APPLICATIONS OF SIMULATED ANNEALING TO GAMES

MARK QUIGLEY
Supervisor: Professor Sarp Akcay
20209931

ACM40960-Projects in Maths Modelling
Project for Master in Data & Computational Science

Abstract

In this paper, we will investigate simulated annealing through sudoku puzzles, explore various methods of picking proposed steps, and fit the initial temperature and cooling rate. We will also look at a failed method for using simulated annealing to solve a Rubik cube and compare with a successful method.

CONTENTS

THESIS CONTENT - CHAPTERS

1 INTORDUCTION

In this chapter we will run through the basics of sudoku puzzles, and explain how to use simulated annealing to solve a sudoku puzzle.

1.1 WHAT IS SUDOKU

A classic sudoku puzzle is a 9×9 grid, with numbers in some of the cells and others left empty an example can be seen below in Figure 1. There are three major structures to note, the rows, columns and nine 3×3 sub grids marked by the bold lines in the example. The aim of the game is to fill the blank cells with numbers 1 to 9 such that each number only appears once in each structure. Clearly stated, the rules are:

1. Each row has every number from 1 to 9 once and only once.
2. Each column has every number from 1 to 9 once and only once.
3. Each sub grid has every number from 1 to 9 once and only once.

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

Figure 1: Example order 3 sudoku puzzle

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

Figure 2: Example of a solved sudoku puzzle

It will be helpful to introduce some terminology here:

Definition 1.1.1. A **cell** is one of the 81 blocks that makes the puzzle.

Definition 1.1.2. A **fixed cell** is a cell with a given value. Somtimes also called a **hint**, though not here.

Definition 1.1.3. A **non fixed cell** is a cell without any assigned value at the start.

Row, **column** and **sub grid** follow the obvious definitions.

The above puzzle in Figures 1 & 2 is referred to as an order 3 puzzle, as there are 3^2 rows, columns and squares, and only numbers 1 to 3^2 are allowed. It should be clear with a few examples in Figures 3 & 4 how this can be generalised to order $n > 2$. There can be a puzzle of order 1, but there is only one.

For the order 4 puzzle, an adapted hexadecimal number scheme which includes G is used, and no 0. This is because in the code I have used 0's to indicate where non fixed cells are. For even higher-order puzzles, base ten numbers are typically used, though this was never a problem I faced in this project as I could not solve them. However, the code is capable of attempting to.

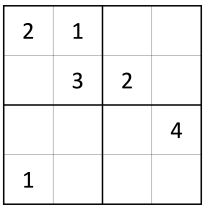

Figure 3: Example order 2 puzzle

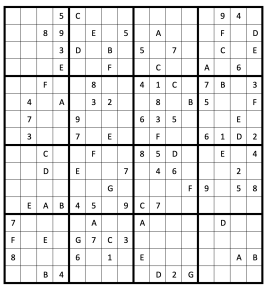

Figure 4: Example order 4 puzzle

1.2 IMPLIMENTING SIMULATED ANNEALING FOR SUDOKU

Before getting into simulated annealing for sudoku, there are first a few definitions we need.

Definition 1.2.1. For a puzzle of order n, a **proposed solution** is any filled in sudoku puzzle that can ignore the rules but has each number 1 to n^2 appear n^2 times, that does not have to obey the rules.

Definition 1.2.2. The **cost of a cell** of a proposed solution is the number of times a rule is broken. As fixed cells are defined to be correct, their cost is zero.

Definition 1.2.3. The **cost of a proposed solution** is the total cost of all non fixed cells of that proposed solution.

Definition 1.2.4. A **neighbour** of a proposed solution is another proposed solution that can be obtained by swapping any two non fixed cells.

The first three definitions are better understood by looking at Figure 5 below. In black are the fixed cells, blue the non fixed cells that are correct (though this won't be known without knowing the true solution) and, in red, non fixed cells that are incorrect. The cost of each cell is marked in red as a sub script.

5	3	1_1	6	7	8	9	1_1	2
6	7	2	1	9	5	3	5_2	8
4_3	9	8	3	4_2	2	4_3	6	7
8	5	9	7	6	1	4_1	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1_1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Figure 5: Example strict proposed solution with costs indicated

Looking at the incorrect non fixed cell in row 3 column 1, the assoceated cost is three as it is on the same row as two other 4's, and same column as another 4. Note that the fixed cell in row 2 column 6 has no cost dispite being on the same row as another 5 as it is a fixed cell. The cost of the proposed solution is $1+1+2+3+2+3+1+1=14$.

