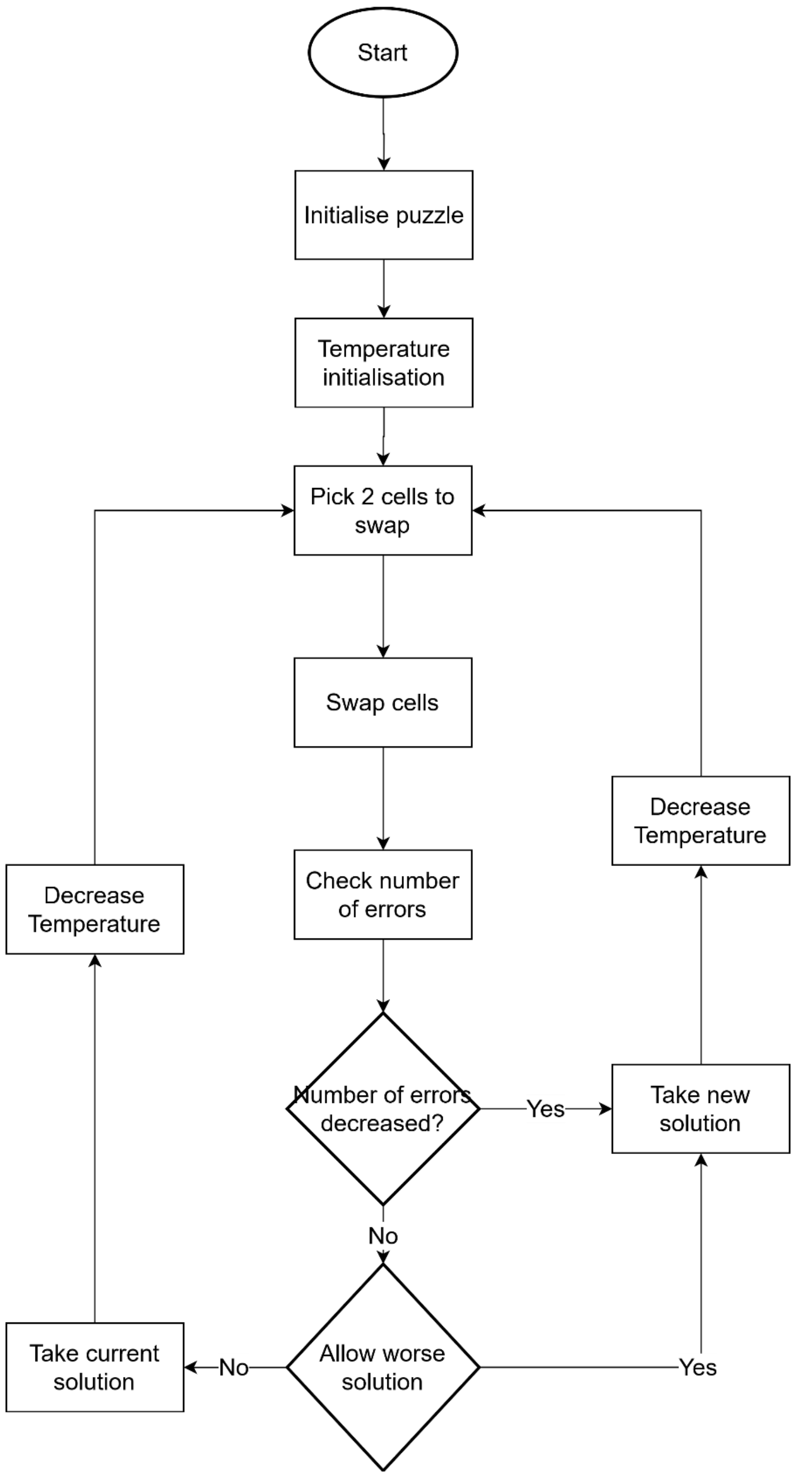


Figure 3.8 Flow chart of Simulated Annealing

Simulated Annealing is an optimization of Hill Climb and offers similar functionality with an improvement on exiting of local minima. The improvement involves using a temperature to decide whether the new solution is taken into the next iteration.

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

Since the puzzle is only initialised once, the temperature initialisation values need to be correct to allow for sufficient exploration of the solution space but also enough to converge on the correct solution to the puzzle and this will require trial and error.

3.6.4 Genetic Algorithm

Genetic Algorithms generate a population of many individual potential solutions, the fitness of each one is calculated, and the best individuals are taken and used to create a new population. This iterates until a solution is found.

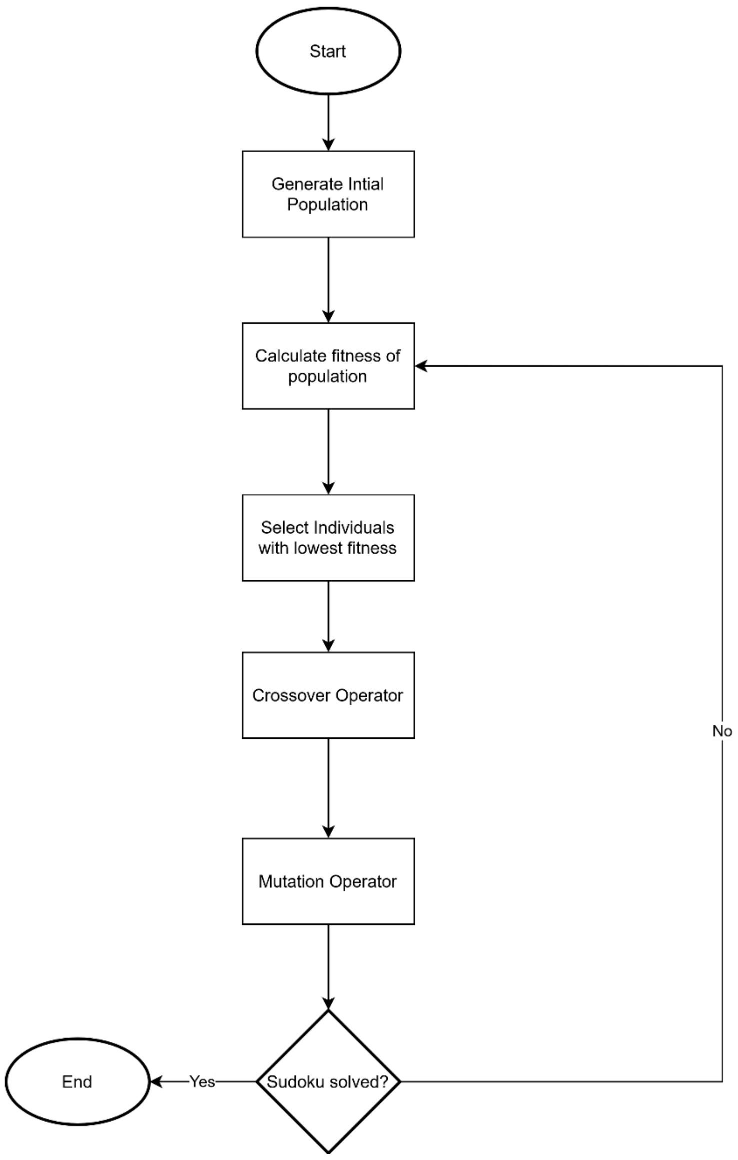

Start
Generate Intial Population
Calculate fitness of population
Select Individuals with lowest fitness
Crossover Operator
Mutation Operator
Sudoku solved?
Yes
End
No

Figure 3.9 Flow chart of Genetic Algorithm

Genetic algorithms contain many different challenging aspects 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 individual allows for the storing of the fitness 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 individuals to move closer to a complete solution. Crossover takes two 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 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 the solutions have been initialised to ensure that the constraints that has been met during crossover 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 implement this in a similar way to Hill Climb and Simulated Annealing with an update made to the constraint being used.

The last challenge will be in error checking. Different constraints are used in the error checking which makes it challenging to accomplish and will require a different method to the previous algorithms. The design of this will be similar to the others but with needing to check sub-boxes which will be new.

4 Implementation

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

4.1 Sudoku Generation

The main idea for this is to provide a way for test data to be generated to allow the algorithms to be tested using the generated puzzle data. There are two methods used in generation of test data – version 1 and 2. The reason for the two versions is generation is an important part in the testing of the algorithms and getting a simpler 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.

4.1.1 Version 1

Mainly due to the iterative design process, this was a simple initial method of generating test puzzles. 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 returns a line from the text file and makes it available to be tested on an algorithm.

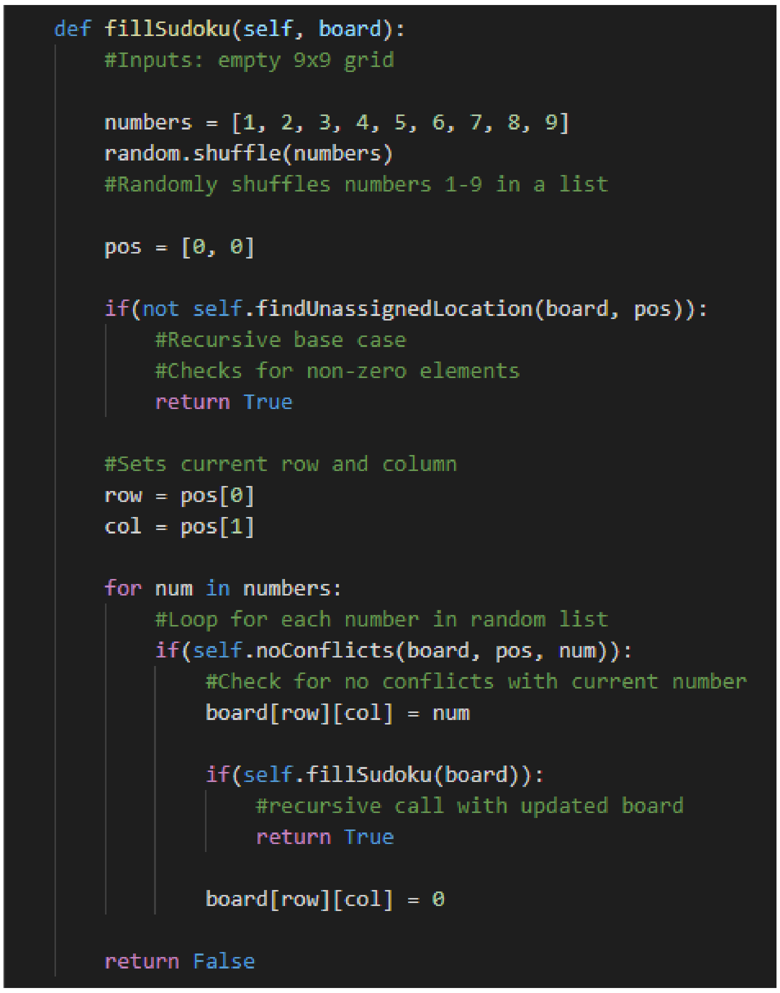
This version was found to create problems within the system due to the time overhead of puzzles being manually added. The amount of time this would take would limit the range of test data that allows the results from the algorithms to be sufficiently tested.

