Comparison of Sudoku Solving Algorithms

Calum Harvey (Student ID: 170349985)

Computer Science

Supervisor: Dr Jason Steggles

Word Count: 2836

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 6](#_Toc33797528)

[1.1 Sudoku 6](#_Toc33797529)

[1.2 Motivation 7](#_Toc33797530)

[1.3 Aim 7](#_Toc33797531)

[1.4 Objectives 7](#_Toc33797532)

[2 Background Research 8](#_Toc33797533)

[2.1 Puzzle Generation 8](#_Toc33797534)

[2.1.1 Pre-generated Puzzles 8](#_Toc33797535)

[2.2.2 Generating Puzzles 8](#_Toc33797536)

[2.2 Algorithms 10](#_Toc33797537)

[2.2.1 Backtracking 10](#_Toc33797538)

[2.2.2 Simulated Annealing 10](#_Toc33797539)

[2.2.3 Genetic Algorithm 10](#_Toc33797540)

[2.2.3 Hill Climb or Tabu Search 10](#_Toc33797541)

[2.3 Existing Systems 11](#_Toc33797542)

[3 Algorithms 14](#_Toc33797543)

[4 Sudoku Generation 14](#_Toc33797544)

[5 Test Bed 14](#_Toc33797545)

[6 Testing 15](#_Toc33797546)

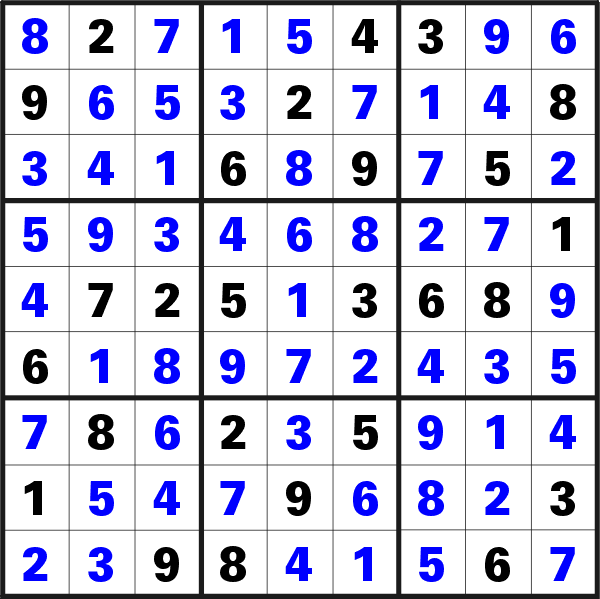
[7 Evaluation 16](#_Toc33797547)

[8 Conclusion 17](#_Toc33797548)

[References 18](#_Toc33797549)

# 1 Introduction

## Sudoku

Sudoku is a logic-based number placement puzzle game that has grown in popularity since it first appeared in French newspapers in the late 19th century. 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. 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.

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 its 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 explore backtracking, stochastic and constraint satisfaction problem sudoku solving algorithms and compare and contrast them using a range of difficulty of puzzles.

## 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.

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.

## 2.1 Puzzle Generation

### 2.1.1 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 [1]. 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 [2].

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 into 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 [3] 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:

* 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:

* 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 [3] 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 dependant 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 [4] 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.

## 2.2 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.

### 2.2.1 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.

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, 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.

### 2.2.2 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. 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 [5] 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 [5] 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 [6] 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.

### 2.2.3 Genetic Algorithm

Genetic algorithms (GA) [7] 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 [8] 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 [9] 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 [10] there are 4 constraints defined with an added constraint of the original numbers in the puzzle remain in their original position, something that [9] 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 [10] 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.

### 2.2.3 Hill Climb or Tabu Search

## 2.3 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 [6] 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. This, however, only considers stochastic algorithms but other types of algorithm may be more efficient or quicker.

This project [11] also contains a variety of stochastic algorithms that can be compared but this implementation takes are more of a generic approach to algorithm design and therefore is more of a test of what algorithms can successfully solve Sudoku puzzles.

This research [12] looks at backtracking, constraint programming and rule-based algorithms for comparison of their efficiency, however there is no implementation of any stochastic algorithms for comparing against the backtracking or constraint programming.

Here [13] 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.

# 3 Algorithms

# 4 Sudoku Generation

# 5 Test Bed

# 6 Testing

# 7 Evaluation

# 8 Conclusion

# References

|  |  |
| --- | --- |
| [1] | “qqWing,” [Online]. Available: https://qqwing.com/. [Accessed 25 February 2020]. |
| [2] | H.-f. Leung and C.-h. Lam, “Progressive Stochastic Search for Solving Constraint Satisfaction Problems,” in *Tools with Artificial Intelligence*, 2003. |
| [3] | T. Mantere and J. Kolijonen, “Solving, rating and generating Sudoku puzzles with GA,” in *Evolutionary Computation*, 2007. |
| [4] | M. Hunt, C. Pong and G. Tucker, “Difficulty-Driven Sudoku Puzzle,” *The UMAP Journal,* vol. 29, no. 3, pp. 343 - 362, 2008. |
| [5] | R. Lewis, “Metaheuristics can Solve Sudoku Puzzles.,” Edinburgh, 2007. |
| [6] | M. Perez and T. Marwala, “STOCHASTIC OPTIMIZATION APPROACHES FOR SOLVING SUDOKU,” Computing Research Repository, 2008. |
| [7] | D. Whitley, “A genetic algorithm tutorial,” *Stat&tics and Computing ,* vol. 4, pp. 65-85 , 1994. |
| [8] | 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. |
| [9] | 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]. |
| [10] | Q. D. Xiu and D. L. Yong, “A novel hybrid genetic algorithm for solving Sudoku,” *Optimization Letters,* vol. 7, p. 241–257, 2011. |
| [11] | “Solving Sudoku boards using stochastic methods and genetic algorithms,” 1 December 2018. [Online]. Available: https://github.com/sraaphorst/sudoku\_stochastic. [Accessed 5 March 2020]. |
| [12] | S. Ekne and K. Gylleus, “Analysis and comparison of solving algorithms for sudoku,” KTH ROYAL INSTITUTE OF TECHNOLOGY, Stockholm, 2015. |
| [13] | 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. |