Comparison of Sudoku Solving Algorithms

Calum Harvey (Student ID: 170349985)

Computer Science

Supervisor: Dr Jason Steggles

Word Count: 13878

Abstract

Declaration

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

Acknowledgements

Callum for teaching me word referencing

Table of Contents

[1 Introduction 7](#_Toc38297114)

[1.1 Sudoku 7](#_Toc38297115)

[1.2 Motivation 8](#_Toc38297116)

[1.3 Aim 8](#_Toc38297117)

[1.4 Objectives 8](#_Toc38297121)

[2 Background Research 9](#_Toc38297122)

[2.1 Sudoku Puzzles 9](#_Toc38297123)

[2.2 Puzzle Generation 9](#_Toc38297124)

[2.2.1 Pre-generated Puzzles 10](#_Toc38297125)

[2.2.2 Generating Puzzles 10](#_Toc38297126)

[2.3 Algorithms 12](#_Toc38297127)

[2.3.1 Backtracking 12](#_Toc38297128)

[2.3.2 Hill Climb 12](#_Toc38297129)

[2.3.3 Simulated Annealing 13](#_Toc38297130)

[2.3.4 Genetic Algorithm 14](#_Toc38297131)

[2.4 Existing Systems 15](#_Toc38297132)

[2.5 Implementation Technologies 16](#_Toc38297133)

[2.5.1 Python 16](#_Toc38297134)

[2.5.2 Visual Studio Code 16](#_Toc38297135)

[2.5.3 NumPy / Matplotlib 16](#_Toc38297136)

[2.5.4 Tkinter 17](#_Toc38297137)

[3 System Design 18](#_Toc38297138)

[3.1 Requirements 18](#_Toc38297139)

[3.2 Methodology 18](#_Toc38297140)

[3.3 High Level Design 19](#_Toc38297141)

[3.4 Design of Graphical User Interface 20](#_Toc38297142)

[3.5 Design of Test Bed 22](#_Toc38297143)

[3.6 Design of Puzzle Generation 23](#_Toc38297144)

[3.7 Design of Algorithms 23](#_Toc38297145)

[3.7.1 Backtracking 24](#_Toc38297146)

[3.7.2 Hill Climb 25](#_Toc38297147)

[3.7.3 Simulated Annealing 27](#_Toc38297148)

[3.7.4 Genetic Algorithm 28](#_Toc38297149)

[4 Implementation 30](#_Toc38297150)

[4.1 Sudoku Generation 31](#_Toc38297151)

[4.1.1 Version 1 31](#_Toc38297152)

[4.1.2 Version 2 31](#_Toc38297153)

[4.2 Algorithms 34](#_Toc38297154)

[4.2.1 Backtracking 34](#_Toc38297155)

[4.2.2 Hill Climb 36](#_Toc38297156)

[4.2.3 Simulated Annealing 38](#_Toc38297157)

[4.2.4 Genetic Algorithm 40](#_Toc38297158)

[4.3 Test Bed 42](#_Toc38297159)

[4.3.1 Puzzle and Algorithm Interaction 42](#_Toc38297160)

[4.3.2 GUI and Analysis 43](#_Toc38297161)

[5 Testing 45](#_Toc38297162)

[5.1 Unit Testing 45](#_Toc38297163)

[5.2 Algorithm Testing 45](#_Toc38297164)

[5.3 Sudoku Generation Testing 48](#_Toc38297165)

[5.4 Test Bed Testing 48](#_Toc38297166)

[6 Evaluation 52](#_Toc38297167)

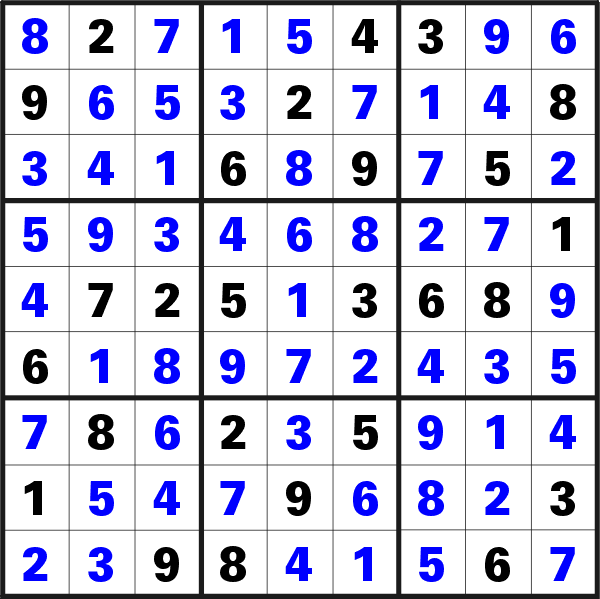
[7 Conclusion 53](#_Toc38297168)

[8 References 54](#_Toc38297169)

[9 Table of Figures 57](#_Toc38297170)

# Introduction

## Sudoku

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

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

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

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

## Motivation

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

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

## Aim

To develop a test bed that allows for the comparison of three main sudoku solving algorithms at a range of difficulty of puzzle.



## Objectives

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

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

# Background Research

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

## Sudoku Puzzles

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

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

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

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

## Puzzle Generation

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

### Pre-generated Puzzles

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

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

### Generating Puzzles

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

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

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

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

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

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

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

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

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

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

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

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

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

## Algorithms

The algorithms most commonly looked at are the stochastic algorithms which are all the ones being compared apart from backtracking. Simulated Annealing, Genetic Algorithm and Hill Climb are all similar in design but have a different take on the stochastic approach.

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

### Backtracking

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

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

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

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

### Hill Climb

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

There are three steps in generic Hill Climb:

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

This can be adapted to allow for sudoku to be solved using this algorithm easily with only one problem.

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

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

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

### Simulated Annealing

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

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

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

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

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

### Genetic Algorithm

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

There are 3 principle stages of a GA:

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

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

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

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

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

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

## Existing Systems

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

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

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

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

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

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

## Implementation Technologies

### Python

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

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

### Visual Studio Code

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

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

### NumPy / Matplotlib

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

This is useful as it allows for array manipulation and since the way a Sudoku puzzle is stored in a computer for an algorithm to solve it, is in an array. To make use of this efficient data structure for Sudoku solving, the use of NumPy greatly increases the way these algorithms can solve puzzles.