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

4.1.2 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 from when algorithms need to be invoked. This is implemented by a single class with a main function to loop the generation for the number of puzzles requested by the Test Bed.

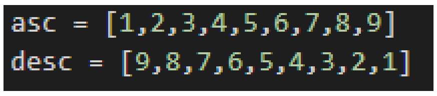
The initial puzzle populating algorithm is seen in fig 4.1. It uses the Backtracking algorithm which was an already implemented algorithms due to its ease in creating randomness. Each attempt at filling a new cell gets a new random order of the numbers 1-9, ensuring additional randomness for the solution. The output is a completely random valid puzzle from a superset of over six sextillion [38] possible combinations



```
def fillSudoku(self, board):
    #Inputs: empty 9x9 grid

    numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
    random.shuffle(numbers)
    #Randomly shuffles numbers 1-9 in a list

    pos = [0, 0]

    if(not self.findUnassignedLocation(board, pos)):
        #Recursive base case
        #Checks for non-zero elements
        return True

    #Sets current row and column
    row = pos[0]
    col = pos[1]

    for num in numbers:
        #Loop for each number in random list
        if(self.noConflicts(board, pos, num)):
            #Check for no conflicts with current number
            board[row][col] = num

            if(self.fillSudoku(board)):
                #recursive call with updated board
                return True

            board[row][col] = 0

    return False
```

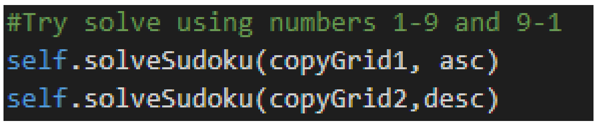
Figure 4.1 Code snippet for filling Sudoku board

The next stage involves cells being selected from the puzzle and removed. The puzzle is checked for having multiple solutions, as a valid Sudoku puzzle must only have one answer. The backtracking algorithm is used since a brute-force algorithm needs to be used here to uniformly search the solution space.



```
asc = [1,2,3,4,5,6,7,8,9]
desc = [9,8,7,6,5,4,3,2,1]
```

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

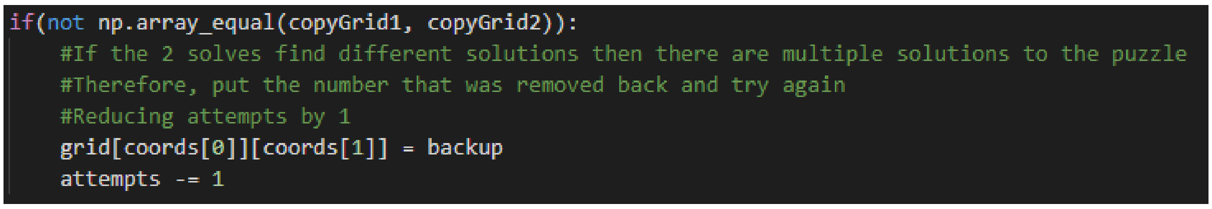


```
#Try solve using numbers 1-9 and 9-1
self.solveSudoku(copyGrid1, asc)
self.solveSudoku(copyGrid2,desc)
```

Figure 4.2 Code snippet for calls for solving puzzle 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.



```
if(not np.array_equal(copyGrid1, copyGrid2)):
    #If the 2 solves find different solutions then there are multiple solutions to the puzzle
    #Therefore, put the number that was removed back and try again
    #Reducing attempts by 1
    grid[coords[0]][coords[1]] = backup
    attempts -= 1
```

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

The algorithm uniformly solves the puzzle using ascending and descending numbers and the results are compared as seen in fig 4.4. If the results from both solves are the same, then both ascending and descending have reached each other somewhere in the middle of the solution space 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.

4.2 Algorithms

Algorithm implementation is the main functionality 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 developed. Algorithms were implemented in a specific order from easiest to hardest. This allowed a staged approach to implementation to be carried out. The more straightforward algorithm implementations allow feedback and improvements to be made in preparation for the more complex algorithms. 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, which makes the progression of implementation easier, with Backtracking being brute-force and Hill Climb the easiest type of Stochastic algorithm, Simulated Annealing is a Hill Climb optimization and Genetic being an improvement on Simulated Annealing.

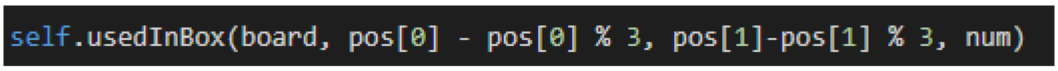
Each of the algorithms contains a "runAlgorithm" function that is called by the Test Bed when the algorithm needs to be executed. On start of this function, the start time is logged and on completion the stop time is logged. The difference provides the time taken for an algorithm and this is returned along with the algorithm iteration count.

4.2.1 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.

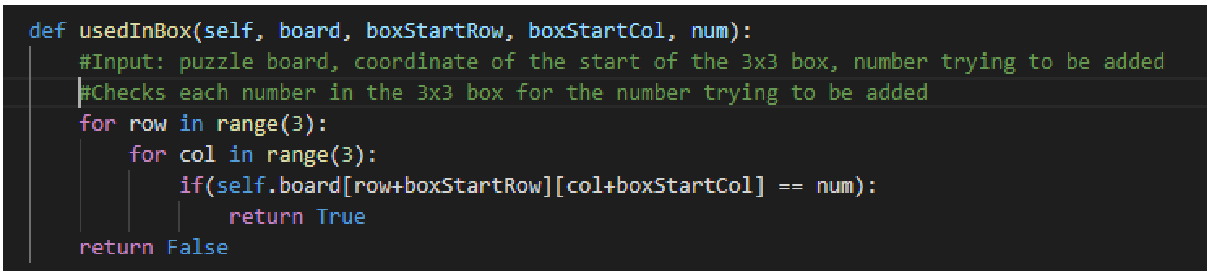
The algorithm has a simplistic design and therefore the implementation is mostly straightforward with a few challenging aspects. The first challenge involved detecting errors as an important part of this algorithm is ensuring each number entered does not cause conflicts within the puzzle.

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.



```
self.usedInBox(board, pos[0] - pos[0] % 3, pos[1]-pos[1] % 3, num)
```

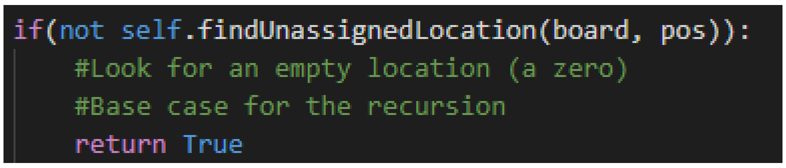
Figure 4.5 Code snippet for getting sub-box of cell



```python
def usedInBox(self, board, boxStartRow, boxStartCol, num):
    #Input: puzzle board, coordinate of the start of the 3x3 box, number trying to be added
    #Checks each number in the 3x3 box for the number trying to be added
    for row in range(3):
        for col in range(3):
            if(self.board[row+boxStartRow][col+boxStartCol] == num):
                return True
    return False
```

Figure 4.6 Code snippet for iterating through sub-box

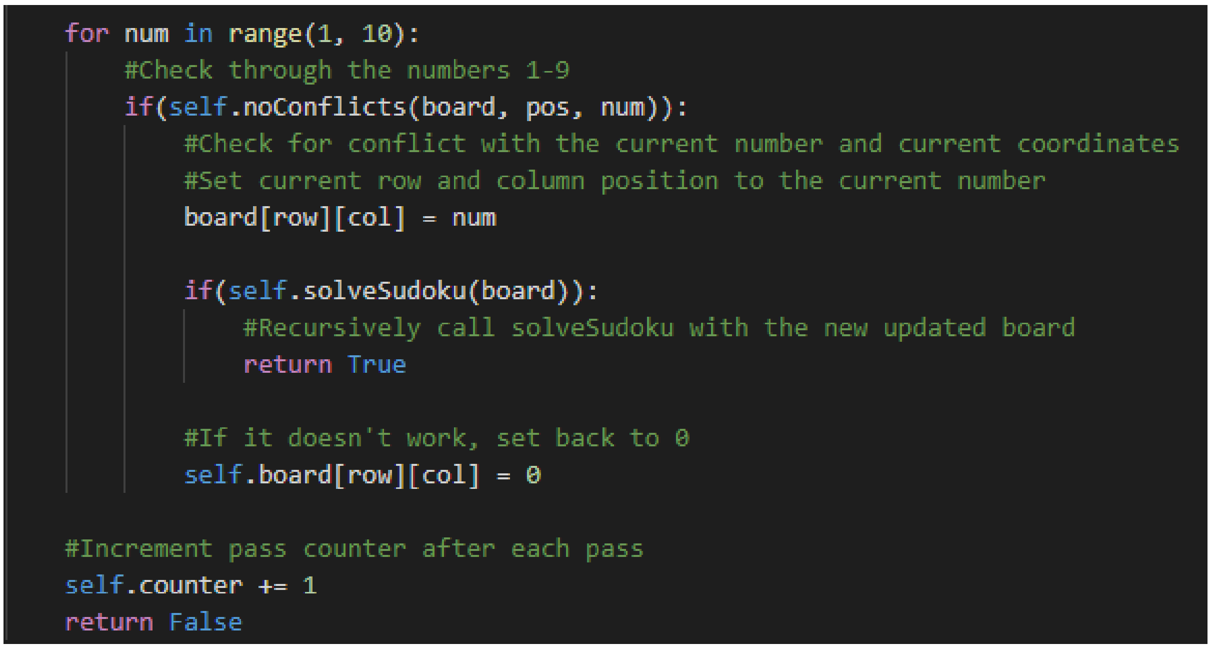
The backtracking function works by recursively calling the solve function for each new cell that is being worked on. This results in the algorithm stack containing all the previous cells' states, meaning if the algorithm backtracks the previous cell can be popped off the stack and continue from where it was paused when the recursion occurred.



```python
if(not self.findUnassignedLocation(board, pos)):
    #Look for an empty location (a zero)
    #Base case for the recursion
    return True
```

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, indicating to all previous states that the puzzle is solved.



```python
    for num in range(1, 10):
        #Check through the numbers 1-9
        if(self.noConflicts(board, pos, num)):
            #Check for conflict with the current number and current coordinates
            #Set current row and column position to the current number
            board[row][col] = num

            if(self.solveSudoku(board)):
                #Recursively call solveSudoku with the new updated board
                return True

            #If it doesn't work, set back to 0
            self.board[row][col] = 0

    #Increment pass counter after each pass
    self.counter += 1
    return False
```