You may have noticed that only the rows and columns have conflicts in them, and not the 3×3 sub grids. This is because the restriction of rule three (sub grids) was places on it. This restriction creats a **strict proposed solution** which will lead to more efficient code, and allows for rule three to be ignored when checking costs. So a slight amendment is needed for the definition of a **neighbour** to make sure all neighbours of strict proposed solutions are also strict.

Definition 1.2.5. A **neighbour** of a strict proposed solution is another strict proposed solution which can be obtained by swaping any two non fixed cells in the same $n\times n$ sub grid. Where n is the order of the puzzle.

Below is the sudo code for simulated annealing with sudoku, but first there are two terms we need to define. T is referd to as the temperature of the system, and r the cooling rate which is $\leqslant 1$. For simplicity these can both be set to 1, but they play an important role in efficient simulated annealing implementations.

1. Let C_1 = cost of current proposed solution, and set initial value for T and r.
   a) Select a non fixed point, α_1, from full grid at random.
   b) Select a different non fixed point, α_2, from the same sub grid as α_1 randomly.
   c) Swap the numbers in α_1 & α_2 to create neigbbouring strict proposed solution.
   d) Let C_2 = cost of the new strict proposed solution.
   e)
      - If $C_2 < C_1$: accept the newly proposed solution as the current proposed solution and let $C_1 = C_2$
      - else: with probability $exp\left[\frac{C_1-C_2}{T}\right]$, accept the newly proposed solution as the current proposed solution and let $C_1 = C_2$.
   f) Let $T = T * r$
   g) If $C_1 \neq 0$: repeat step 2

[4]

In summary, simulated annealing takes a strict proposed solution and samples one of its neighbours, immediately accepting it as the next step if it has a lower cost, otherwise sometimes accepting, depending on the difference in the cost and the temperature of the system.

To explain the impact of the temperature, T, and cooling rate, r, it's easiest if we set both to 1. The code performs a random walk around the space, immediately accepting any proposed moves that have a lower cost and sometimes accepting moves that have a greater cost. Keeping r as 1 and decreasing T, the probability of accepting a move with a higher cost is decreased too, and vice versa. This is the purpose of r. Early in the algorithm, T is relatively high and is able to explore many strict proposed solutions that are far apart, as the probability of accepting moves with a higher cost is high. However, as T decreases geometrically, the chance of accepting moves to neighbours with a higher cost is more unlikely, and it starts to only accept moves to neighbours with a lower cost. The hope is that with the high initial temperature, the system will find itself in a region with the solution. As the temperature cools, the probability of moving out of that region decreases, and the system will find the local minimum for the region it is in.

1.3 EXTENDING THE ALGORYTHEM

Having written code that would pick the non-fixed points totally randomly, I found that none of the puzzles were solved. Looking into the problem, I found a paper [1, p. 4], which discussed preferring to sample cells with high cost. The paper implemented this using exponentials, as I discuss later, but first, I wrote a program to a linear function of the cost. To explain it helps first to gain a better understanding of how the original sampler worked. Each point was assigned a probability by calculating a CDF over all the cells, and this was done by adding 1 to the CDF for each non fixed cell, then normalizing with the last value of the CDF, see Figures 7 & 8 for examples. Then the inverse CDF method could be used to sample cells. This code was easily adapted to assign a higher probability to cells with a higher cost by adding the cost of a cell to the CDF instead of 1. However, this didn't work as non fixed cells with 0 cost would not be sampled. So each non fixed cell to $1 + \mathrm{cost}$ before normalization. With this change, the algorithm was able to find solutions.

To extend the idea of favouring sampling cells with a higher cost, I implemented two more ways of calculating the CDF to the constant and linear versions discussed above. The square method had each non fixed cell contribute $(1 + \mathrm{cost})^2$ to the CDF before normalization, and the exponential method had each non fixed cell contribute

Exp [cost] to the CDF before normalization. From a few runs, both of these appeared to be giving quicker results than the linear method. An example for each of these CDF's is given for the CDF of a 3×3 sub grid is given in Figures 6, 7, & 8. The same idea follows for the full grid.