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

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

### Tkinter

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

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

# System Design

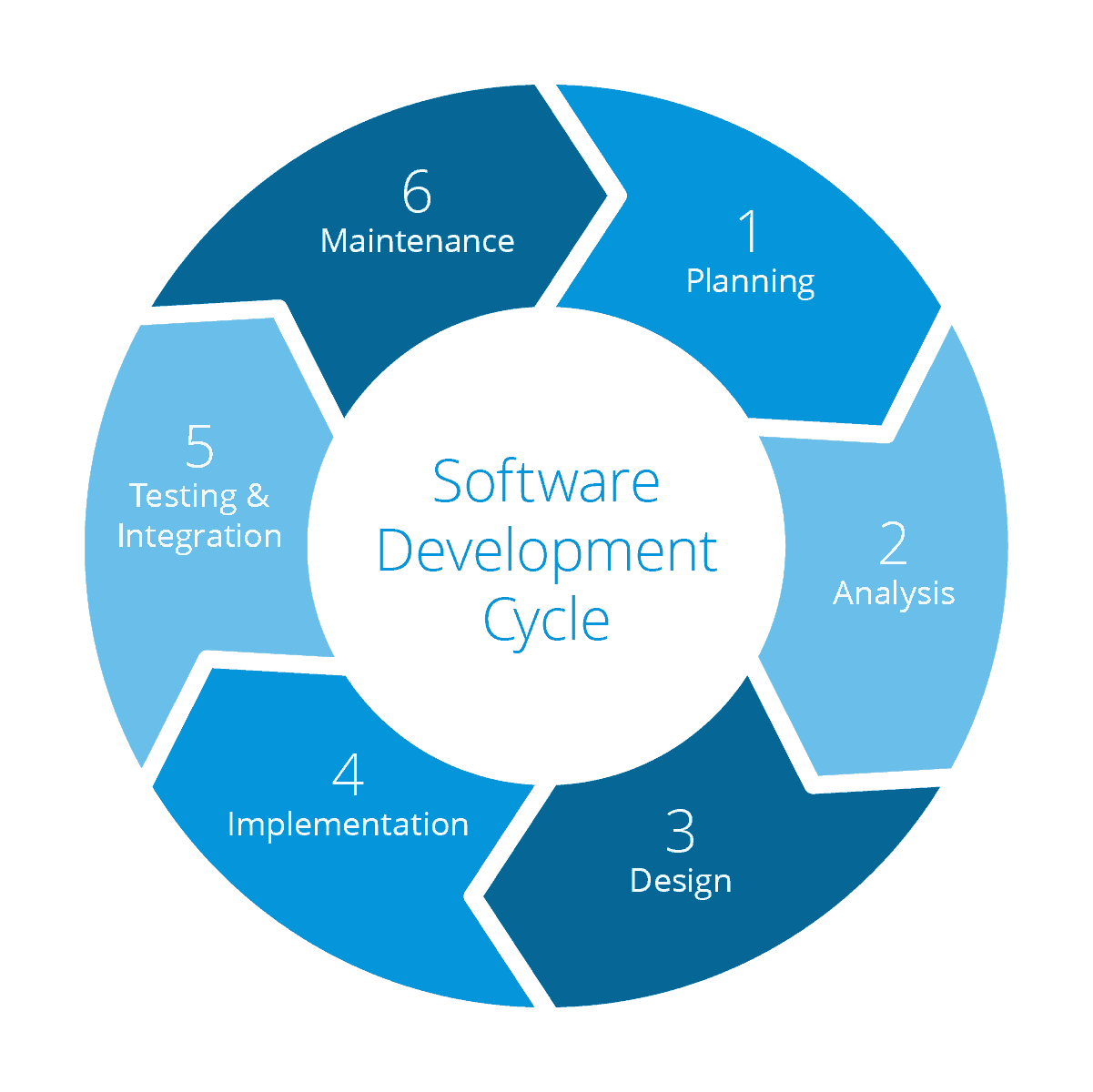
This chapter describes the requirements of the system in order to design a solution that solves the problem. It also shows the design process that will be used for the creation of the tool in the next chapter.

## Requirements

The requirements for the project are listed below:

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

## 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.

https://images.mendix.com/wp-content/uploads/Artboard-1@2x-701x700.png

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

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

## High Level Design

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

Figure 3.1 High Level Design of System

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

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

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

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

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

## Design of Graphical User Interface

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

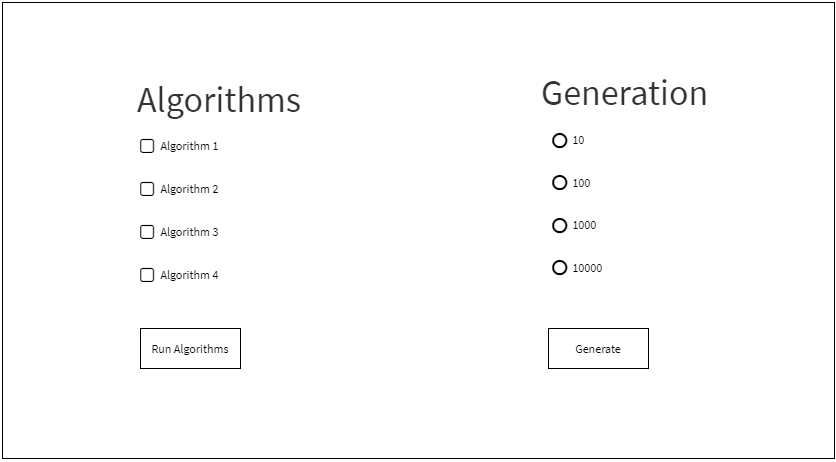
The user interface will consist of a main page with all the necessary inputs for the user to make. It will consist of 2 main aspects, with one side allowing the user to pick the algorithms that are going to be compared and the other side letting the user pick the number of sudoku boards to be generated.

Figure 3.2 GUI design for System

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

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

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

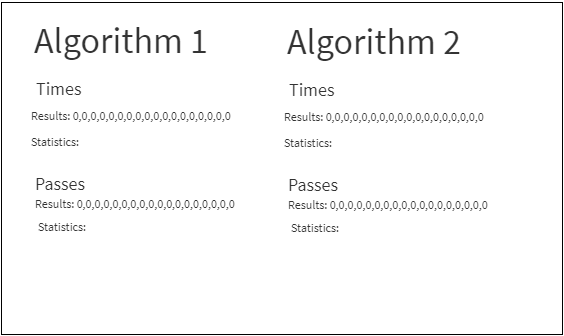


Figure 3.3 Mock of output GUI

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

## Design of Test Bed

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

For puzzle generation, as seen above, the user will select the number of puzzles that are to be created. This is then passed to the test bed that can then call the puzzle generator to produce the necessary test data for the system.

When the algorithms are being run, the test bed’s job is to allow for a smooth integration of the test data with the algorithms. The test bed will receive a list of algorithms to be run from the user, allowing one algorithm at a time to be executed - using the test data - through the test bed. This will return the time taken and the passes through the algorithm that can then be taken by the test bed and stored until all the algorithms are executed.

The final functionality of the test bed is the analysis of the data which will come after all algorithms have been executed and their results returned. This raw data can then be aggregated and analysed before it is sent to the user interface for the user to view. This is also a time where more in depth analysis can be done like graphing the outputs or more detailed statistics.

## Design of Puzzle Generation

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

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

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

Puzzle File

Puzzle Generator

Test Bed

Test Bed

Figure 3.4 Design of interaction between test bed and puzzle generation

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

## Design of Algorithms

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

### Backtracking

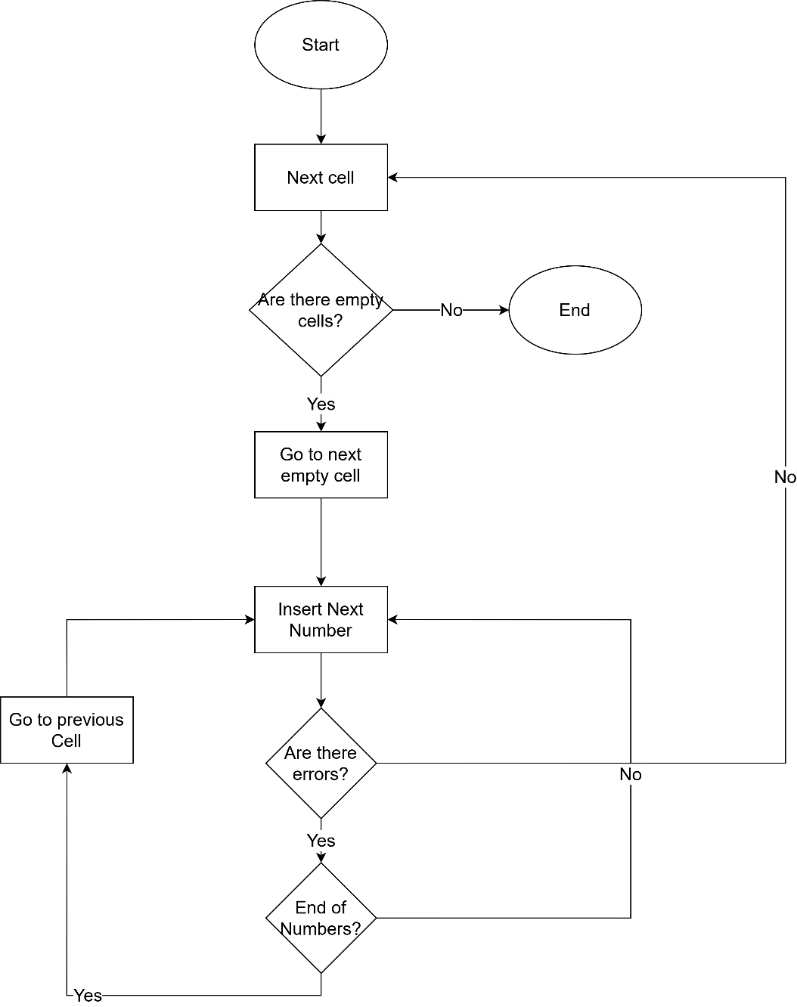
After backtracking is passed a puzzle to solve, the algorithm starts at the top-right corner of the puzzle and loops through each empty cell making sure the numbers it adds do not cause errors.

