Comparison of Sudoku Solving Algorithms

Calum Harvey (Student ID: 170349985)

Computer Science

Supervisor: Dr Jason Steggles

Word Count: 5064

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](#_Toc36767757)

[1.1 Sudoku 7](#_Toc36767758)

[1.2 Motivation 8](#_Toc36767759)

[1.3 Aim 8](#_Toc36767760)

[1.4 Objectives 8](#_Toc36767761)

[2 Background Research 9](#_Toc36767762)

[2.1 Sudoku Puzzles 9](#_Toc36767763)

[2.2 Puzzle Generation 9](#_Toc36767764)

[2.2.1 Pre-generated Puzzles 10](#_Toc36767765)

[2.2.2 Generating Puzzles 10](#_Toc36767766)

[2.3 Algorithms 12](#_Toc36767767)

[2.3.1 Backtracking 12](#_Toc36767768)

[2.3.2 Hill Climb 12](#_Toc36767769)

[2.3.3 Simulated Annealing 13](#_Toc36767770)

[2.3.4 Genetic Algorithm 14](#_Toc36767771)

[2.4 Existing Systems 15](#_Toc36767772)

[2.5 Implementation Technologies 16](#_Toc36767773)

[2.5.2 Python 16](#_Toc36767774)

[2.5.1 Visual Studio Code 16](#_Toc36767775)

[2.5.3 NumPy / Matplotlib 16](#_Toc36767776)

[2.5.4 Tkinter 17](#_Toc36767777)

[3 System Design 18](#_Toc36767778)

[3.1 Requirements 18](#_Toc36767779)

[3.2 Methodology 18](#_Toc36767780)

[3.3 High Level Design 19](#_Toc36767781)

[3.4 Design of Graphical User Interface 20](#_Toc36767782)

[3.5 Design of Test Bed 21](#_Toc36767783)

[3.6 Design of Puzzle Generation 22](#_Toc36767784)

[3.7 Design of Algorithms 23](#_Toc36767785)

[4 Sudoku Generation 24](#_Toc36767786)

[4.1 Version 1 24](#_Toc36767787)

[4.2 Version 2 24](#_Toc36767788)

[5 Algorithms 24](#_Toc36767789)

[5.1 Backtracking 24](#_Toc36767790)

[5.2 Simulated Annealing 25](#_Toc36767791)

[5.3 Genetic Algorithm 25](#_Toc36767792)

[6 Test Bed 25](#_Toc36767793)

[7 Testing 26](#_Toc36767794)

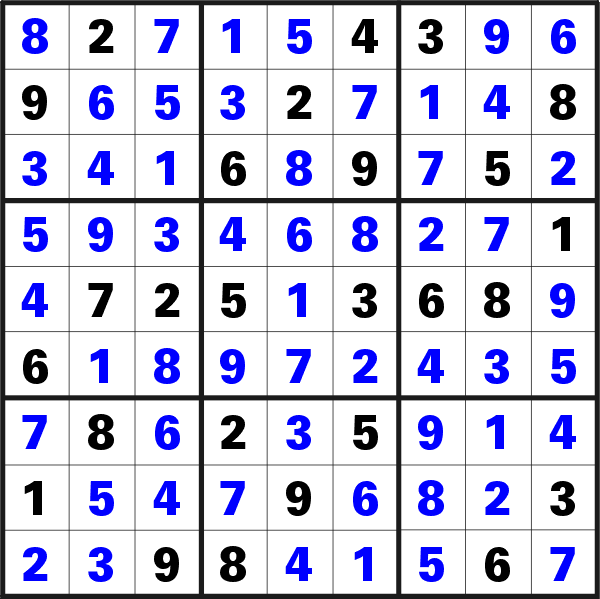
[8 Evaluation 27](#_Toc36767795)

[9 Conclusion 28](#_Toc36767796)

[References 29](#_Toc36767797)

# 1 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.

## 1.4 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

# 2 Background Research

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

## 2.1 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.

## 2.2 Puzzle Generation

Sudoku puzzle generation is important to this project as it allows for vast amounts of test data to be created and for the test data to be designed to 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.

### 2.2.1 Pre-generated Puzzles

The simplest way for an algorithm to be tested on a valid sudoku board is to take already generated puzzles from the internet or Newspaper and 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.