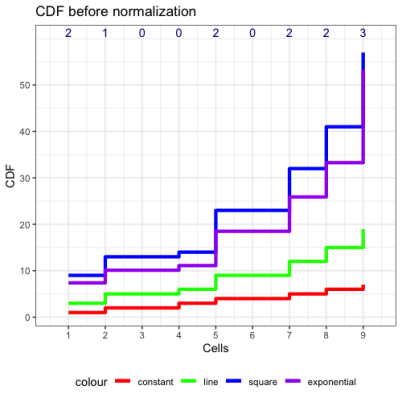

Figure 7: Pre normalized CDF from Figure 6

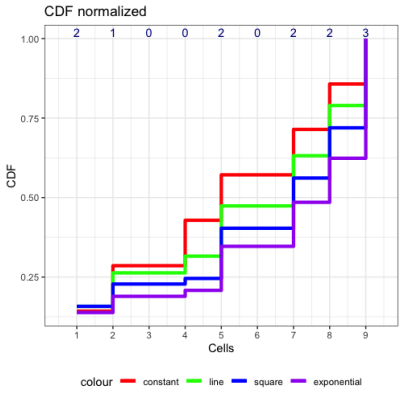

Figure 8: Normalized CDF from Figure 6

In Figure 6, the black numbers indicate the fixed cells, and the blue the non fixed. The red subscripts the cost of each cell. The cost of each cell is shown above each cell in Figures 7, & 8. The cells are orderd left to right then by row so that the two fixed points are cell numbers 3 and 6, which can be seen by the extended flat portions of each CDF in Figures 7, & 8.

2_2	3_1	1
4_0	5_2	7
6_2	8_2	9_3

Figure 6: Example sub grid with costs

Focusing on the last cell, which has a cost of 3, the largest of the sub grid. We can see that the exponential CDF gives the larges probability of sampling the last cell as about 1/3 of all y values will lead to this cell. In contrast to cell 4, which has a cost of 0, the exponential CDF gives this the smallest chance of sampling this one.

For the implementation, I focused my computing power on the linear, squared and exponential CDF's. I spent some time on the constant CDF just to show how poor it was.

2 INTRODUCTION TO THE CODE

In this section, we will be looking at the structure of the code used to implement the above algorithms and discuss some preliminary results.

2.1 THE STRUCTURE

To accommodate the different CDFs, each has it's own header file (ending in .h). Each header just contains the class, *Sudoku_SA*, and everything needed to use it. Each is almost exactly the same as the others, whit the only differences coming from the calculation of the CDF's; once the CDF's are calculated, the sampling procedures are the same. For the constant CDF header file, the CDF is made a member of the class and created in the constructor. This avoids recalculating the CDF for each new sample as is required for the other header files. For these, the CDF is only calculated when a sample is needed. There are two sampling functions, *sample_full_grid*, which samples a non fixed cell from the entire n^2xn^2 grid, and *sample_sub_grid*, which samples from the same nxn sub grid as a provided point, the one from *sample_full_grid*.

2.2 THE STRUCTURE OF SUDOKU_SA

Sudoku_SA has four structures which hold the data about the current solution:

- ***order*** (intiger): holds the order of the puzzle. Remains unchanged from initialization.
- ***solution*** (2D array of integers): holds the current proposed solution.
- ***fixed_points*** (2D array of Boolean values): marks the positions of the fixed cells are with 1's.
- ***cost*** (2D array of integers): holds the cost of each cell for the solution above.

There are two constructors for *Sudoku_SA*, one to read in a puzzle from a provided file, and a second to copy another *Sudoku_SA* instance. The input file containing the puzzle can take many forms, a few examples are given below in Figure 9. All read elements are assumed to be numbers, being passed to int's. The first number is

the order of the puzzle, the next n^4 make up the puzzle itself. The data is read in using the stream extraction operator (»), so each number should be separated by a blank space. The non - fixed cells are marked with 0's.

```
3
082 005 139
015 080 000
907 000 000
020 036 078
090 507 020
170 290 060
000 000 604
000 060 790
269 700 580
```

```
3
082 005 139
015 080 000
907 000 000

020 036 078
090 507 020
170 290 060

000 000 604
000 060 790
269 700 580
```

```
3082005139
015080000
907000000
020036078
090507020
170290060
000000604
000060790
269700580
```

```
308200513901508000
009070000000020036
078090507020170290
006000000060400000
607902697005800
```

Figure 9: Examples of possable input files for *Sudoku_SA*

2.3 DESIGN OF *sudoku_general.cpp*

I've designed the code so that each hader file, with a small change, can be used in the same *.cpp* file, *sudoku_general.cpp*. Each header has a corresponding include statment at the top of *sudoku_general.cpp*, that are commented out, simply uncoment the relavent header file, and *sudoku_general.cpp* is ready to compile.

To avoid running endlessly when parameters don't allow for convergence, I only allowed for 10 attempts before stopping. When stopped, it will output a '+' in front of the number of steps taken to indicate that the observation is right-censored.

As the number of steps is generally greater than $1,000$, up to $1,000,000$, understanding the order of magnitude of the output at a glance is difficult. To help, the output number of steps is divided by $1,000$ to use the decimal to help with this distinction.