Figure 3.5 Flow chart of Backtracking

It first checks if there are any empty cells left in the board, this is the exit condition of the algorithm since, if this satisfied then the puzzle is either solved or it has no solution. Then it moves to the empty cell and tries the first number in the list of valid numbers which are 1 to 9.

If the inserted number causes an error then it checks if the number inserted is the last one e.g. 9. If it is the last number then it loops back to the previous cell and inserts the next number, looking for errors. If it is not the last number, then the algorithm tries the next number in the list on the same cell as before, again, looking for any errors that this causes to the puzzle.

If a number is found for a cell that does not cause any errors, then the algorithm iterates and as long as there is another empty cell – meaning the puzzle hasn’t been solved yet – it continues.

### Hill Climb

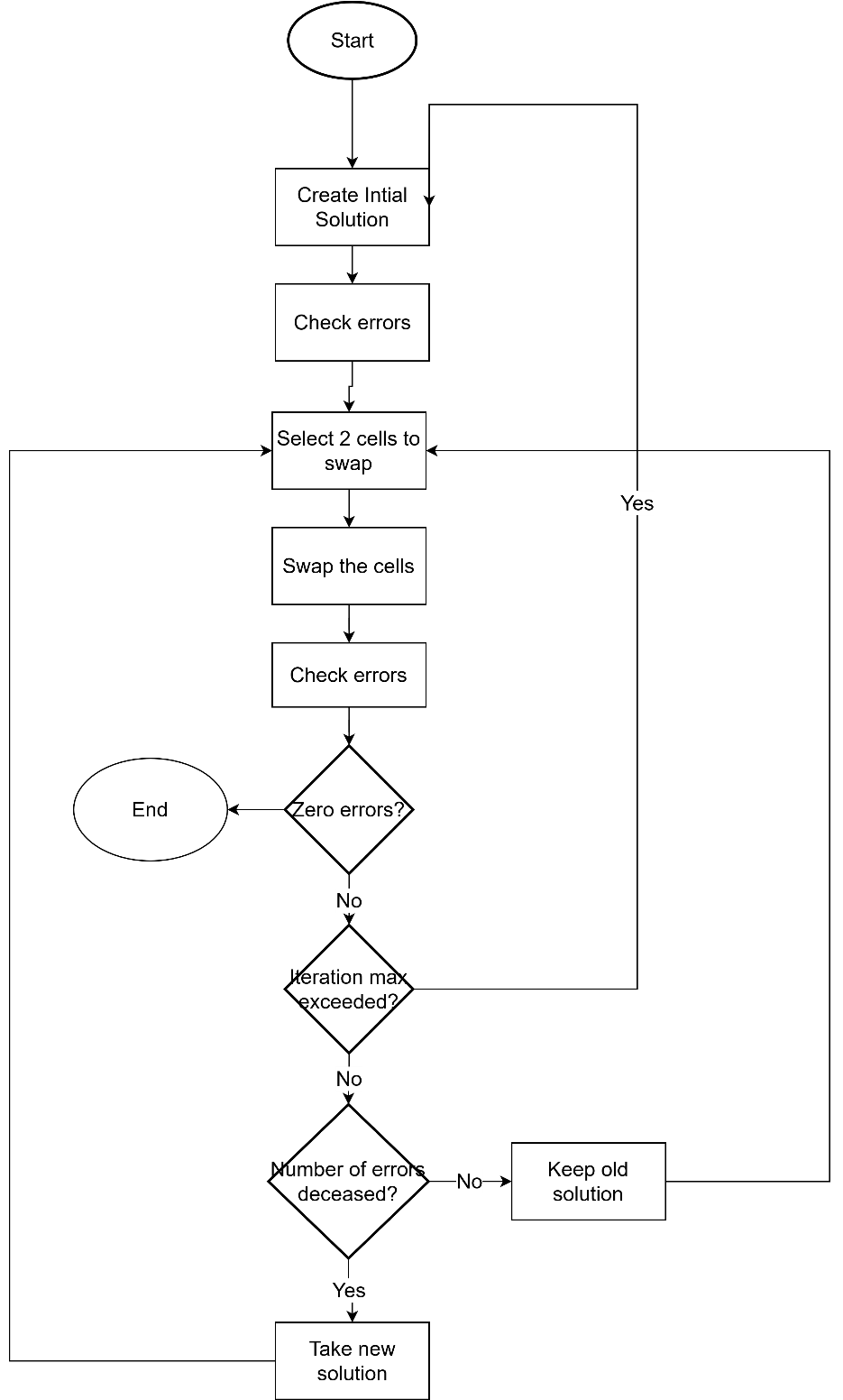
Hill Climb involves iterating through different versions of the puzzle, only moving to a new potential solution which it is closer to the real actual solution.

Figure 3.6 Flow chart of Hill Climb

Hill climb begins by creating an initial solution that meets 2 of the 3 constraints of a solved Sudoku puzzle, the number of errors are checked before any changes are made. Then, 2 non-fixed numbers in the same row, column or 3x3 box - dependent on which constraint was not satisfied in the original initialisation - are selected.

These 2 cells are then swapped and the number of errors are checked again. If the number of errors is 0 then the puzzle is solved and the program ends, if the errors are more than 0 then the number of iterations made through the swap part of the algorithm Is checked to see if the maximum has been reached. If it has then the algorithm loops up and another initial solution is generated.

When the iteration max has not been exceeded, the number of errors before and after the swap is compared. If the number of errors decreases between the old and new solutions, the new solution is taken into the next iteration of the algorithm, if it does not change or gets worse than the old solution is kept, and the new solution discarded.