### 2.2.2 Generating Puzzles

There are two main advantages of generating puzzles as test data, the first is that you can control the difficulty of the puzzles that are 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.

## 2.3 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.

### 2.3.1 Backtracking

Brute-force backtracking is the most basic and least intelligent algorithm for sudoku solving. In the theme of all brute-force algorithm, it involved searching the whole solution space for 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.

### 2.3.2 Hill Climb

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

### 2.3.3 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.

### 2.3.4 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.

## 2.4 Existing Systems

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

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.

## 2.5 Implementation Technologies

### 2.5.2 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.

### 2.5.1 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.

### 2.5.3 NumPy / Matplotlib

NumPy [31] is a Python package for scientific computing that is based around using N-dimensional arrays and high level mathematical functions for operating on 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.

### 2.5.4 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.

# 3 System Design

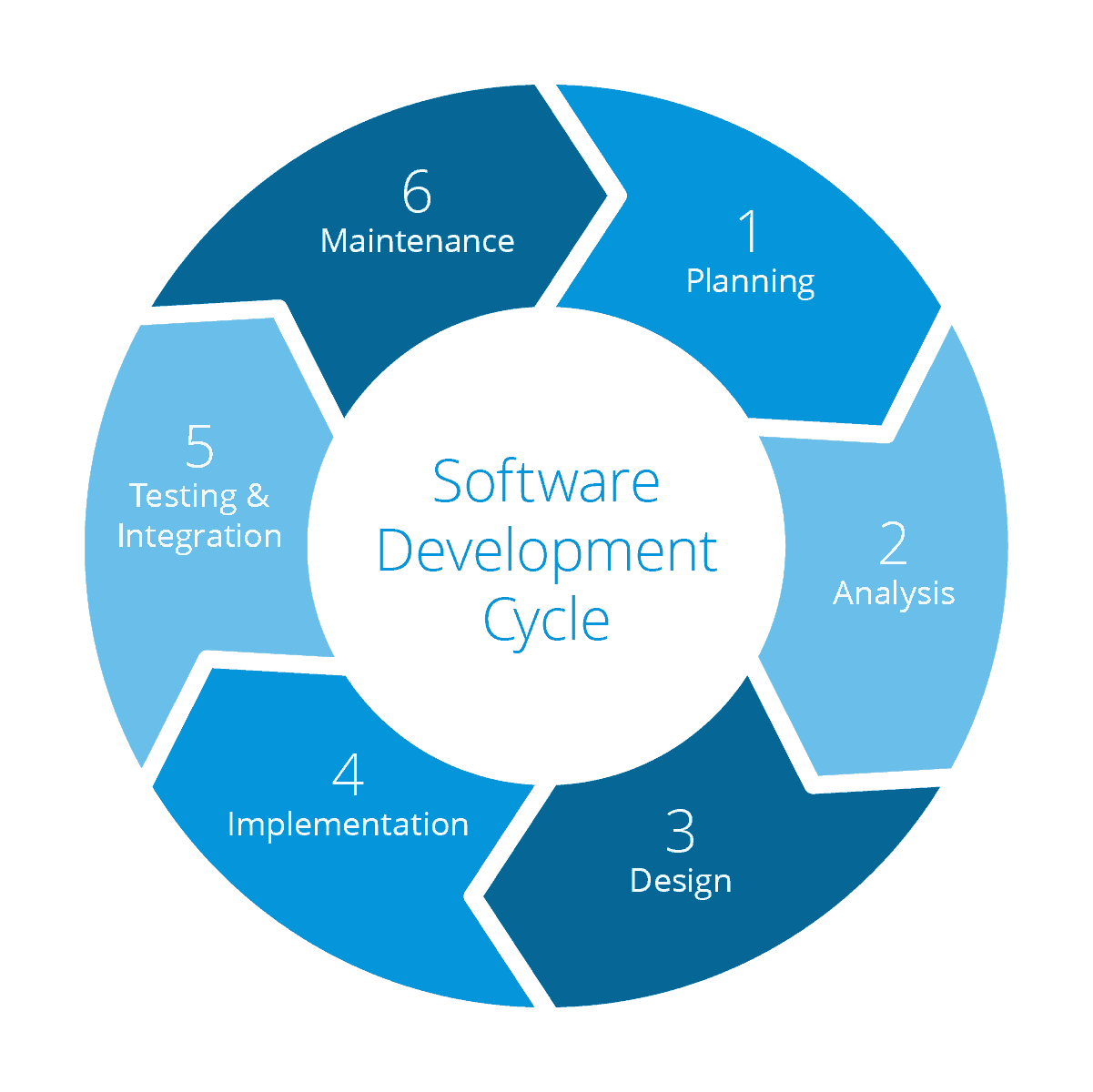
This chapter describes the requirements of the system in order to design a solution that solves the problem. It also shows the design process that will be used for the creation of the tool in the next chapter.