2.4 ATTAINING PUZZLES

To get puzzles on which to test the algorithms, I used those from the website https://www.websudoku.com/, as I was able to write a bash script, *find.sh*, that could download the puzzles, and put them into .txt files that could be fed into the program. The far-left example in Figures 9 is one of these. It was important to write a script to do this as doing it manualy is very prone to mistakes.

However, there were two problems with this. Firstly they only provided puzzles with between 45 and 57 non fixed cells, which does not include the 64 non fixed cells of a minimal order 3 puzzles. Secondly, some puzzles in that interval aren't sampled very often, for 48 non fixed cells, I only got 2 puzzles after overloading over 1,000 puzzles. Neither of these proved to be too concerning as to get a rea-

sonable estimate of the mean number of steps to solve each puzzle, I ran each 30 times. This took so long that doing many more puzzles would have proven unmanageable. As a compromise, I decided to use the available puzzles to pick the best parameters, then see how it performs outside the region of the fitted parameters.

2.5 RUNNING PUZZLES

As so many puzzles had to be tested, 30 times for each set of parameters, I ran them through another, short, bash script, *run_100.sh*. This would take a set of parameters from a file, *variables.txt*, then for each combination of parameters, it would run the puzzle 30 times. The results were in a file put into a subdirectory, *outputs*. An explanation of the output is given in Figures 11

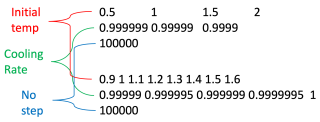

Figure 10: Examples of *variables.txt*

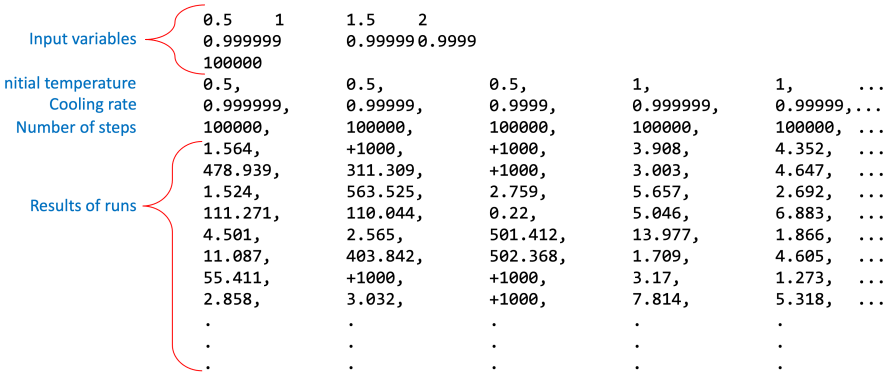

Figure 11: Examples output for *run_100.sh* with explanation of each line

3

FITTING PRAMATERS

This section will look at fitting the parameters and how these fits compare with more difficult puzzles.

3.1 FITTING PARAMATERS

There are three parameters to tune in the simulated annealing algorithm, the initial temperature, colling rate and the number of steps before restarting. Initially, the plan was to tune all three parameters; however, as mentioned before, the running time for a single puzzle can be quite long. The time to perform the first search, as we will see below, had it included just three values for the number of steps would have been over a week. I decided that the least important parameter was the number of steps. With well-chosen values for initial temperature and colling rate, it shouldn't be very common to need to repeat. This had the added benefit of making the search space 2 dimensional and easily plottable. in Figure 12. With the benefit of hindsight, this reasoning did not hold, but still the right decision given my limited computing power.

I decided that the number of iterations was the lease important as if the other two were chosen well, the number of repeats should be small, I chose 100,000 as from some initial analysis this appeared to be around the optimum value. Now with only 2 paramaters I was also able to visualise the parameter space on a grid. As mentioned before, the constant CDF didn't give any solutions, so that could be avoided too, leaving three header files to test.

My plan was to test run times. However, this proved impractical, if not impossible. The long-run times led to high temperatures and the slowing down of the CPU. Ignoring that, the change of ambient temperature from day to night and between days would also mess about the calculations. Instead I used the number of steps until a solution was found as a proxy.

The below plots inf Figure 12 show the results of an initial search of the parameter space with each header file. The more blue a pair of parameters, the fewer steps required. The grey cells indicate that there was a puzzle that was never solved after running *sudoku_general.cpp* thirty times. The numbers given for the cells indicate the number of times funning *sudoku_general.cpp* did not converge to a solution.