### Simulated Annealing

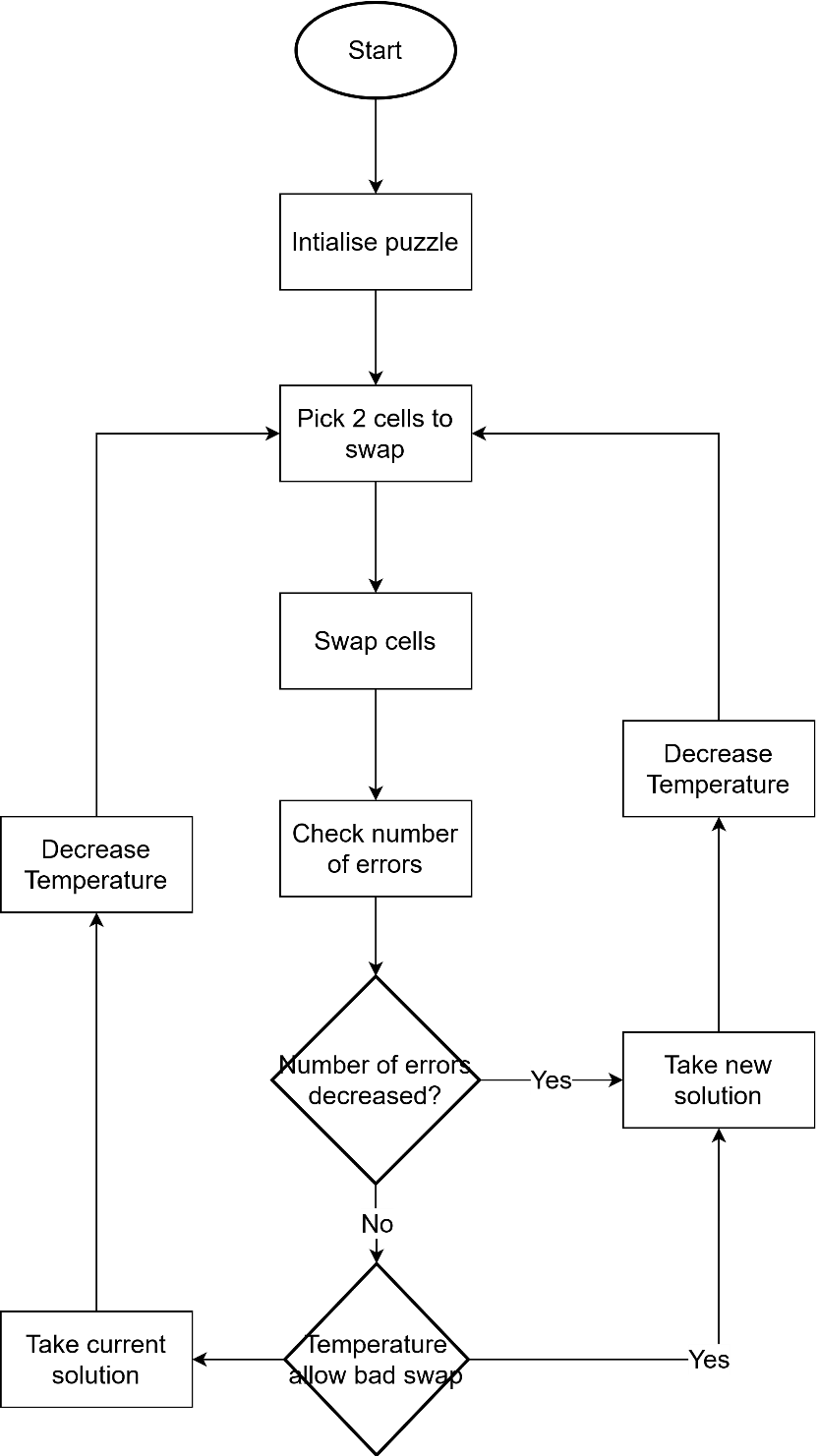
Simulated Annealing is an algorithm that is an optimization of Hill Climb but instead of only moving to a better solution when the error is lower, it also moves to worse solutions when the temperature is high.

Figure 3.7 Flow chart of Simulated Annealing

The algorithm starts by creating an initial implementation in the same way as Hill Climb – meeting only 2 of the 3 constraints of a valid Sudoku board. It then picks to cells to swap then swaps then and checks the errors, which is the same as Hill Climb. If the number of errors has decreased, then the new solution is taken but if it has not then the algorithm moves to the new addition on top of Hill Climb.

The temperature is a measurement of how far through the algorithm has progressed. As the temperature decreases the algorithm should be closer to a solution, meaning that if the temperature is lower than less bad swaps should be allowed to be made. After the number of errors has not decreased, the current temperature of the algorithm is checked and based on it, there is a chance between 0 and 1 that the worse solution will be taken. If the temperature is higher than the chance is higher and if the temperature is low, then the chance is lower.

Then, the temperature is decreased not dependent on which solution is being taken – the old one or the new one, allowing for the chance for a worse solution to be taken to be less.

### Genetic Algorithm

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

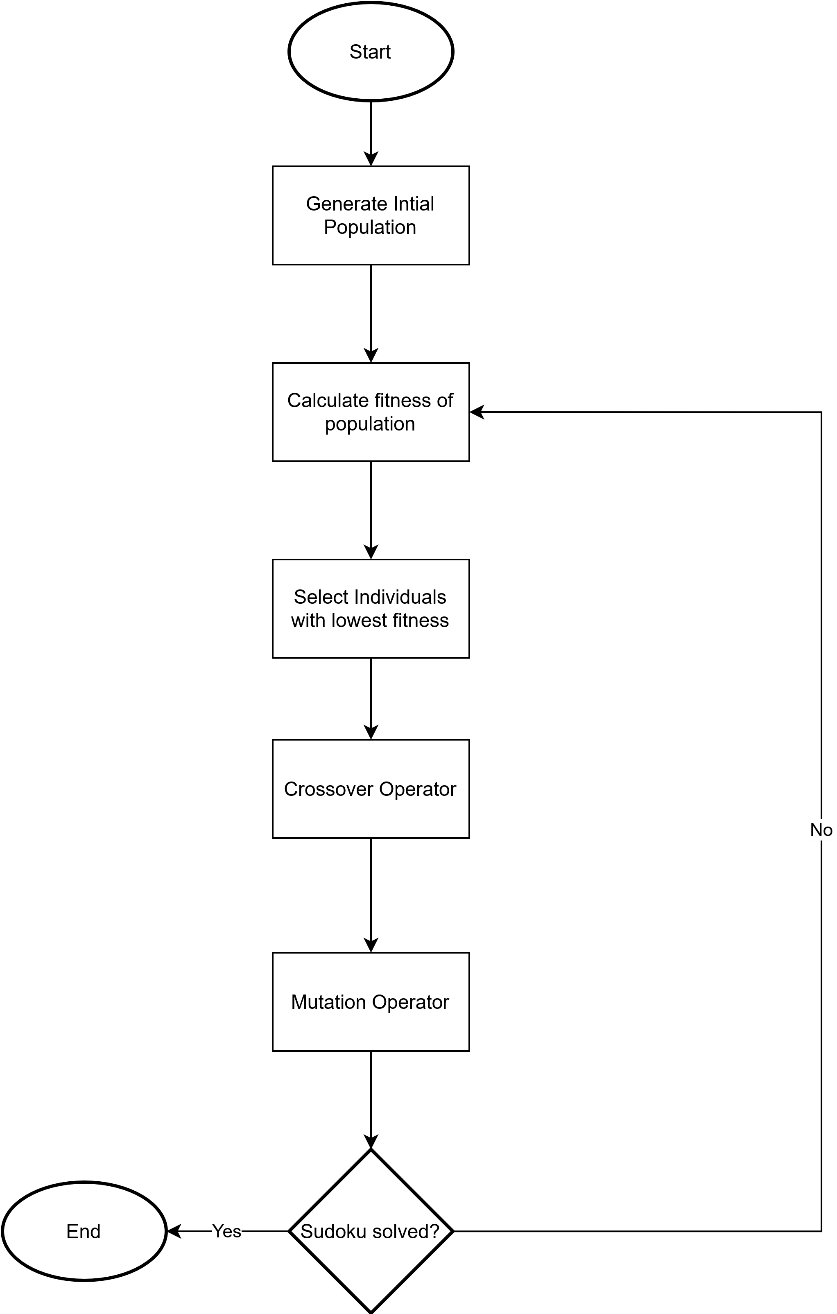


Figure 3.8 Flow chart of Genetic Algorithm

The algorithm starts by generating a population of potential solutions to the Sudoku puzzle, for each potential solution in the population, the fitness (how many errors are in the it) is calculated. The fitness of each one is checked, and each individual is labelled with their fitness.

Next individuals are selected using a selection rate that determines whether the fitter or weaker of 2 individuals should be taken, allowing for a wider population and not diverging on a solution too early.

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

# Implementation

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

## Sudoku Generation

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

### Version 1

Due to the iterative way that the project had been designed, this was a simplistic design that used a single class with most of the work done manually. Using [36] each refresh gives the user a new puzzle which can then be copied into a text file with the empty cells replaced with zeros. This Fetch class, when called can then get a line from the text file and return it to be tested on an algorithm.

### Version 2

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

The algorithm first starts by solving an empty 9x9 grid using one of the algorithms implemented already. Backtracking, explained later in the chapter, worked the best for this due to being able to set the randomness of it by shuffling the order that it tries to insert the numbers each time the function is called, meaning the completed board is completely different every time the algorithm is run.