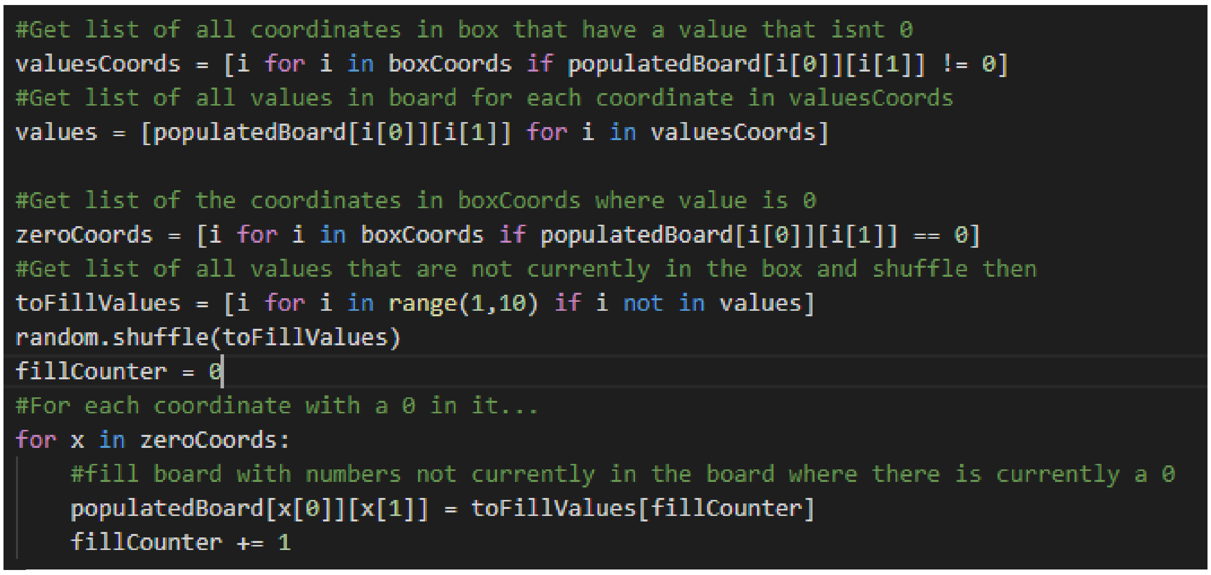
Figure 4.8 Code snippet for backtracking fixed loop

4.2.2 Hill Climb

The premise of Hill Climb involves the algorithm incrementally moving towards the optimum solution like that of climbing a hill. It is more efficient than brute-force for finding the optimum solution for a problem as it only increments towards the solution and therefore does not search the whole solution space.

Initialisation of the puzzle is fundamental as it is important for making changes to move towards a solution and for calculating the number of errors, also one of the steps used in all stochastic algorithm.

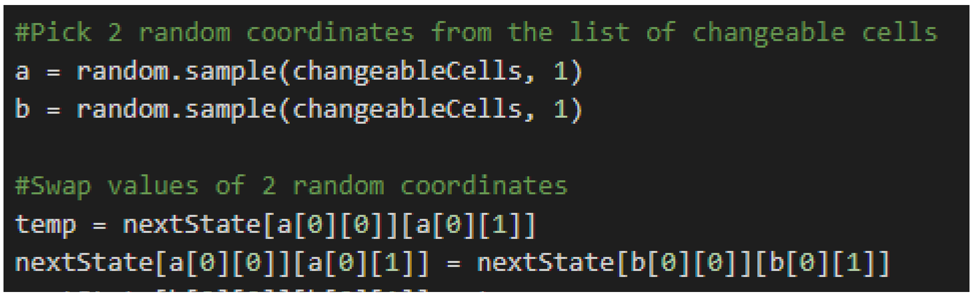
The initialisation process is implemented by setting all 3x3 sub-box to contain the numbers 1-9. For each box, the fixed numbers are counted to determine which numbers are missing. These are then added to all the zero value slots at random, allowing all the numbers to exist in each sub-box, this is illustrated in fig 4.9.



```
#Get list of all coordinates in box that have a value that isnt 0
valuesCoords = [i for i in boxCoords if populatedBoard[i[0]][i[1]] != 0]
#Get list of all values in board for each coordinate in valuesCoords
values = [populatedBoard[i[0]][i[1]] for i in valuesCoords]

#Get list of the coordinates in boxCoords where value is 0
zeroCoords = [i for i in boxCoords if populatedBoard[i[0]][i[1]] == 0]
#Get list of all values that are not currently in the box and shuffle then
toFillValues = [i for i in range(1,10) if i not in values]
random.shuffle(toFillValues)
fillCounter = 0
#For each coordinate with a 0 in it...
for x in zeroCoords:
    #fill board with numbers not currently in the board where there is currently a 0
    populatedBoard[x[0]][x[1]] = toFillValues[fillCounter]
    fillCounter += 1
```

Figure 4.9 Code snippet for puzzle initialisation

Climbing towards an optimal solution is done using 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.

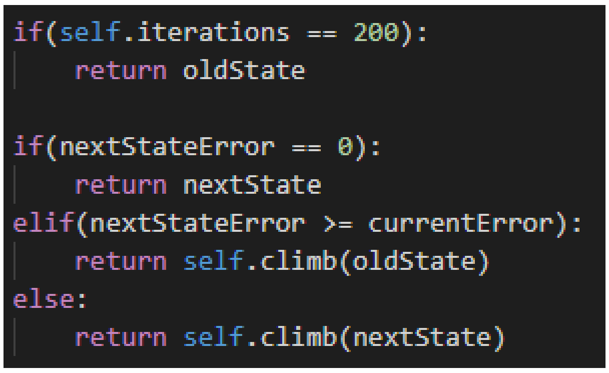


```
#Pick 2 random coordinates from the list of changeable cells
a = random.sample(changeableCells, 1)
b = random.sample(changeableCells, 1)

#Swap values of 2 random coordinates
temp = nextState[a[0][0]][a[0][1]]
nextState[a[0][0]][a[0][1]] = nextState[b[0][0]][b[0][1]]
```

Figure 4.10 Code snippet of swapping of cells in Hill Climb

The error calculation operates 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 are then taken and the difference between the length of the set and 9 is added to the score. 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 provide 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 along with a list of conditionals, 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 the recursion is halted. The second part is based on the number of errors in both solutions and depending on that, the more optimal solution is taken into the next iteration.

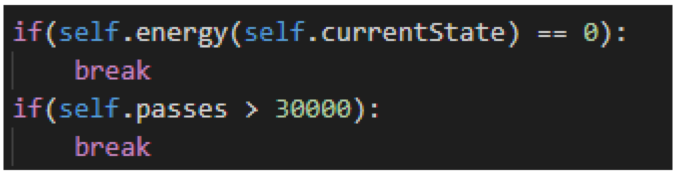


```
if(self.iterations == 200):
    return oldState

if(nextStateError == 0):
    return nextState
elif(nextStateError >= currentError):
    return self.climb(oldState)
else:
    return self.climb(nextState)
```

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

Iterations allow this algorithm to deal with finding local minima as there is a high probability when solving a Sudoku puzzle that the algorithm will go down the wrong path towards a solution. To fix this, the algorithm must loop back to re-initialise the puzzle and start the climbing again to find a different 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 creating better solutions. Therefore, seen in fig 4.12, there is a timeout based on the number of re-initialisation of the puzzle have been made that stops the algorithm from running infinitely.



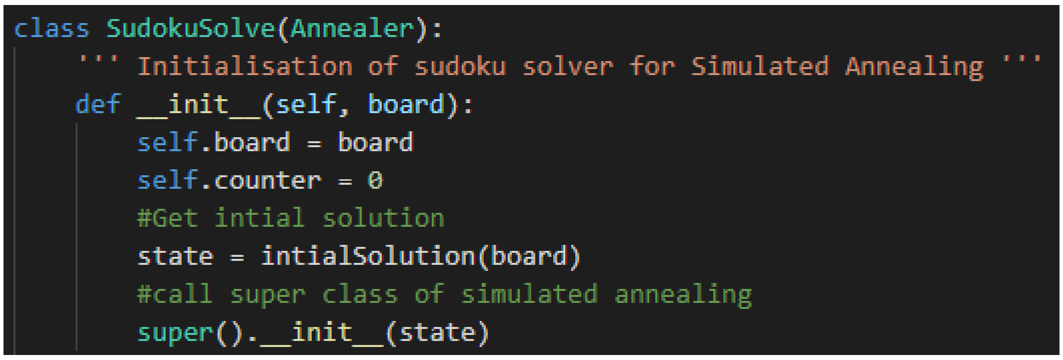
```
if(self.energy(self.currentState) == 0):
    break
if(self.passes > 30000):
    break
```

Figure 4.12 Code snippet of timeout condition in Hill Climb

4.2.3 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. One 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 two functions and a set of parameters and performs the Simulated Annealing on an array. Using Python provided access to these libraries which allowed concentration on other parts of the project. This library is inherited by a class, seen in fig 4.13, and the two functions that are used by the library called move and energy are implemented in this class, these functions are almost identical to that in Hill Climb.



```python
class SudokuSolve(Annealer):
    ''' Initialisation of sudoku solver for Simulated Annealing '''
    def __init__(self, board):
        self.board = board
        self.counter = 0
        #Get intial solution
        state = intialSolution(board)
        #call super class of simulated annealing
        super().__init__(state)
```

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

When running Simulated Annealing the library requires a set of parameters to control temperature changes. If these are set incorrectly then the puzzle may not be solved.

The library requires these four input parameter:

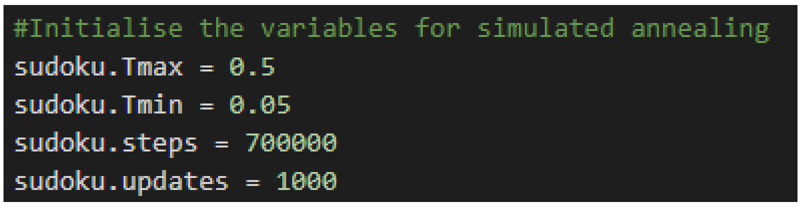
- Start temperature
- End temperature
- Number of iterations
- Number of updates

Temperature start and end are important as the temperature defines the probability that a bad solution is accepted by the algorithm. If the starting temperature is too high, then worse solutions are accepted for too long and an optimal solution is never reached. If too low, then the solution space is not wide enough 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 completing 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 Sudoku solving, there needs to be a 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. 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. 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.



```
#Initialise the variables for simulated annealing
sudoku.Tmax = 0.5
sudoku.Tmin = 0.05
sudoku.steps = 700000
sudoku.updates = 1000
```

Figure 4.14 Code Snippet of Simulated Annealing parameters

As seen in fig 4.14, 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 in the command line when the algorithm is run which gives an insight into how close the algorithm is to completion.