The calculations of the mean and median are wrong because of the censoring in the data. The means were calculated by simply ignoring any censored data. While not the actual mean, any pair of parameters

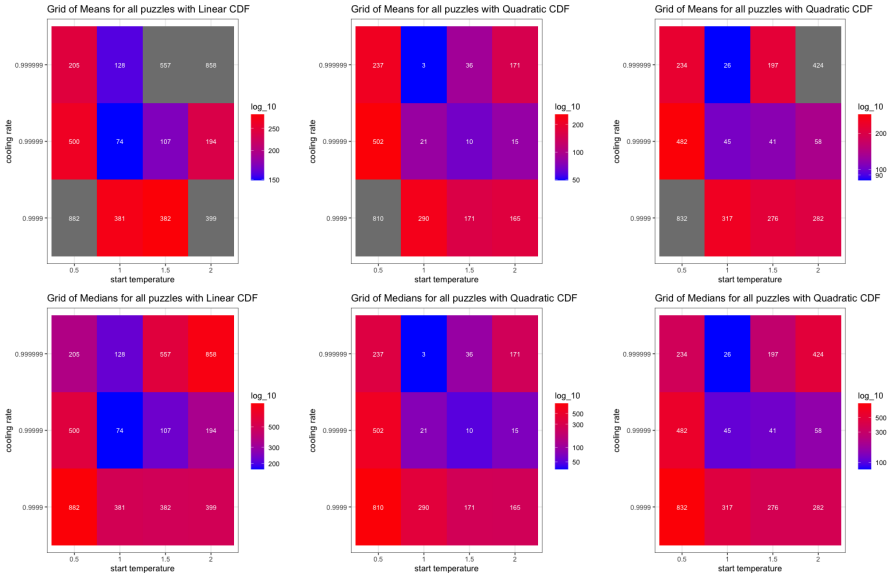

Figure 12: Results from fit of inital parameters search. The colour of the blocks indicate the average run times, and the number indicates the number of attempts to fail to find a solution.

with enough censored observations to significantly affect the calculation will be so poor that they can be safely ignored. The median uses all the data, taking the censored time as the actual time, this is more sound as the median only cares about the order, and any pair of parameters with the censoring time as the median is poor enough to ignore. I didn't want to completely ignore the censored observations, so I included the number of censored observations on the graph.

Looking at all the plots as a whole, it looks like they have similar optimum parameters around a starting temperature of 1, and cooling rate around 0.99999 and 0.999999. However, an initial temperature of 1.5 also performs well too. For a finer search i chose to search initial temperature 0.7 to 1.6 in 0.1 increments, and cooling rates 1, $1-5\times10^{-}7$, $1-10^{-}6$, $1-5\times10^{-}6$, $1-10^{-}5$, & $1-5\times10^{-}5$. The results of this search can be seen below in Figure 13.

It looks like the exponential model has a minimum at an initial temperature of 0.9 and cooling rate of 0.999999. The quadratic looks to be minimized for several values around inital temp of 1, cooling rate between 0.999999 and 1. The linear model has a similar cooling rate to the others but a significantly lower initial temperature. If we find the minimum values for each model, we get the results in Table 1. Interestingly they are all the minimum for both mean and median.

As suspected, the Linear model performs the worst, on average requiring more than the 100,000 steps of one attempt to converge. It also failed to find a solution 10 times more than the quadratic model. Interestingly, all the models chose very small cooling rates. The smallest of 0.999999 will only cool the initial temperature by

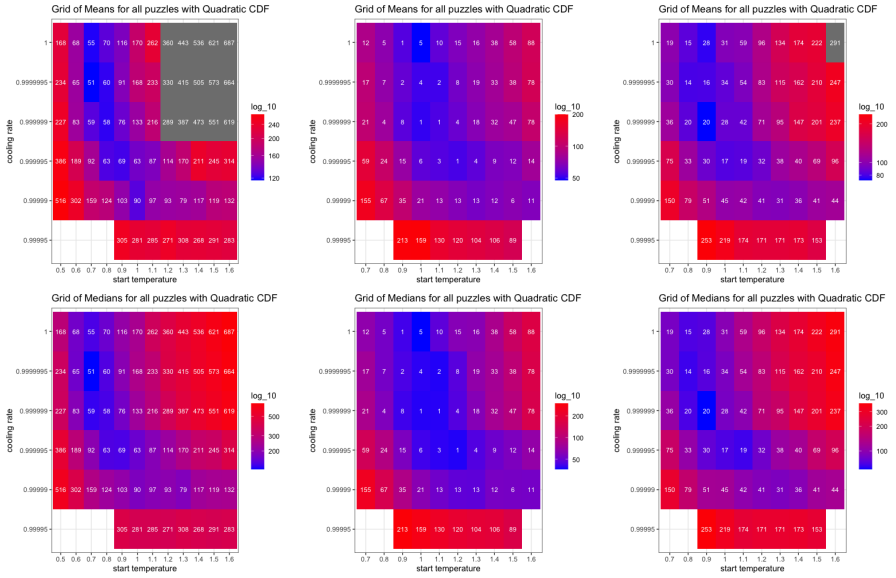

Figure 13: Results from fitting of fine parameter search.