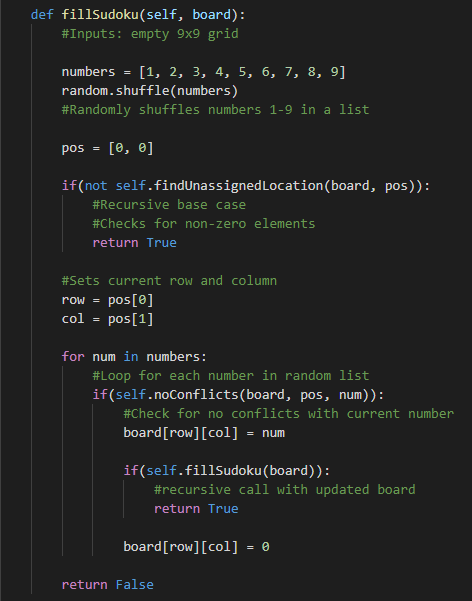


Figure 4.1 Code snippet for filling Sudoku board

Once the board is solved, an empty cell is selected by the algorithm, making sure it isn’t already zero. The number in the cell is then saved before being set to zero, 2 copies of the incomplete puzzle are then made, and both are sent me be solved by Backtracking again. This is because a puzzle might have multiple solutions and this makes it a bad puzzle, therefore using Backtracking we can check easily for multiple solutions. If, when solving, the numbers 1-9 are checked for each cell in ascending and descending order then, when there is only one solution, they will meet in the middle and return the same solution as they start at opposite ends of the solution space and brute force their way through every possibility.

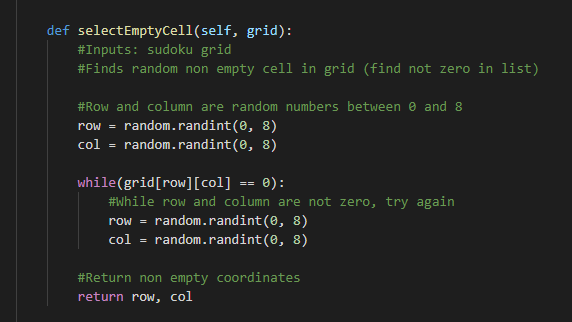


Figure 4.2 Code snippet for selecting empty cell

Numbers are removed from the board one at a time by looping through the empty cell selection and solving of the almost completed puzzles until there are multiple solutions to the puzzle. Then the number is put back into the board and an “attempts” variable is decreased by 1. This is a counter of the number of attempts have been made to the board to remove a number that have resulted in multiple solutions being returned. Once this becomes 0 then the puzzle is added to a 2d array containing all the puzzles to be written to the text file. The main function loops for the number of puzzles selected to be generated and then each puzzle is written to the file on a separate line before the algorithm terminates.

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

## Algorithms

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

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

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

### Backtracking

Backtracking is a recursive algorithm that calls itself for every new cell it attempts to find the correct value for, meaning that if there are no more empty cells then no more recursive calls need to be made and the puzzle is solved. There is a function for called findingUnassignedLocation that iterates through ever cell in the puzzle looking for the next empty one, working as a base case for the recursion if it cannot find an empty cell.

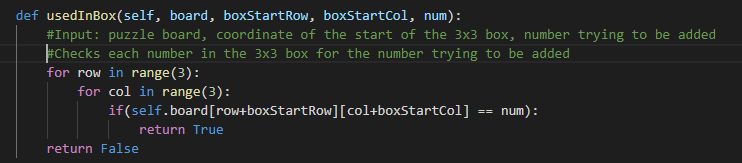
Each cell is then iterated through the numbers 1-9, each time checking if there are conflicts with putting that number in that specific cell. This is done by checking the row, column and 3x3 sub-box that the cell belongs to. The row and column checks are simple due to the puzzle being stored in a Numpy array and therefore being easy to iterate through. The 3x3 box is done by finding the top right corner of the box that the cell is in using modulo and then iterating through 3 cells along and 3 cells down to check the whole box.

Figure 4.3 Code snippet for getting sub-box of cell

Figure 4.4 Code snippet for iterating through sub-box

If there is a conflict, then the loop iterates again, the next number is tried for conflicts. If there aren’t any conflicts then the number is put into the board and the function is called again with the new board, meaning it will move to the next empty cell.

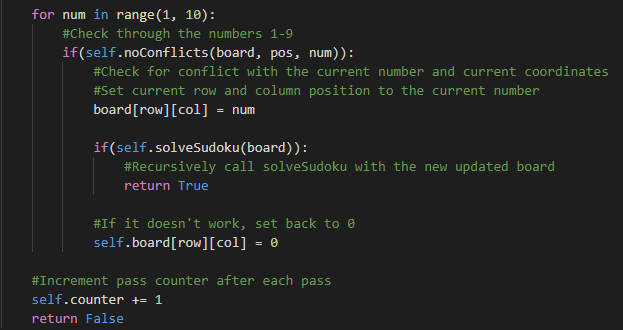
The backtracking portion of the algorithm comes, as seem above, when all the numbers 1-9 are exhausted in a single cell. When this happens and the loop finishes, the function returns false and since it has been recursively called the cell before continues its iterations through the numbers from where it left off until either it finds a new number to fit in the cell that doesn’t cause a conflict, or the loop runs out and it backtracks again to the previous cell.

Figure 4.5 Code snippet for backtracking fixed loop

The algorithm is run by calling the runAlgorithm function that instantiates the backtracking object and the board, it also initialises the pass counter. Then the algorithm is run while it is timed by the number of seconds it took. This, as well as the number of passes, is then returned to the Test Bed and used to compare.

One of the main problems with this algorithm came with the recursion as this is a very simplistic concept to understand but the implementation is slightly harder. Making sure there is a base case and identifying the condition that causes the algorithm to identify that the algorithm has solved the puzzle was a challenge.

### Hill Climb

Hill Climb involves moving towards the correct solution each time like climbing a hill to reach the top. The first step for the algorithm is the initialisation of the solution so that it meets 1 of the 3 constraints for Sudoku. In this algorithm the most efficient way was to make every 3x3 sub-box have the numbers 1-9 by getting each box one at a time and creating a list of the numbers already predetermined to be in that box. These are then subtracted from all the numbers 1-9 to give a list that shows the numbers that are not included in the box. These are then shuffled and inserted into all the empty cells in the box.

For solving the algorithm, there are 3 functions: the first is for detecting a solving sudoku and for running the other functions; the second is for calculating the energy (the number of errors in the potential solution); and the third is for climbing to new potential solutions.

The first function is simple, it initialises the solution then tries to climb before checking if the result of climb is the correct solution. If it is not, then it iterates again re-initialising the solution to try again. Since this algorithm can run indefinitely there is an added statement to check if the algorithm has tried 30,000 iterations of climbing, this avoids an infinite loop that could potentially happen as the algorithm will exit if it has reached that maximum.

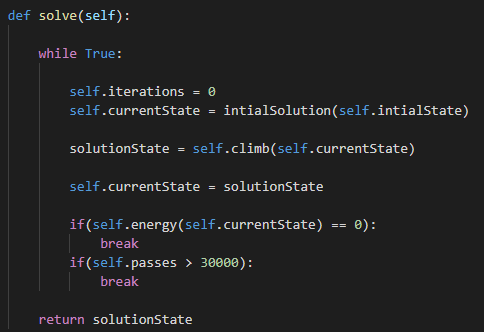


Figure 4.6 Hill Climb solve code snippet

Figure 4.7 Code snippet of solve function in Hill Climb

The second and third functions are linked together as to climb the energy calculations are required. To climb 2 non-fixed cells from one of the 9 3x3 boxes in the puzzle are selected and then swapped, allowing the box to still contain the numbers 1-9 in it. The energy of the old solution and the new solution are calculated by calling the energy function and then compared again a set of statements. If the maximum number of iterations has been reached, which is set to 200, or the number of errors is 0 then the puzzle is returned, and it handled by the first function.

The second part of the statement are recursive calls to the climb function using either the new solution or the old solution dependent on whether the energy of the new state is more or less than the old state. The number of iterations are also increased to allow a base case for the recursion to exit.