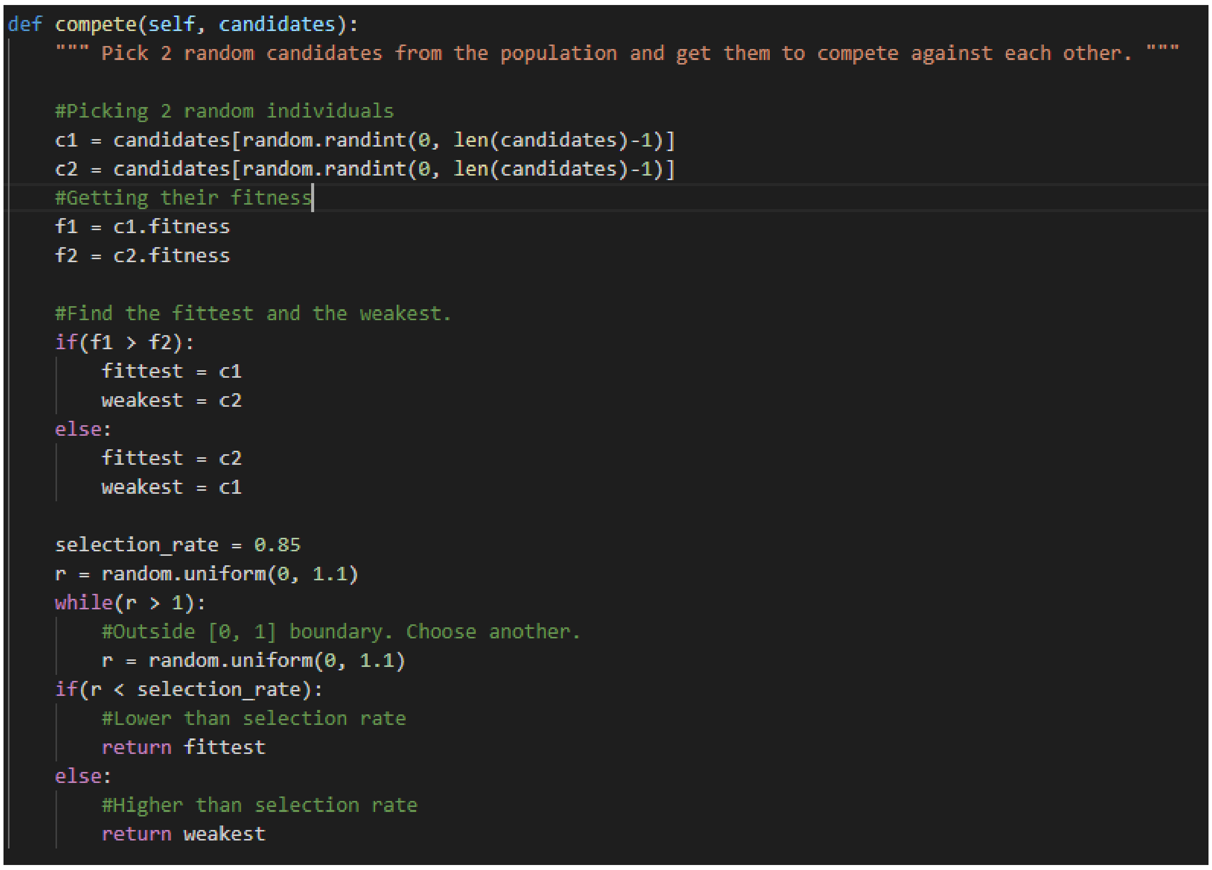
4.2.4 Genetic Algorithm

Genetic Algorithm (GA) is the most complex algorithm implemented due to the complexity of the components which make up the algorithm. There exists a Python library which implements GA, but it cannot be effectively used for Sudoku solving. This is because it is focused on being used for conventional GA use which is finding the global optimum of a problem when there could be multiple.

This algorithm implementation makes use of objects, as stated in the design, with each individual potential solution 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 of each individual to give the strongest candidates to mutate and crossover.

The algorithm uses the same method for initialisation of puzzles as the previous algorithms except it uses the puzzle rows instead of the sub-box constraint as it makes crossover easier to accomplish.

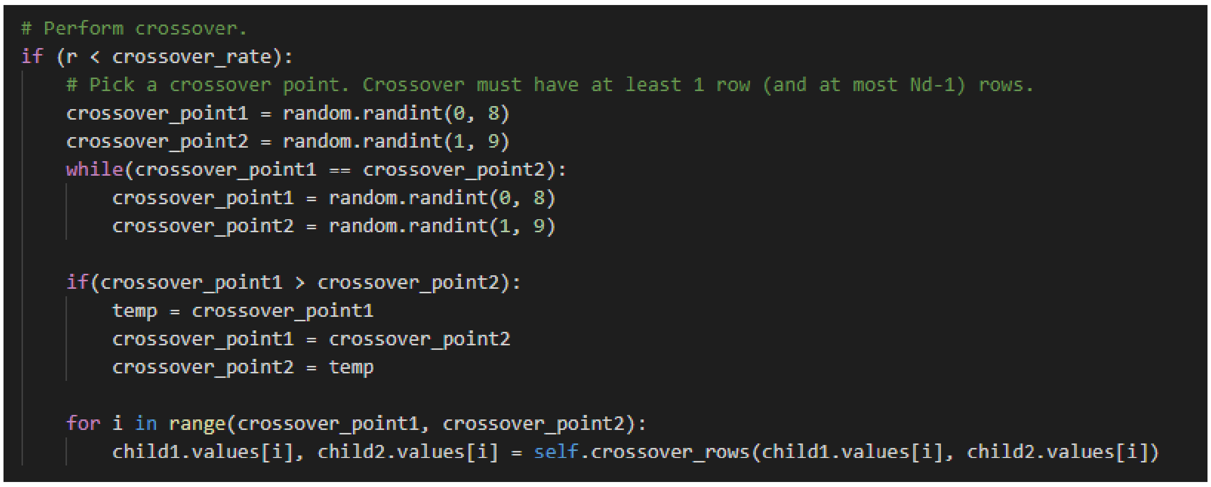
A number of processes are performed on the population when a new population is being created. The first is in the tournament class, seen in fig 4.15, that takes two individuals and using a selection probability either choses the fitter or weaker individual to ensure that the population is a wide solution space and does not converge on an answer too quickly.



```python
def compete(self, candidates):
    """ Pick 2 random candidates from the population and get them to compete against each other. """

    #Picking 2 random individuals
    c1 = candidates[random.randint(0, len(candidates)-1)]
    c2 = candidates[random.randint(0, len(candidates)-1)]
    #Getting their fitness
    f1 = c1.fitness
    f2 = c2.fitness

    #Find the fittest and the weakest.
    if(f1 > f2):
        fittest = c1
        weakest = c2
    else:
        fittest = c2
        weakest = c1

    selection_rate = 0.85
    r = random.uniform(0, 1.1)
    while(r > 1):
        #Outside [0, 1] boundary. Choose another.
        r = random.uniform(0, 1.1)
    if(r < selection_rate):
        #Lower than selection rate
        return fittest
    else:
        #Higher than selection rate
        return weakest
```

Figure 4.15 Code snippet of compete function in Genetic Algorithm

Crossover is seen in fig 4.16, it takes two parent individuals from the population and selects two rows in the puzzle. Then all the rows between the two points are swapped from one individual to the other to create two new child individuals.



```python
# Perform crossover.
if (r < crossover_rate):
    # Pick a crossover point. Crossover must have at least 1 row (and at most Nd-1) rows.
    crossover_point1 = random.randint(0, 8)
    crossover_point2 = random.randint(1, 9)
    while(crossover_point1 == crossover_point2):
        crossover_point1 = random.randint(0, 8)
        crossover_point2 = random.randint(1, 9)

    if(crossover_point1 > crossover_point2):
        temp = crossover_point1
        crossover_point1 = crossover_point2
        crossover_point2 = temp

    for i in range(crossover_point1, crossover_point2):
        child1.values[i], child2.values[i] = self.crossover_rows(child1.values[i], child2.values[i])
```

Figure 4.16 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 rows are used instead of sub-boxes. A random row is selected and then within that row, two 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 repeated for every individual in the population after each one has gone through crossover process. The result of this is a new population and the whole process can then be repeated.

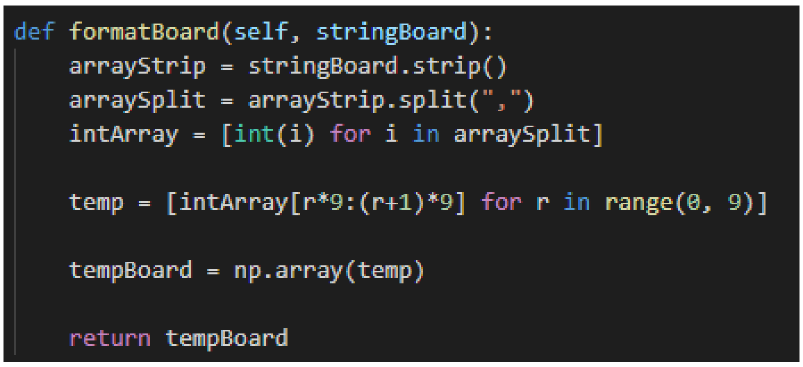
The work on implementation and testing the algorithm was complex and while the main logic and algorithm implementation was developed, time constraints meant that this could not be completed. The work to progress and complete the algorithm has been highlighted in future work and due to the way the project was structured, the other algorithms are available for comparison.

4.3 Test Bed

The test bed consists of two components with the designed aim of bringing the system together and provide a result that is effective and readable. The first component interacts with the Sudoku generation and the algorithms, extracting puzzles from the files and passing them to algorithms. The second is the GUI which provides the user an easy way to interact with the system and the analysis element which displays the data in an easy and understandable way.

4.3.1 Puzzle and Algorithm Interaction

Puzzle Generation, as we saw previously in the chapter, stores the generated puzzles in a text file. First the file is opened, and a specific line is identified. The format of the stored board is a one-dimensional list in the text file, and this is required to be reformatted into a two-dimensional NumPy array, illustrated in fig 4.17, as the algorithm input. The text file board is first split by each number then grouped into rows of nine numbers and formatted as the NumPy array.



```
def formatBoard(self, stringBoard):
    arrayStrip = stringBoard.strip()
    arraySplit = arrayStrip.split(",")
    intArray = [int(i) for i in arraySplit]

    temp = [intArray[r*9:(r+1)*9] for r in range(0, 9)]

    tempBoard = np.array(temp)

    return tempBoard
```

Figure 4.17 Code snippet of board formatting in Test Bed

This array is passed to a runAlgorithm function that is passed a number, corresponding to each algorithm, depending on which algorithm is selected to be run. Then each puzzle from the text file is called one at a time and passed to the algorithm to be run. The results are stored in arrays to be returned after all the algorithms have been complete.