CDF Model	Linear	Quadratic	Exponential
start temp	0.7	1	0.9
cool rate	0.9999995	1	0.999999
No. steps	100,000	100,000	100,000
mean steps	116.430	47,522	72,799
median steps	123.771	36,328	60,083
No fail	51	5	14

Table 1: Top proforming results from each header file.

$0.999999^{10{,}000} \approx 0.9$ at the end of a search. Arguably even more interesting is that the Quadratic has chosen to have no cooling, instead of looking at the have and an acceptance rate of $exp\left[-|\Delta \text{cost}|\right]$ for all steps with a greater cost. This makes it into more of a random search than simulated annealing

3.2 ASSESING FIT & GENERALIZATIONS

To assess the performance of the Quadratic model for more difficult problems, I tested it on two types of problems. First, on order 3 puzzles with only 17 fixed cells, these are the most difficult order 3 puzzles as those with any fewer fixed cells have multiple solutions [3]. Second, on order 4 problems, I used a wide range to get an idea of their performance over most order 4 puzzles. However, the program didn't find solutions often for either class of problems. To avoid numerous censored data point, I adapted the code to print the

lowest attained cost after 100,000 steps. The below Figures show the means and medians in both their colours and the values in the blocks.

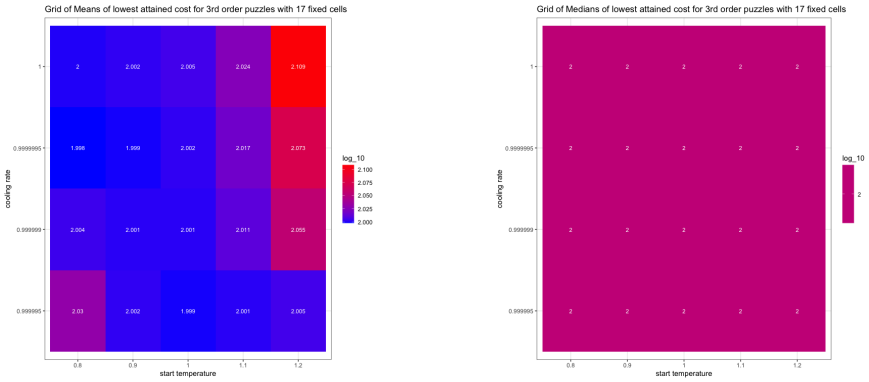

Figure 14: mean and medians of number of steps for each model over a range of possable paramaters

For the order 3 puzzles with only 17 fixed cells, we get a similar result to the fitted values. This time however, the minimum comes from an initial temperature of 0.8 and cooling rate of 0.9999995. Though all the values only differ on the fourth significant figure. The plot of medians is of limited value all the lowest costs are 2.

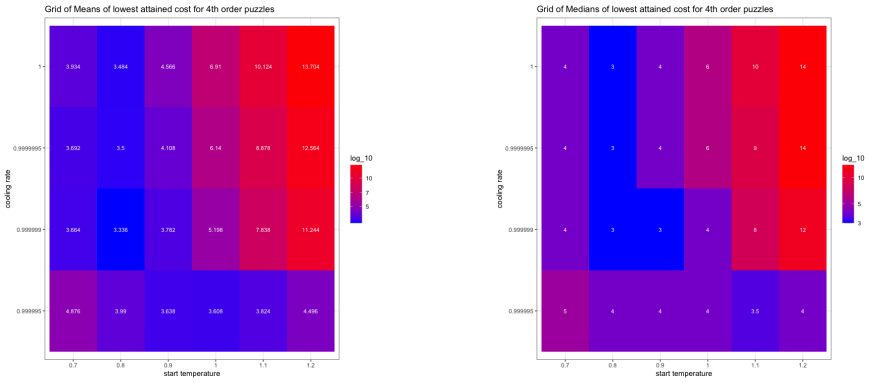

Figure 15: mean and medians lowest attained cost for 4th order puzzles for various paramaters of the quadtratic model arroung it's best fitted values

Surprisingly, some of the order 4 puzzles found solutions, though it was not very common, and some of the other puzzles never found one. The best parameters look to have changed too, with a lower initial temperature of 0.8, while the best cooling rate could of 0.999999.