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

### Simulated Annealing

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

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

The climb class is very similar to the climb function in Hill Climb as it selects 2 random cells within a random box that are non-fixed and swaps them, almost identical to Hill Climb but without moving to the new solution.

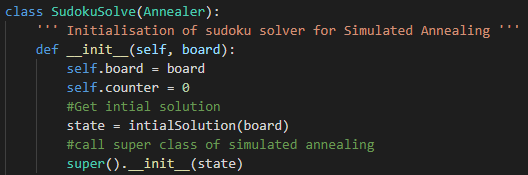
The second is energy which does exactly the same thing has Hill Climb except instead of taking an input parameter of the puzzle does not take a parameter, as the puzzle is stored and changed in the class object itself.

Figure 4.8 code snippet of intialisation of SudokuSolver class for Simulated Annealing

Figure 4.9 Code snippet of initialisation of SudokuSolver class for Simulated Annealing

After these have been implemented, in the initialisation of the object, the superclass is initialised with the intialSolution (the populated board) and given the set of parameters that are needed for Annealing to occur, the algorithm runs.

These parameters define 4 things:

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

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

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

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

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

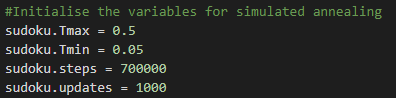


Figure 4.10 Code snippet of Simulated Annealing parameters

Figure 4.11 Code Snippet of Simulated Annealing parameters

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

### Genetic Algorithm

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

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

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

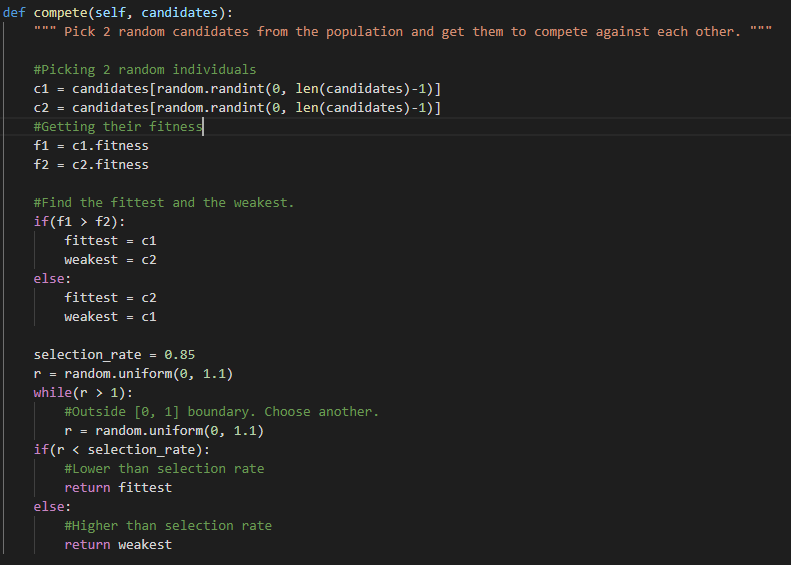
A number of processes are done to the population when the new population is being created. The first is in the tournament class that takes 2 individuals and using a selection rate either choses the fitter or weaker individual to ensure that the population as a wide solution space and doesn’t converge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

Figure 4.12 Code snippet of compete function in Genetic Algorithm

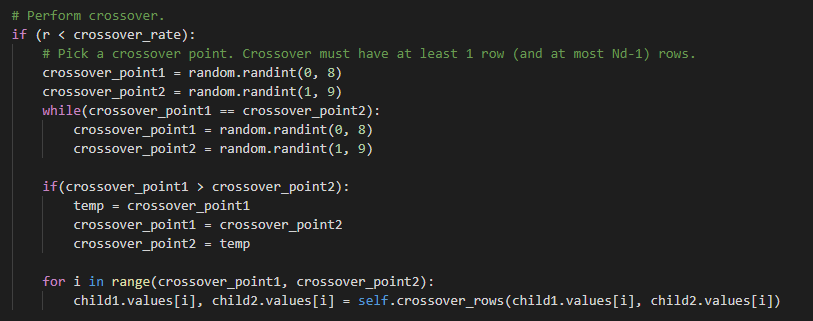
The algorithm uses the same method of initialisation of puzzles as the previous algorithms except it uses the row instead of the sub-box constraint for always keeping correct as it makes crossover easier to accomplish. Using this, crossover takes 2 parent individuals from the population and selects 2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

Figure 4.13 Code snippet of crossover in Genetic Algorithm

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

This continues as long as an individual does not contain the fitness 0 and therefore is a solved puzzle or the number of iterations has reached the maximum and therefore results in a timeout.

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

## Test Bed

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

### Puzzle and Algorithm Interaction

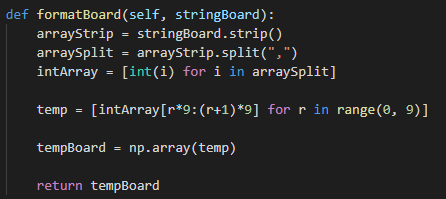
Puzzle Generation, as we saw previously in the chapter, puts the puzzles it generates into a text file that can then be read from, this is done here. First the file is opened, and a specific line is taken based on a parameter. This board is then formatted as it is stored in a one-dimensional list in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

Figure 4.14 Code snippet of board formatting in Test Bed

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

### GUI and Analysis

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

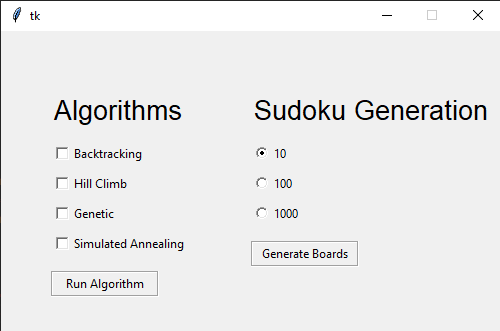
The GUI was implemented as per the design in the previous chapter due to it being very intuitive.

Figure 4.15 GUI for algorithm selection and puzzle generation

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

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

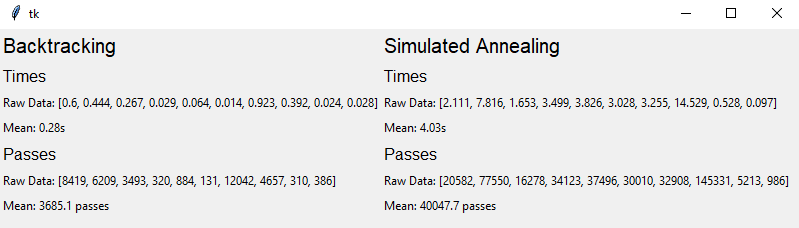
The results from the algorithms being run are displayed in 2 different ways both with graphics.