4.3.2 GUI and Analysis

The GUI and analysis are linked together to take advantage of the analysis results being output to the user via the GUI and therefore the analysis calculations are implemented within the GUI class.

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

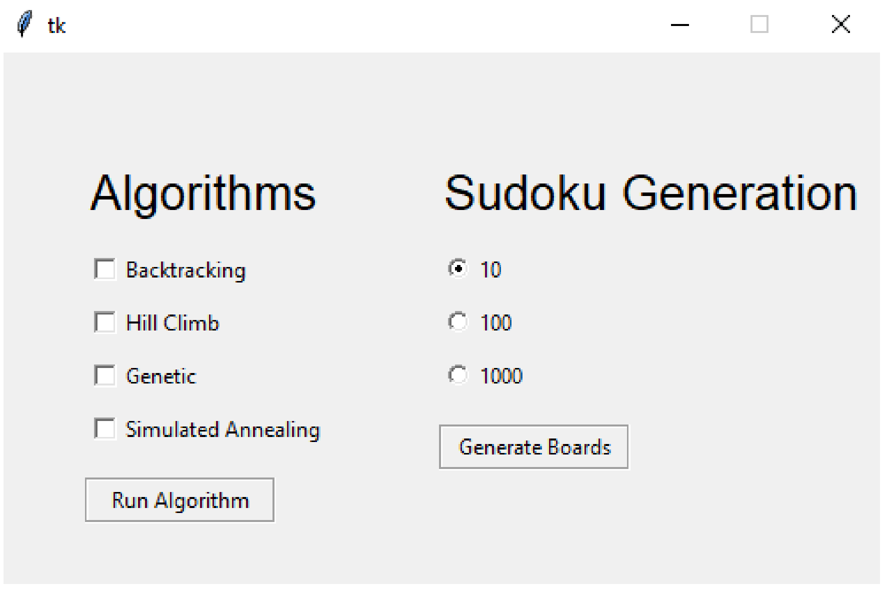


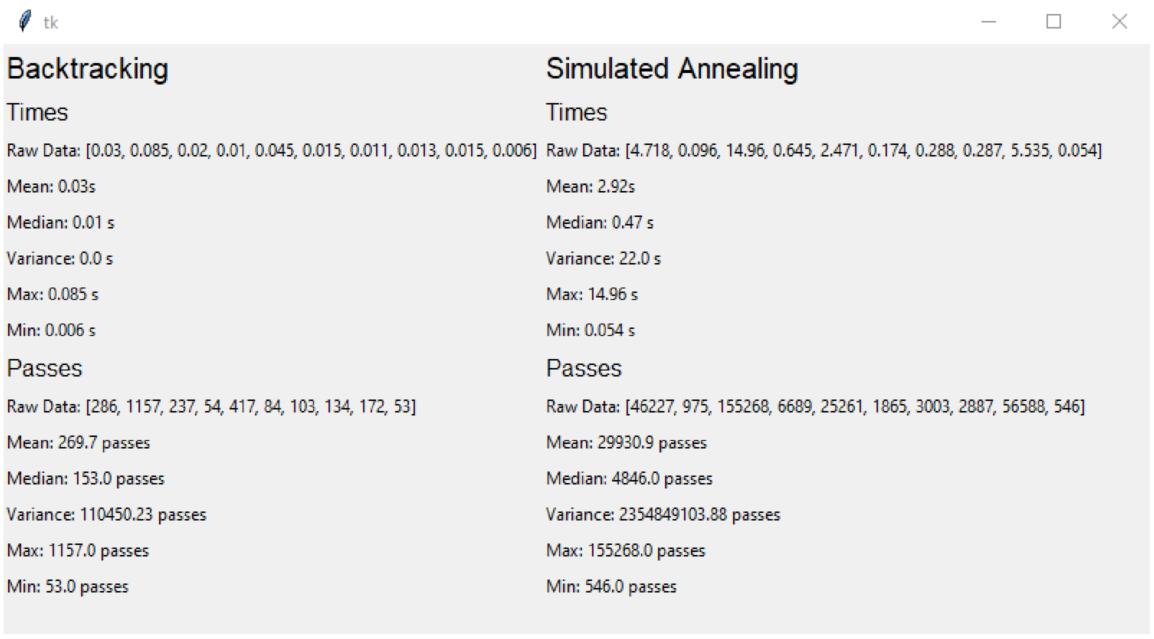
Figure 4.18 GUI for algorithm selection and puzzle generation

Fig 4.18 shows the GUI for the main window. The algorithms on the left are checkboxes that allow the user to select any number of the algorithms that are to be compared and the right consists of the number of Sudoku puzzles to be generated using radio buttons. When puzzle 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 a limitation in the updating of 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 two different measures, in text and graphical form.



Backtracking

Times

Raw Data: [0.03, 0.085, 0.02, 0.01, 0.045, 0.015, 0.011, 0.013, 0.015, 0.006]

Mean: 0.03s

Median: 0.01 s

Variance: 0.0 s

Max: 0.085 s

Min: 0.006 s

Passes

Raw Data: [286, 1157, 237, 54, 417, 84, 103, 134, 172, 53]

Mean: 269.7 passes

Median: 153.0 passes

Variance: 110450.23 passes

Max: 1157.0 passes

Min: 53.0 passes

Simulated Annealing

Times

Raw Data: [4.718, 0.096, 14.96, 0.645, 2.471, 0.174, 0.288, 0.287, 5.535, 0.054]

Mean: 2.92s

Median: 0.47 s

Variance: 22.0 s

Max: 14.96 s

Min: 0.054 s

Passes

Raw Data: [46227, 975, 155268, 6689, 25261, 1865, 3003, 2887, 56588, 546]

Mean: 29930.9 passes

Median: 4846.0 passes

Variance: 2354849103.88 passes

Max: 155268.0 passes

Min: 546.0 passes

Figure 4.19 GUI for result display

The first is the raw data from the experiment, which comprises of the times and number of passes, seen in fig 4.19. 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 mean, median, variance, and maximum and minimum are calculated using the Statistics Python library. These statistics can be extended using library functions.

The second method of displaying results is in graphical form.

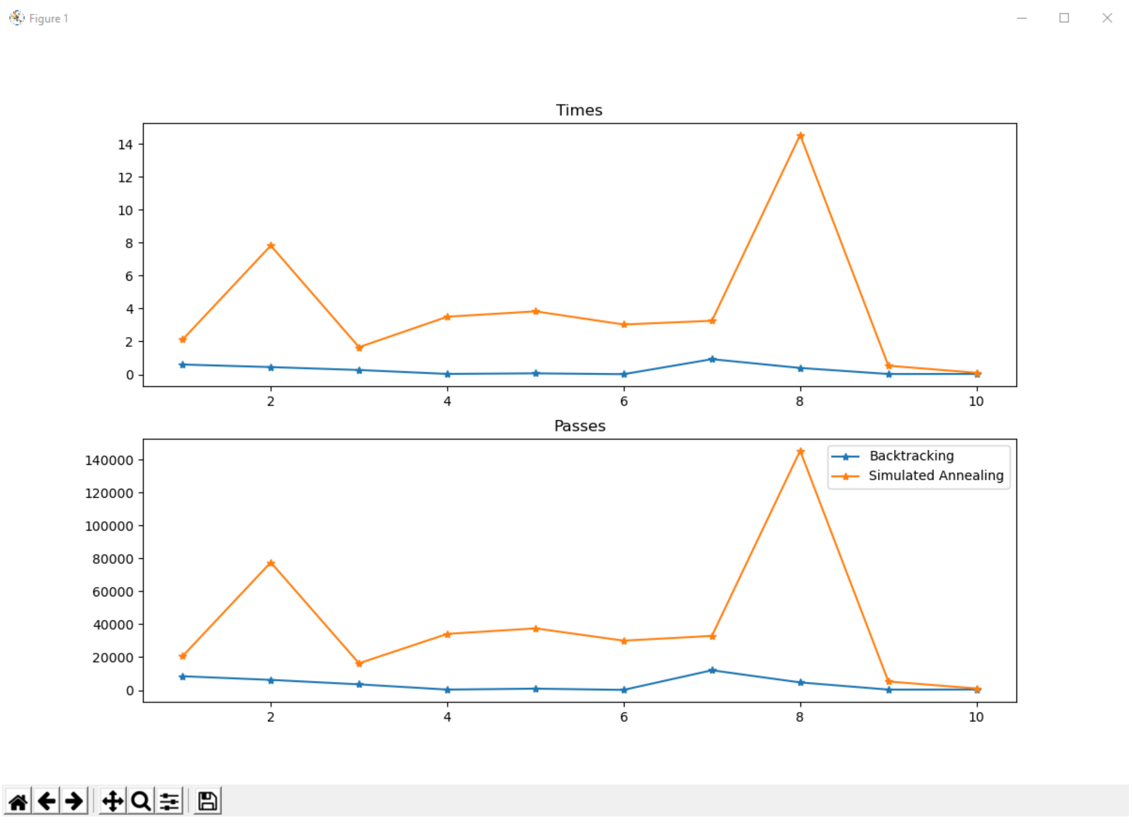


Figure 4.20 GUI graph of results

Fig 4.20 displays the same raw data that is shown in the list, however, is a lot easier to identify which algorithm is the most efficient due to the plotting and use of different colours. This is implemented 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.

5 Testing

Testing of the system is important as it allows us to make sure that the implementation developed is working as intended and giving the correct and expected results. Exhaustive testing is carried out and even though no system is bug free, as many bugs as possible are identified by the testing. The structure for this testing [40] 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 and are seen in full in Appendix A.

5.1 Unit Testing

Unit testing is known as white-box testing and involves testing individual components of the software, the purpose being to make sure each unit of the software is functioning as designed. The components in this project are the methods in the different sections: Algorithms, Sudoku Generation and Test Bed.

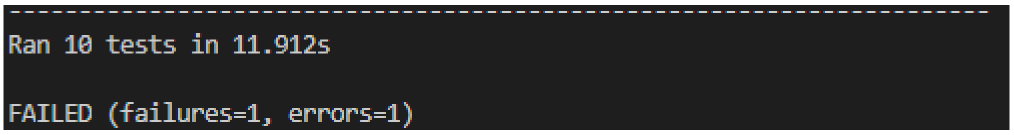
Unit testing in Python can be done with a number of different libraries but the easiest is Unittest [41] which is integrated into standard Python. These tests use assert statements to check if actual output equals expected output. Unit testing was carried out continuously during the development by testing methods as they were implemented.