For both these groups of the puzzle, it appears that while it is rare to solve them with simulated annealing, the cost function frequently get very close to 0.

4

EXTENDION TO RUBIC CUBE

This section will look at code used to solve a Rubik cube using simulated annealing and compare it to that of another paper.

With the aim of extending this project while remaining focused on simulated annealing, I wrote another program designed to solve a Rubik cube. Having never solved one myself, I wanted to see if I could easily develop my own program without checking if it had been done before. When my code did not work, I found a paper that discusses how they got it working, and we will be examining the differences.

4.1 MY ALGORITHM

As this is only intended as a demonstration of possable problems with simulated annealing, we will avoid the long explanation given for sudoku. We will be sticking to the clasic $3 \times 3 \times 3$ Rubik cube, Though I wrote my code to be easily adapted to larger cubes. It is assumed that the idea of rotating faces of the blocks so that all colours on a face match is understood without and explanation required.

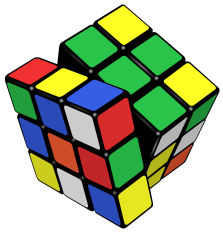

Figure 16: Example Rubik cube

In Figure 17 is an example of a flatende cube, and in Figure 18, how this was stored in the program as six 3×3 arrays of intigers.

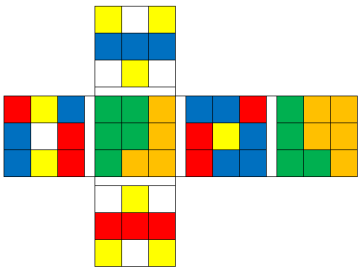

Figure 17: Example example Rubik cube

```
        6 1 6
        4 4 4
        1 6 1

2 6 4   3 3 5   4 1 2   3 5 5
4 1 2   3 3 5   2 6 4   3 5 5
4 6 2   3 5 5   2 1 4   3 3 5

        1 6 1
        2 2 2
        6 1 6
```

Figure 18: Figure 1 as held in code

The idea of moving between neighbours with an initial temperature and cooling rate is the same as for sudoku. We only need to define what is a neighbour and the cost function.

Definition 4.1.1. A **neighbour** of a state of a Rubik cube is any state which can be obtained by rotating one face.

Definition 4.1.2. The **cost** of a state of a Rubik cube is the sum over each face of the cost of that face given by 9 - the number of occurrences of the most common colour on that face.

In my code, I tried both sampling rotations of faces at random, and finding the cost of each neighbouring state and using that to construct a linear CDF, much as in the sudoku puzzle. Neither of these found any solutions, nor reduce the cost considerably.

4.2 COMPARING TO A FUNCTIONING ALGORITHM

Having written the code, I found a paper [5], which used a very similar method to what I tried, just using a different cost function.

Definition 4.2.1. The **cost** of a state of a Rubik cube is the sum for each face 8 - number of colours on the face that match the middle one.

As an example, take Figure 17, and take the face with yellow in the centre square, my cost for this face would be $9 - 5 = 4$ as there are 5 blue squares. In contrast, the cost for that face from the paper would be $8 - 0 = 8$, as no other squares are yellow.

This small but meaningful difference was between the programs was the difference between one that was pretty much worthless and a one that worked almost every time. This points to both the power of simulated annealing and its downfall. The algorithm presented by Saeidi, is relatively basic and effective; however, the complex nature of moving between neighbours of a Rubik cube meant that a small difference could ruin it.

5

CONCLUSING AND FURTHER RESEARCH

5.1 CONCLUSINS

As discussed at the end of the previous section, simulated annealing can be a very powerful tool when applied to problems that it is designed for. Simulated annealing is best applied to problems where we want to find a minimum of the cost function that is close to the global minimum, if not the global minimum itself. In this paper, we have looked at using simulated annealing to find the one global solution, and when the search space is small, as in the sudoku puzzle used to fit the parameters, this is very much possible. However, as the search space grows, an intimate knowledge of the problem may be needed as in the Rubik cube or prove to be near impossible, as for the order 4 sudoku puzzles.

5.2 FURTHER DEVELOPMENTS

While I think further research would lead to limited results, I would like to understand how the number of steps affects the efficiency of the code. With computing time forming the main limitation, there are a couple of minor efficiency improvements that can be made to slightly improve run times.

The least efficient part of my sudoku code comes from calculating the cost and CDF for each step. Firstly, the CDF should be held between steps and only recalculated when a neighbour is accepted. As explained in the paper by Lewis [2, p. 5], the cost for every cell doesn't need to be updated after a swap. Only the cells on the same row or column do. This could be implemented both for the cost and the CDF pre normalization if the contributions for each cell were held between steps. These improvements should significantly improve run times, though it will still take several hours to run for several parameters over a few puzzles.