Figure 4.16 GUI for result display

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

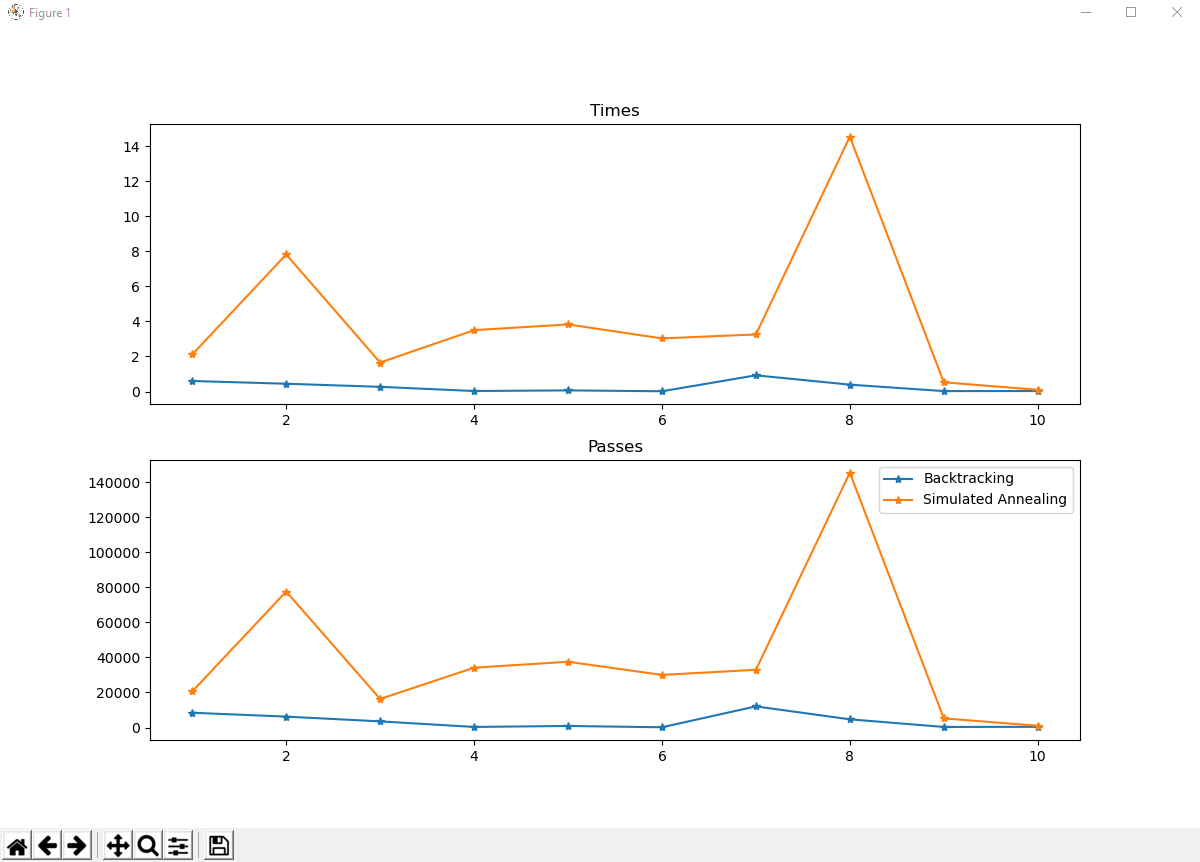
The second method of displaying results is through a graph.

Figure 4.17 GUI graph of results

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

# Testing

Testing of the system is important as it allows us to make sure that the implementation that has been developed is working as intended and giving the correct and expected result. Due to the implementation being done in iterations, after each phase of implementation, testing can be done to make sure each new part is functioning. This works well with the solving algorithms and Sudoku generation as they are implemented individually, and each section can be tested before moving on to the next one. The test bed can be tested only after all other sections are implemented as the functionality requires the use of the other sections.

Testing was done using a number of different methods, for the sections with clear components unit testing was done with system testing done on the overall tool after implementation is complete.

## Unit Testing

The unit testing involves testing individual components of the software being tested. The purpose of them is to make sure each unit of the software is working, with a unit being the smallest testable component of the system, and this should mean the overall software should work. A unit test should contain a few inputs with a single output that can then be tested for correctness.

Unit testing in Python can be done with a number of different libraries but the easiest is unittest which is integrated into Python and therefore can just be imported instead of having to be downloaded. Then unittest.TestCase can be inherited by a class with tests in it and when the main function is run, all tests starting with the “test” keyword will then automatically be run. Using assert statements to define whether the test passes or fails with a console output of what caused the tests to fail.

Unit testing, for this system, is done on the algorithms as well as the test bed as the components can be checked. Tests are written after each section is implemented making sure to not retest components that have already been previously tested.

## Algorithm Testing

Algorithms were tested as they were implemented using unit testing for individual functions in each algorithm and overall algorithm testing. Unit testing is not able to fully test the algorithms due to the randomness involved in how the puzzles are solved and therefore it is hard to test this on a specific scenario and get the same result every time which is how unit tests are conducted.

There is also a large amount of overlap between the algorithms in some of the functions being used, e.g. the initialisation of the puzzle, that only needs to be tested once throughout the whole implementation of all the algorithms.

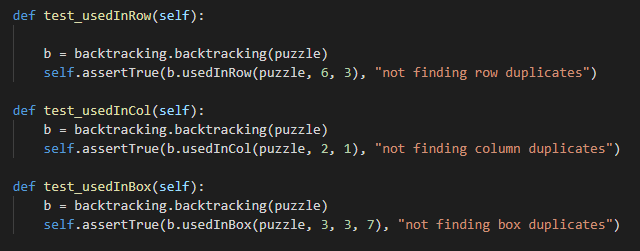
Backtracking can be unit tested completely due to it having deterministic results for the algorithm in a specific puzzle due to it iterating through each cell in order. Each function used to check each of the 3 constraints for a valid puzzle can be validated by manipulating the input to break the constraint and check that the test detects the error.

Figure 5.1 Code snippet of constraint checks for backtracking

Both Hill Climb, Simulated Annealing and Genetic are all stochastic algorithms and therefore they all initialise the puzzle before beginning working towards the solution. Since Simulated Annealing is an optimization of Hill Climb, they both use the same initialisation function and therefore by testing this which a unit test it can be guaranteed to work in both algorithms.

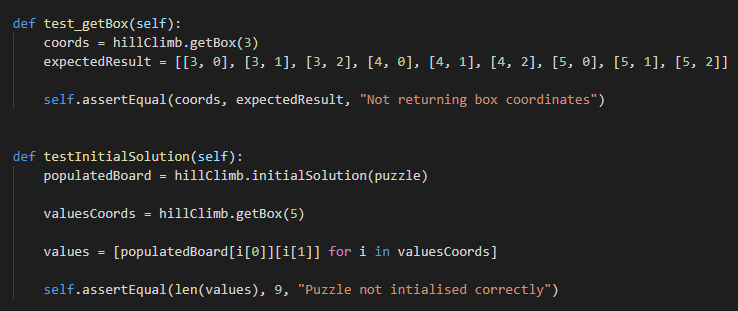


Figure 5.2 Code snippet for initial solution unit tests

Initial solution is tested by checking if a 3x3 box in the initialised puzzle has all the numbers 1-9 in it. This is the condition that is met when the initialisation happens that allows numbers to be swapped within sub-boxes. The selection of boxes also can be tested at the same time as the 2 functions are both used in Hill Climb and Simulated Annealing.

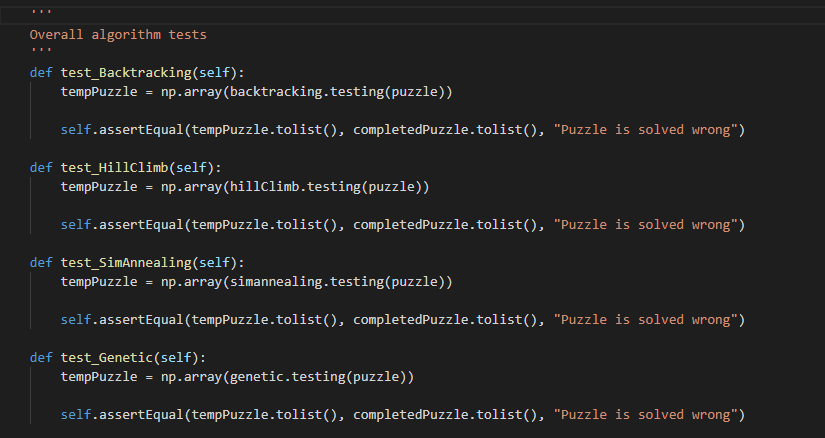
When testing the algorithm as a whole, a puzzle to solve and the completed version of the puzzle were defined. The same puzzle was used for each of the algorithms when testing to be able to check correctness of the output being given by the test.

Figure 5.3 Code snippet of algorithm unit tests

When the algorithms are being tested, functions are used from each algorithm that is a simplified version of the runAlgorithm as it doesn’t need to count the time taken or the iterations. It, instead, returns the puzzle that has been solved by the algorithm and this is compared to the actual completed puzzle that is saved in the test class. If they are equal then the test has passed, if not then the test has failed.