5.1.1 Algorithm Testing

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

Between the algorithms, there are identified areas of common functionality used in all of the algorithms, for example puzzle initialisation, which means they only need to be tested once. Algorithm testing was carried out in order of implementation so first was Backtracking then Hill Climb, Simulated Annealing and Genetic.

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. This was achieved by passing in a puzzle to solve and comparing the result to the expected solved puzzle.



```
----------------------------------------------------------------------
Ran 10 tests in 11.912s

FAILED (failures=1, errors=1)
```

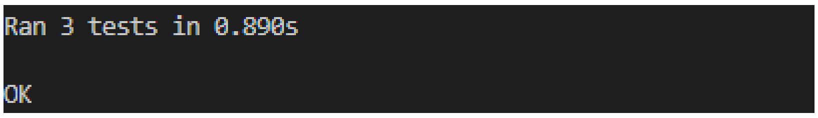
Figure 5.1 Output of algorithm unit tests

The output for the unit tests run on the algorithms are in figure 5.1, showing all the tests passed except for one failure and one error. The failed test is 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 subsequently returns an error.

5.1.2 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 components due to a different puzzle being created by the algorithm 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.

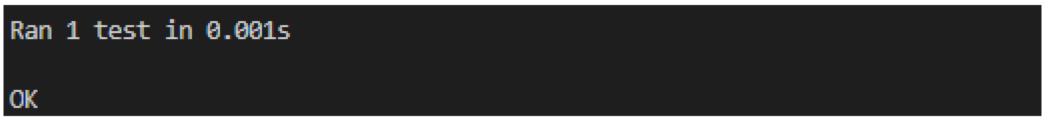
The Test Bed also has limitations in unit testing due to the majority of the functionality in the test bed being embedded in the GUI. This means that the most appropriate level of testing in this case is system testing. The unit tests for this section are located in Appendix A2.



```
Ran 3 tests in 0.890s

OK
```

Figure 5.2 Output of Sudoku Generation unit tests

Sudoku Generation test results are displayed in fig 5.2, showing that all tests passed. This is a small number of tests due, as discussed earlier, to the overlap with Backtracking and generation having a limited number of methods.



```
Ran 1 test in 0.001s

OK
```

Figure 5.3 Output of Test Bed unit tests

The results for unit testing of the Test Bed are seen in fig 5.3. It only consists of retrieval of the board due to, as already mentioned, the functions being embedded within the GUI and therefore tested in the system testing. The test involves opening the file, retrieving a specific line, and formatting the data. A call is then made, and the results are compared.

5.2 System Testing

System testing is black box which means there is no knowledge of the inner code design and testing is done from the GUI. Tests were written and carried out manually with results compared against expected outputs.

The GUI testing consisted of tests of the Sudoku generation and the algorithm selection. The radiobuttons are tested by clicking each one making sure the selected number changes in the GUI and on completion of the generation, the text file can be checked that the correct number of puzzles were created. Sudoku generation tests are shown in Appendix A3.

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 time and iteration results produced by the algorithms, these varied widely dependent on the puzzle and randomness. The time taken was challenging to test as rough estimates of the expected time had to be used. This had the potential to be inaccurate but provided a reliable proof of correctness.

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 produced 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 algorithm 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 due to the algorithm timing out instead of crashing. Test number 29, the Genetic algorithm test, fails as the algorithm produces an error and crashes before being able to produce a result.

5.3 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 numbers of puzzles with the Hill Climb and Simulated Annealing algorithms there was found to be an issue with swapping cells. There was a edge case that potentially caused the number of changeable cells in a box to be less than 2 and therefore when 2 were selected at random, the algorithm crashed. This was fixed by checking the number of changeable cells before swapping 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. This resulted in an empty window of results and an empty graph. Fixing this involved creating a check for no algorithms being selected and an error message being output to the user.

- 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 allowed an exact copy to be made of the array.

6 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.

6.1 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 it is unknown.

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, a statistical approach can be taken. The important statistics for each algorithm were the mean, median, variance and, minimum and maximum as there are a number of factors that determine the best algorithm.

The most accurate results are created 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 graph points, meaning most of the data is unreadable. It does, however, give insight into the performance of the algorithms.

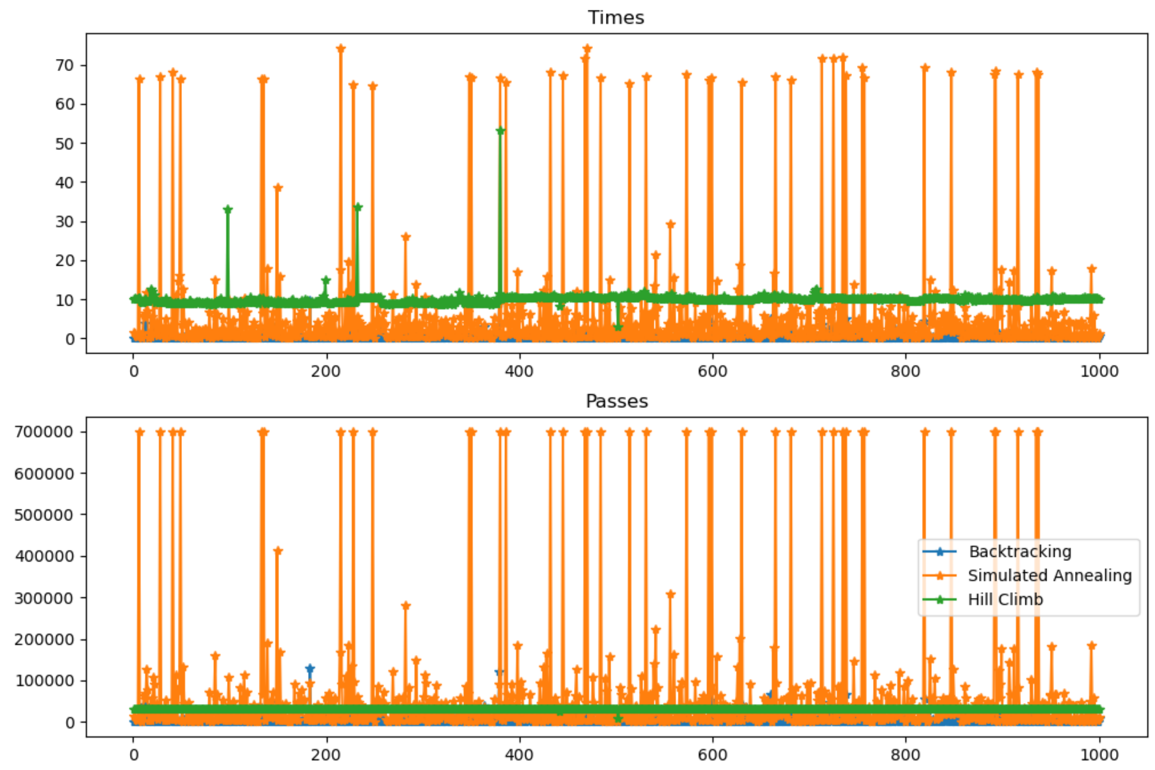


Figure 6.1 Result graph for 1000 puzzles

The results we can see from fig 6.1 is that Simulated Annealing has the biggest variance between the lowest and highest times and passes whereas the other two algorithms 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, displayed 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.

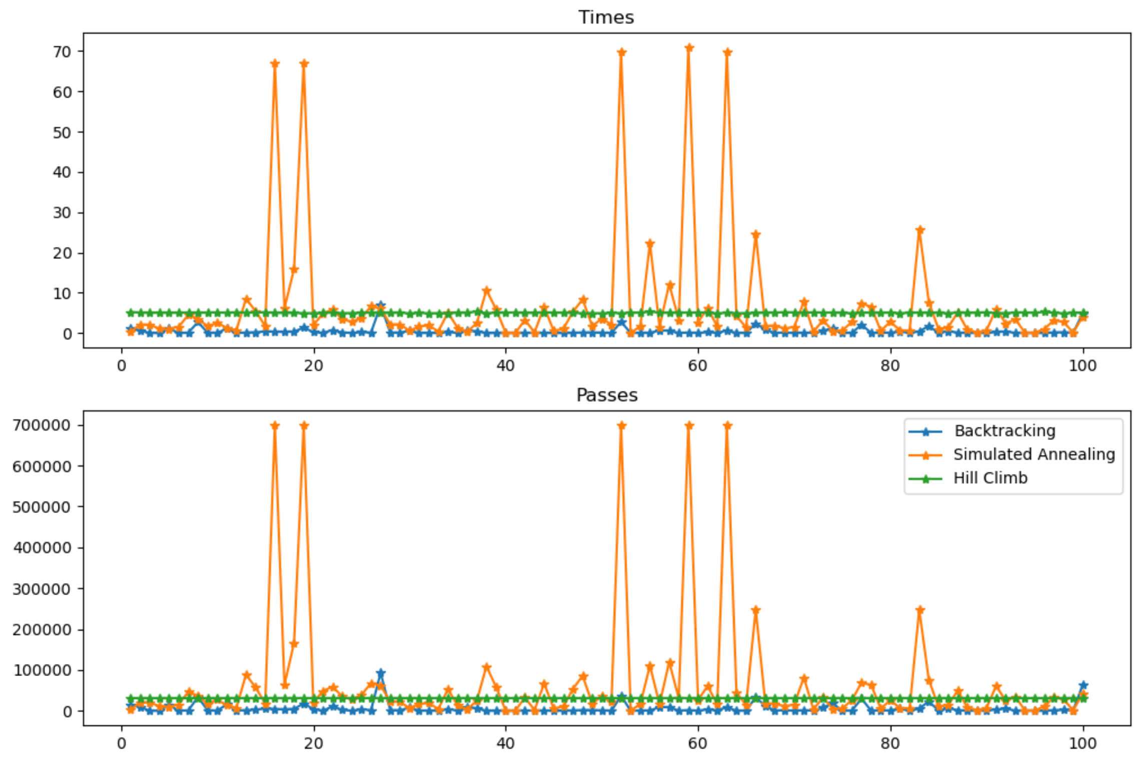
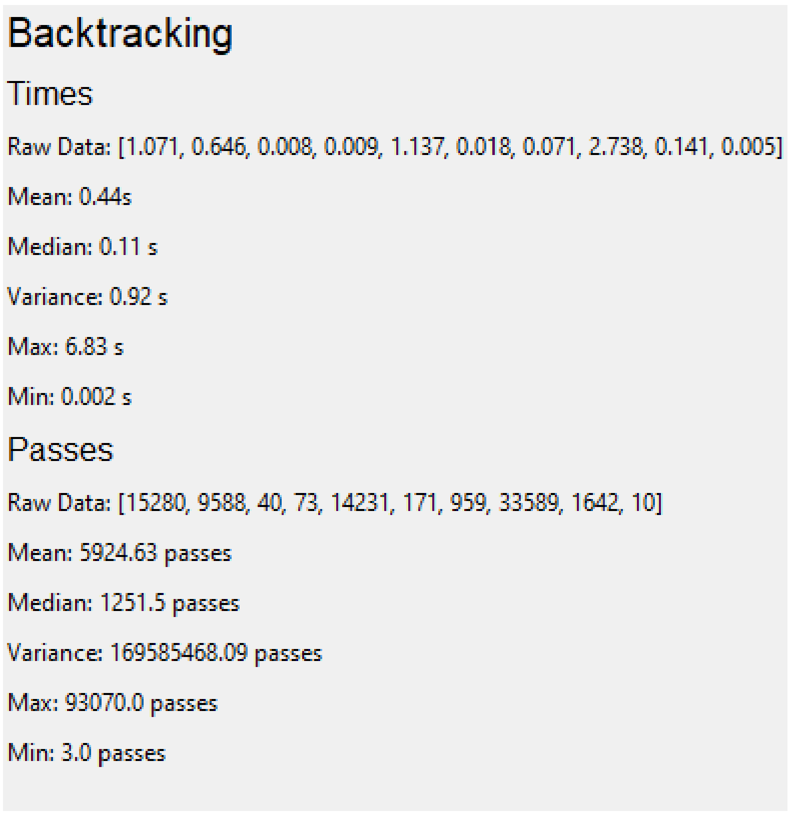