## 3.1 Requirements

The requirements for the project are listed below:

1. The tool should provide an implementation of multiple sudoku solving algorithms.
2. The test bed should allow puzzle generation and algorithm execution.
3. It should provide the ability to generate sudoku puzzles.
4. The user should be able to choose the number of puzzles to be generated.
5. It should allow for algorithms to be compared.
6. The algorithms that are to be compared should be able to be selected by the user.
7. 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

## 3.2 Methodology

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

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.

## 3.3 High Level Design

The first step for designing the system is to identify the main components that need to be implemented and how they 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.

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

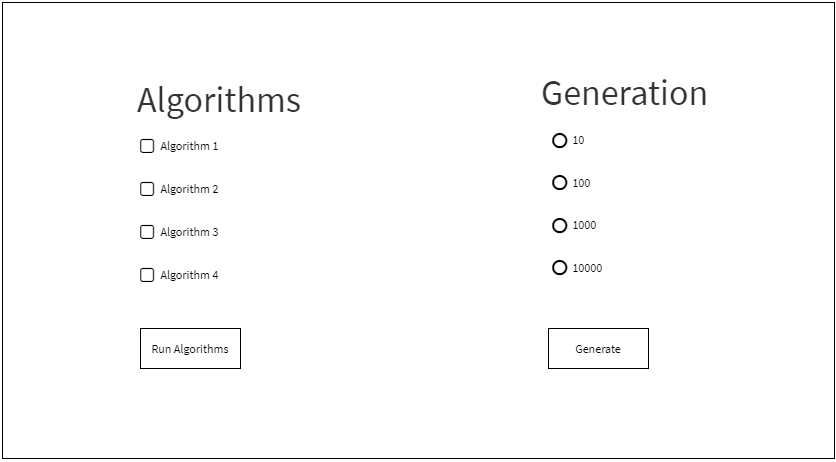
Puzzle Generation: The component that generates the puzzles based on the requests from the test bed.

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

## 3.4 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.

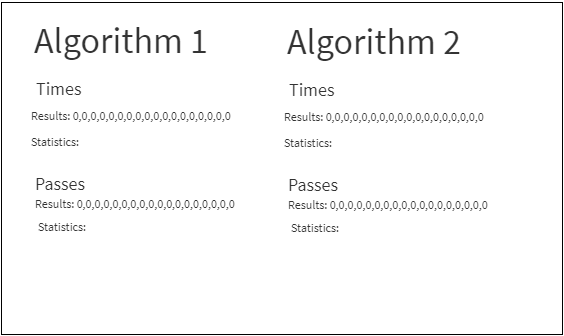
The user interface will consist of a main page with all the necessary inputs for the user to make. It will consist of 2 main aspects, with one side allowing the user to pick the algorithms that are going to be compared and the other side letting the user pick the number of sudoku boards to be generated.



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.



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.

## 3.5 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.

## 3.6 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

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.

## 3.7 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.

### 3.7.1 Backtracking

After backtracking is passed a puzzle to solve, the algorithm starts at the top-right corner of the puzzle and

# 4 Sudoku Generation

## 4.1 Version 1

## 4.2 Version 2

# 5 Algorithms

## 5.1 Backtracking

## 5.2 Simulated Annealing

## 5.3 Genetic Algorithm

# 6 Test Bed

# 7 Testing

# 8 Evaluation

# 9 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] | D. Marshall, “Steepest Ascent Hill Climbing,” [Online]. Available: http://users.cs.cf.ac.uk/Dave.Marshall/AI1/steep.html. [Accessed 1 April 2020]. |
| [36] | 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. |