Figure 5.4 Output of algorithm unit tests

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

## Sudoku Generation Testing

Sudoku generation also has a large amount of randomness that inhibits the amount of unit testing that can be done on the component due to a different puzzle being created by it each time. However, there are 3 tests that have been written to cover the functionality of this part of the system.

There is 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 fillSudoku function is tested to make sure all the cells are filled with numbers but not that the numbers were entered to make a valid puzzle, this is because the component uses a lot of same functionality as Backtracking for creating the puzzle and therefore by testing the Backtracking constraint tests it will work. There is also a test for selecting a non-empty cell from the puzzle as it is important that this works for creating puzzles to be solved.

The last test checks that the creation of the incomplete puzzle is done correctly by passing the completed puzzle and comparing the result from the function against the incomplete version of the puzzle passed to the function.

Figure 5.5 Output of Sudoku Generation unit tests

## Test Bed Testing

The unit testing for the Test Bed only consists of retrieval of the board as all the rest of the functions are embedded within the GUI and therefore can be tested in the system testing.

The unit test opens the file, gets a specific line and formats it. It then calls the function and compares the results.

Figure 5.6 Output of Test Bed unit tests

Due to the Test Bed being the base of the whole system, all the other testing done on this component comes under system testing as it calls all the other parts of the system. System testing is black box which means there is no knowledge of the inner code design, meaning it makes the most sense to do this using the GUI to allow the GUI to be tested as well as the system as a whole.

Within the GUI there are the algorithms and the Sudoku generation that need to be tested. The radiobuttons are tested by clicking each one making sure the selected number changes and the board generation can be checked for each number of puzzles.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Number** | **Input** | **Expected Output** | **Actual Output** | **Pass/Fail** |
| 1 | Click “10” puzzle radio button | “10” puzzle button is selected | “10” puzzle button is selected | Pass |
| 2 | Click “100” puzzle radio button | “100” puzzle button is selected | “100” puzzle button is selected | Pass |
| 3 | Click “1000” puzzle radio button | “1000” puzzle button is selected | “1000” puzzle button is selected | Pass |
| 4 | 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 |
| 5 | 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 |
| 6 | 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 |

Table 5.1 Tests for Sudoku Generation GUI

Tests numbers 1-3 are GUI testing for the generation part of the system and 4-6 are testing the functionality of the Test Bed and if the passing of the correct number of puzzles to the puzzle generator is being done successfully. To prevent no button being selected, on launch of the system, the current number of puzzles is counted, and the corresponding radio button is automatically selected.

The algorithm part of the testing of Test Bed is done in the same way as the Sudoku generation with the GUI tests on the button as well as testing the calling of the algorithms with the correct boards. For testing the results produced by the algorithms, the time taken is very hard to test although outputs are given by the algorithms to the console so a rough estimate of the expected time taken can be used. For iterations, putting the counter increaser 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 logical common sense for the number it is producing.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Number** | **Input** | **Expected Output** | **Actual Output** | **Pass/Fail** |
| 12 | “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 |
| 13 | “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 |
| 14 | “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 |
| 15 | “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 5.2 Some tests for algorithm execution through GUI

Test number 13 is the Hill Climb test which passes even though the it fails the unit test and doesn’t return the solved puzzle. This is because the Hill Climb algorithm still produces a time and iteration result and therefore can display these returns even though the algorithm itself did not solve the puzzle. Test number 15, the Genetic algorithm test, fails as the algorithm produces an error and therefore crashes before being able to produce a result.

# Evaluation

## Results

## Project Requirements

# Conclusion

# 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] | “Sudoku,” [Online]. Available: https://sudoku.com/. [Accessed 12 April 2020]. |
| [37] | D. Marshall, “Steepest Ascent Hill Climbing,” [Online]. Available: http://users.cs.cf.ac.uk/Dave.Marshall/AI1/steep.html. [Accessed 1 April 2020]. |
| [38] | 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. |

# Table of Figures

[Figure 3.1 High Level Design of System 20](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc38296976)

[Figure 3.2 GUI design for System 21](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc38296977)

[Figure 3.3 Mock of output GUI 22](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc38296978)

[Figure 3.4 Design of interaction between test bed and puzzle generation 23](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc38296979)

[Figure 3.5 Flow chart of Backtracking 24](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc38296980)

[Figure 3.6 Flow chart of Hill Climb 26](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc38296981)

[Figure 3.7 Flow chart of Simulated Annealing 27](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc38296982)

[Figure 3.8 Flow chart of Genetic Algorithm 29](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc38296983)

[Figure 4.1 Code snippet for filling Sudoku board 32](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc38296984)

[Figure 4.2 Code snippet for selecting empty cell 33](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc38296985)

[Figure 4.3 Code snippet for getting sub-box of cell 34](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc38296986)

[Figure 4.4 Code snippet for iterating through sub-box 34](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc38296987)

[Figure 4.5 Code snippet for backtracking fixed loop 35](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc38296988)

[Figure 4.6 Hill Climb solve code snippet 36](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc38296989)

[Figure 4.7 Code snippet of solve function in Hill Climb 36](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc38296990)

[Figure 4.8 code snippet of intialisation of SudokuSolver class for Simulated Annealing 38](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc38296991)

[Figure 4.9 Code snippet of initialisation of SudokuSolver class for Simulated Annealing 38](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc38296992)

[Figure 4.10 Code snippet of Simulated Annealing parameters 39](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc38296993)

[Figure 4.11 Code Snippet of Simulated Annealing parameters 39](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc38296994)

[Figure 4.12 Code snippet of compete function in Genetic Algorithm 40](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc38296995)

[Figure 4.13 Code snippet of crossover in Genetic Algorithm 41](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc38296996)

[Figure 4.14 Code snippet of board formatting in Test Bed 42](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc38296997)

[Figure 4.15 GUI for algorithm selection and puzzle generation 43](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc38296998)

[Figure 4.16 GUI for result display 44](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc38296999)

[Figure 4.17 GUI graph of results 44](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc38297000)

[Figure 5.1 Code snippet of constraint checks for backtracking 46](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc38297001)

[Figure 5.2 Code snippet for initial solution unit tests 46](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc38297002)

[Figure 5.3 Code snippet of algorithm unit tests 47](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc38297003)

[Figure 5.4 Output of algorithm unit tests 47](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc38297004)

[Figure 5.5 Output of Sudoku Generation unit tests 48](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc38297005)

[Figure 5.6 Output of Test Bed unit tests 48](file:///C:\Users\Calum\Documents\GitHub\sudoku-disseration\documentation\Dissertation.docx#_Toc38297006)

Testing Done

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Number** | **Input** | **Expected Output** | **Actual Output** | **Pass/Fail** |
| 1 | Click “10” puzzle radio button | “10” puzzle button is selected | “10” puzzle button is selected | Pass |
| 2 | Click “100” puzzle radio button | “100” puzzle button is selected | “100” puzzle button is selected | Pass |
| 3 | Click “1000” puzzle radio button | “1000” puzzle button is selected | “1000” puzzle button is selected | Pass |
| 4 | 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 |
| 5 | 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 |
| 6 | 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 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Number** | **Input** | **Expected Output** | **Actual Output** | **Pass/Fail** |
| 7 | Click “Backtracking” check box | “Backtracking” check box selected | “Backtracking” check box selected | Pass |
| 8 | Click “Hill Climb” check box | “Hill Climb” check box selected | “Hill Climb” check box selected | Pass |
| 9 | Click “Simulated Annealing” check box | “Simulated Annealing” check box selected | “Simulated Annealing” check box selected | Pass |
| 10 | Click “Genetic” check box | “Genetic” check box selected | “Genetic” check box selected | Pass |
| 11 | Click “Run Algorithms” with no check boxes selected | Error message saying no algorithms were selected | Error message saying no algorithms were selected | Pass |
| 12 | “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 |
| 13 | “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 |
| 14 | “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 |
| 15 | “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 |
| 16 |  |  |  |  |
| 17 |  |  |  |  |
| 18 |  |  |  |  |