Figure 6.2 Result graph for 100 puzzles

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

As seen in fig 6.3, there are a number of statistics given for each algorithm. These are calculated using all the raw data but for the display, the raw data has been capped at the first ten results. The full results are found in Appendix C.



```
Backtracking
Times
Raw Data: [1.071, 0.646, 0.008, 0.009, 1.137, 0.018, 0.071, 2.738, 0.141, 0.005]
Mean: 0.44s
Median: 0.11 s
Variance: 0.92 s
Max: 6.83 s
Min: 0.002 s
Passes
Raw Data: [15280, 9588, 40, 73, 14231, 171, 959, 33589, 1642, 10]
Mean: 5924.63 passes
Median: 1251.5 passes
Variance: 169585468.09 passes
Max: 93070.0 passes
Min: 3.0 passes
```

Figure 6.3 Raw output and statistics results for Backtracking algorithm

Mean	Backtracking	Hill Climb	Simulated Annealing
Times (s)	0.44	4.91	8.02
Passes	5,924	30,200	78,726

Table 6.1 Mean results from running algorithms

Table 6.1 shows the mean of all the results, indicating to us that Backtracking 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 the other algorithms with a time of under half a second against over five seconds of the other two.

Median	Backtracking	Hill Climb	Simulated Annealing
Times (s)	0.11	4.91	2.63
Passes	1,251	30,200	27,523

Table 6.2 Median results from running algorithms

The median results in table 6.2 show differerent results from the mean data results in table 6.1. It does, however, reinforce the results that Backtracking is the fastest and most efficient. The difference identified in the results is that Simulated Annealing is faster and more efficient than Hill Climb in terms of median.

Max/Min	Backtracking	Hill Climb	Simulated Annealing
Times (s)	6.9/0.002	5.2/4.7	73.7/0.043
Passes	93,070/3	30,200/30,200	700,000/406

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 in table 6.3. It reinforces that Backtracking is the best algorithm but it shows that the minimum time from Simulated Annealing is closer than expected to Backtracking although, the maximum values 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 without completion. Simulated Annealing also reaches this maximum iterations, however, as shown by the mean and median 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, the brute-force approach is the best in terms of speed and efficiency. 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.

The results could change if the size and complexity of the puzzles were increased, for example 16x16 puzzle dimensions. In this case brute-force would require a larger number of iterations to check all possible solutions. This would allow the more complex algorithms to show their more efficient designs.

6.2 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, three were successful and therefore even though Genetic was attempted and not fully implemented, the requirement was satisfied.

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

The test bed offers the user an option of two functions to 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 an option 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 to 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 and benchmarking 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.

This could be improved by allowing the user to select a specific number of puzzles but for this system the powers of ten are sufficient and make the GUI more user friendly.

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 displayed and the best algorithm identified. 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 algorithms to be compared. With each algorithm run in order from top to bottom of the list depending on how many are selected.

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. The tool provides enough information that a user with knowledge of Sudoku and algorithms would have sufficient information to understand the tool and results produced. 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 and similar statistics of all the data are 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.

6.3 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 [42]. 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.

7 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.

7.1 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 provide modular separation. This section generated a number of puzzles based on the defined number given to the algorithm and output that number of test puzzles to a text file. The third 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 controlled through an interface, with further calls dependent on the inputs made by the user.

These sections were implemented and tested one by one as the system relies on all these components working and therefore testing incrementally was essential. Testing uncovered a number of bugs especially in 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, since blackbox testing requires no knowledge of inner workings of code, and not all test cases were covered.

7.2 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 popular 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. Four algorithms were initially selected, but only three were completed fully due to the challenges posed by the complexities of the Genetic algorithm and therefore three algorithms were completed but not all selected algorithms were completed.

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 four 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 timeframe which was disappointing.

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 through graphs.

Although this is the fourth objective, the test bed was incrementally developed alongside the other components of the system as more functionality was added. During design and development, we found that the test bed required more functionality than solely the comparison of algorithms as it needed to act as a base for the other components 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 increase the amount of test data and therefore used for each algorithm being compared.

Objective 6: Implement algorithms into the test bed

Algorithm implementation was carried out 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 explained the fourth algorithm is time dependant, there are only three working implementations of algorithms in the system, Genetic algorithm was implemented but not error free. 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.

7.3 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 absorb academic papers. This was a new experience for me as, although I had read papers previously, the extent that I had to read and finding useful information was a valuable skill I had to learn.

- **Knowledge of algorithms:** I came into this project with almost no knowledge of stochastic or machine learning algorithms. I have learnt during both the research and development of the algorithms that my knowledge has grown in this area and I appreciate the similarities and differences in the algorithms.

- **Python:** Although I had used basic Python while at school, using the language in an application space it was designed for increased my knowledge tremendously. Algorithms and data analysis are areas that interest me so having this opportunity to learn it in a focused task 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.

- **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.

7.4 Challenges

- **Research:** During the beginning of initial research, starting with a very limited knowledge base and trying to understand these difficulty concepts was challenging. Since academical papers expect a certain level of pre-existing knowledge of the topic, additional background research was required, and finding the motivation to do research on the algorithms especially was challenging to begin with.

- **System Size:** Dealing with the different algorithms as well as the puzzle generation and test bed all as different components was difficult to visualise to begin with. 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 - once I started commenting and leaving appropriate white space the code became more readable and maintainable.

- **Python Libraries:** Given the data being collected and the analysis being done I had to utilise a lot of different libraries for data that I had never used before. This was challenging 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 about why they are useful through the experience of using them.

- **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.

7.5 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 pivot the tool into more of a teaching tool.

- Changing to 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 and enabling potentially faster computation.

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] “The Four Levels of Software Testing,” Segue technologies, [Online]. Available: https://www.seguetech.com/the-four-levels-of-software-testing/. [Accessed 2020 May 2].

[41] “Unit testing framework,” Python, [Online]. Available: https://docs.python.org/3/library/unittest.html. [Accessed 2 May 2020].

[42] “Form Builder,” Newcastle University, [Online]. Available: https://services.ncl.ac.uk/itservice/collaboration-services/form-builder/. [Accessed 1 May 2020].

[43] D. Marshall, “Steepest Ascent Hill Climbing,” [Online]. Available: http://users.cs.cf.ac.uk/Dave.Marshall/AI1/steep.html. [Accessed 1 April 2020].

[44] 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.

[45] [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].

[46] [Online]. Available: https://images.mendix.com/wp-content/uploads/Artboard-1@2x-701x700.png. [Accessed 23 April 2020].

Table of Figures

Table of Tables

Appendices

Appendix A: Testing Done

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

A1: Algorithm Unit Tests

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

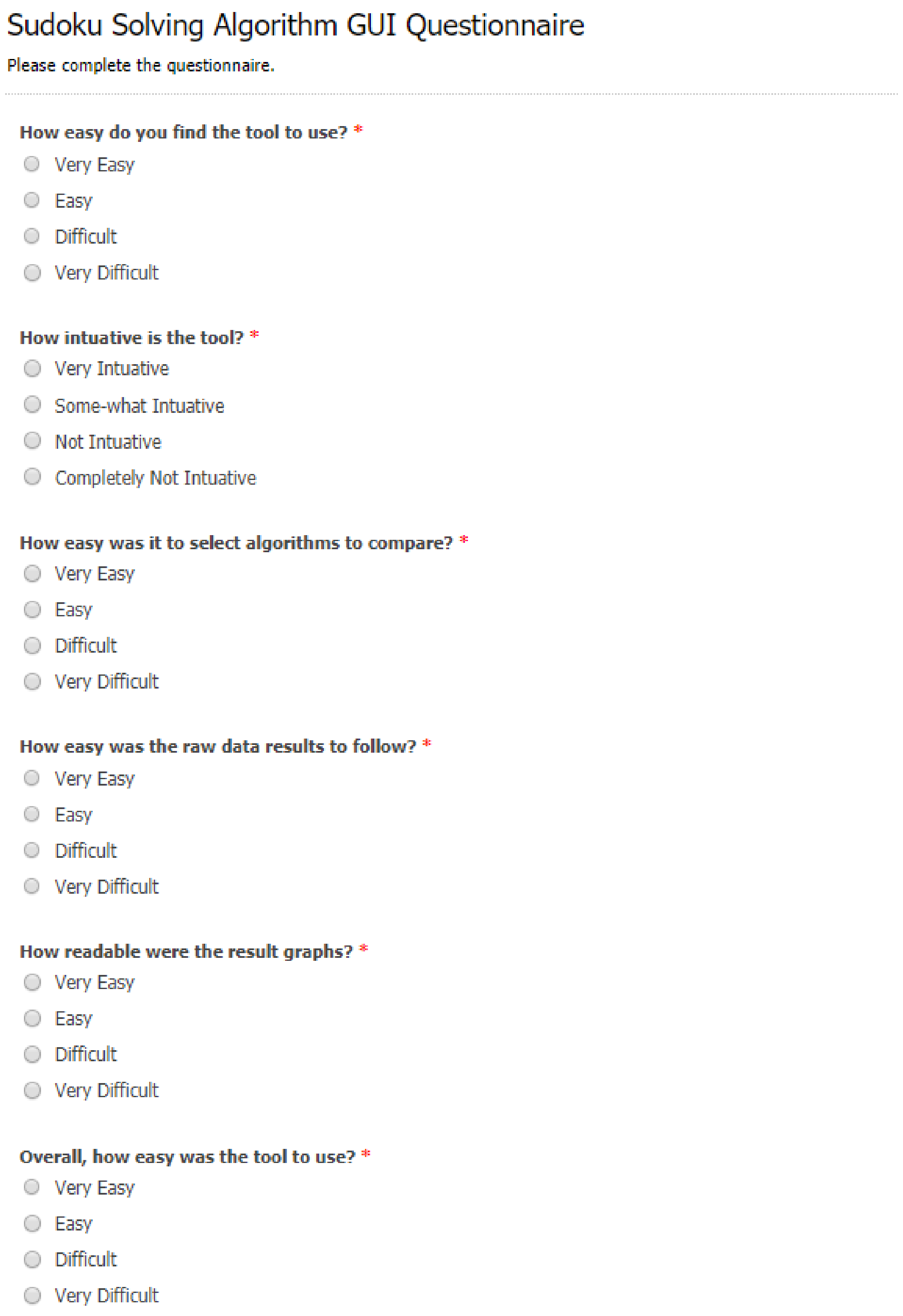
A3: GUI Puzzle Generation Tests

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



Sudoku Solving Algorithm GUI Questionnaire

Please complete the questionnaire.

How easy do you find the tool to use? *

- ○ Very Easy
- ○ Easy
- ○ Difficult
- ○ Very Difficult

How intuative is the tool? *

- ○ Very Intuative
- ○ Some-what Intuative
- ○ Not Intuative
- ○ Completely Not Intuative

How easy was it to select algorithms to compare? *

- ○ Very Easy
- ○ Easy
- ○ Difficult
- ○ Very Difficult

How easy was the raw data results to follow? *

- ○ Very Easy
- ○ Easy
- ○ Difficult
- ○ Very Difficult

How readable were the result graphs? *

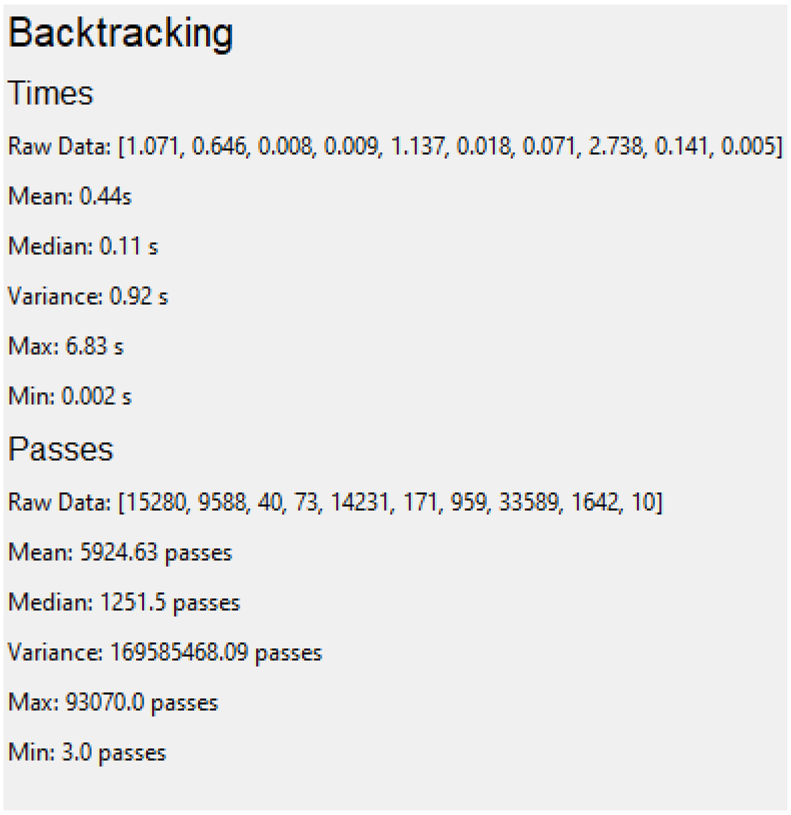
- ○ Very Easy
- ○ Easy
- ○ Difficult
- ○ Very Difficult

Overall, how easy was the tool to use? *

- ○ Very Easy
- ○ Easy
- ○ Difficult
- ○ Very Difficult

Appendix C: Full Results

C1: Backtracking



Backtracking

Times

Raw Data: [1.071, 0.646, 0.008, 0.009, 1.137, 0.018, 0.071, 2.738, 0.141, 0.005]

Mean: 0.44s

Median: 0.11 s

Variance: 0.92 s

Max: 6.83 s

Min: 0.002 s

Passes

Raw Data: [15280, 9588, 40, 73, 14231, 171, 959, 33589, 1642, 10]

Mean: 5924.63 passes

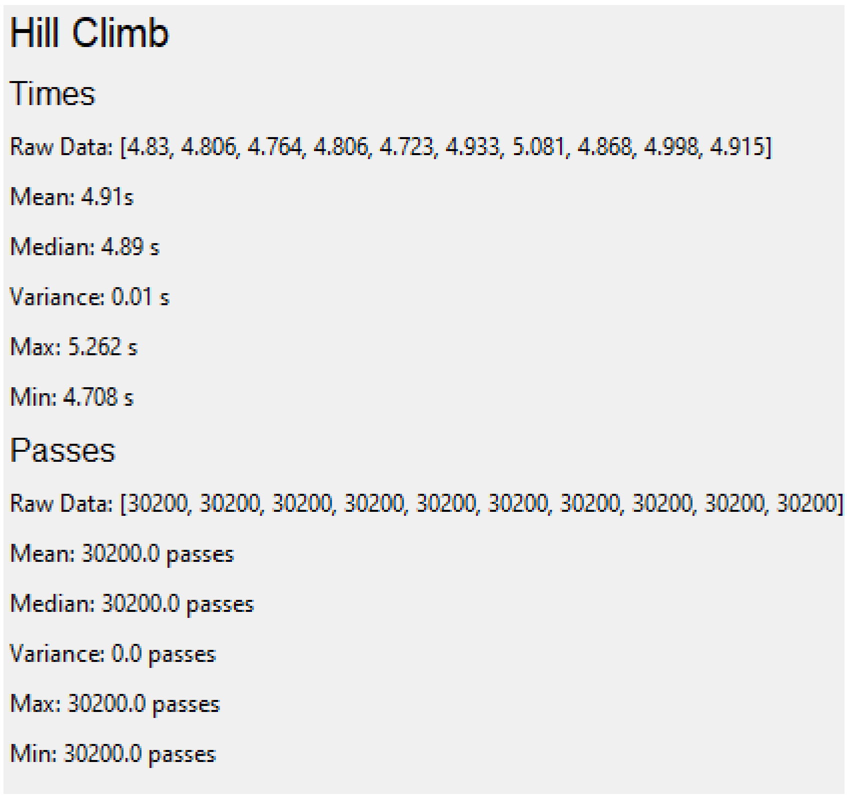
Median: 1251.5 passes

Variance: 169585468.09 passes

Max: 93070.0 passes

Min: 3.0 passes

C2: Hill Climb



Hill Climb

Times

Raw Data: [4.83, 4.806, 4.764, 4.806, 4.723, 4.933, 5.081, 4.868, 4.998, 4.915]

Mean: 4.91s

Median: 4.89 s

Variance: 0.01 s

Max: 5.262 s

Min: 4.708 s

Passes

Raw Data: [30200, 30200, 30200, 30200, 30200, 30200, 30200, 30200, 30200, 30200]

Mean: 30200.0 passes

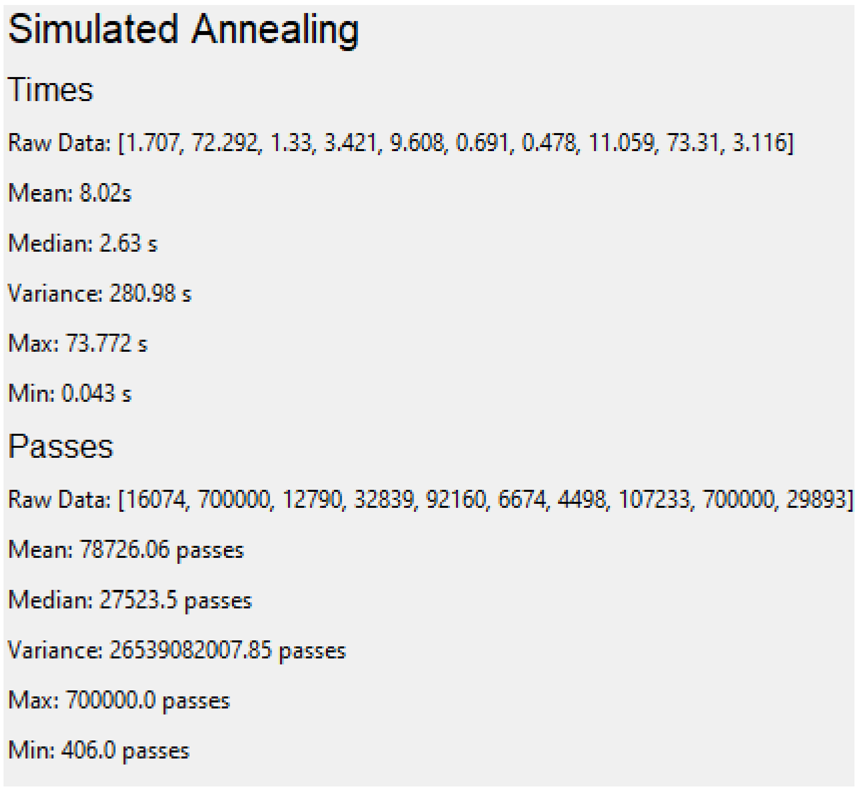
Median: 30200.0 passes

Variance: 0.0 passes

Max: 30200.0 passes

Min: 30200.0 passes

C3: Simulated Annealing



Simulated Annealing

Times

Raw Data: [1.707, 72.292, 1.33, 3.421, 9.608, 0.691, 0.478, 11.059, 73.31, 3.116]

Mean: 8.02s

Median: 2.63 s

Variance: 280.98 s

Max: 73.772 s

Min: 0.043 s

Passes

Raw Data: [16074, 700000, 12790, 32839, 92160, 6674, 4498, 107233, 700000, 29893]

Mean: 78726.06 passes

Median: 27523.5 passes

Variance: 26539082007.85 passes

Max: 700000.0 passes

Min: 406.0 passes