BIBLIOGRAPHY

[1] Eric C Chi and Kenneth Lange. "Techniques for solving sudoku puzzles." In: *arXiv preprint arXiv:1203.2295* (2012).

[2] Rhyd Lewis. "Metaheuristics can solve sudoku puzzles." In: *Journal of heuristics* 13.4 (2007), pp. 387–401.

[3] Gary McGuire, Bastian Tugemann, and Gilles Civario. "There is no 16-clue Sudoku: Solving the Sudoku minimum number of clues problem via hitting set enumeration." In: *Experimental Mathematics* 23.2 (2014), pp. 190–217.

[4] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, 2007, pp. 549–555.

[5] Shahram Saeidi. "Solving the Rubik's Cube using Simulated Annealing and Genetic Algorithm." In: *International Journal of Education and Management Engineering* 8.1 (2018), p. 1.

6

APPENDIX: FINE PLOTS

The below plots are the same as from Figures 12 & 13, as in section 3, except, the plots are show results for each set of number of non fixed cells.

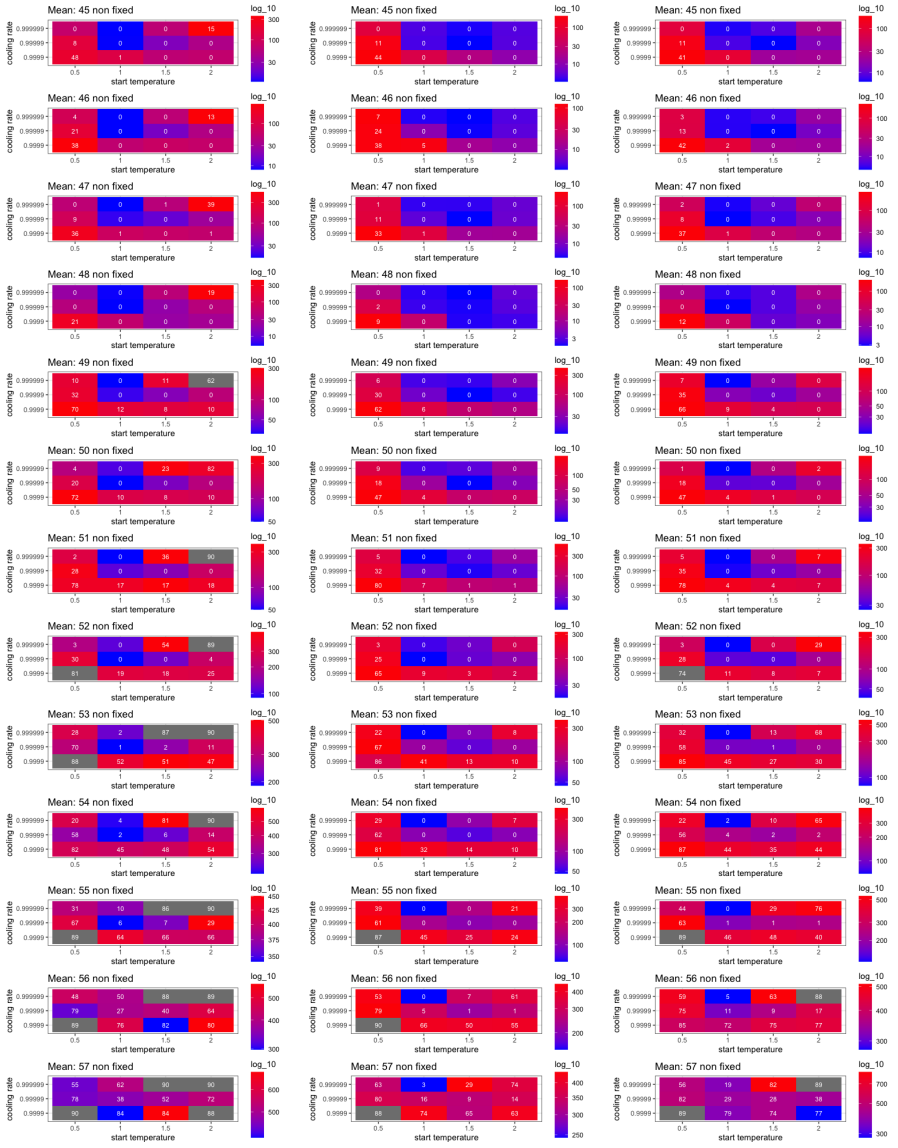

Figure 19: Results from fit of inital parameters search for each group of non fixed cells.

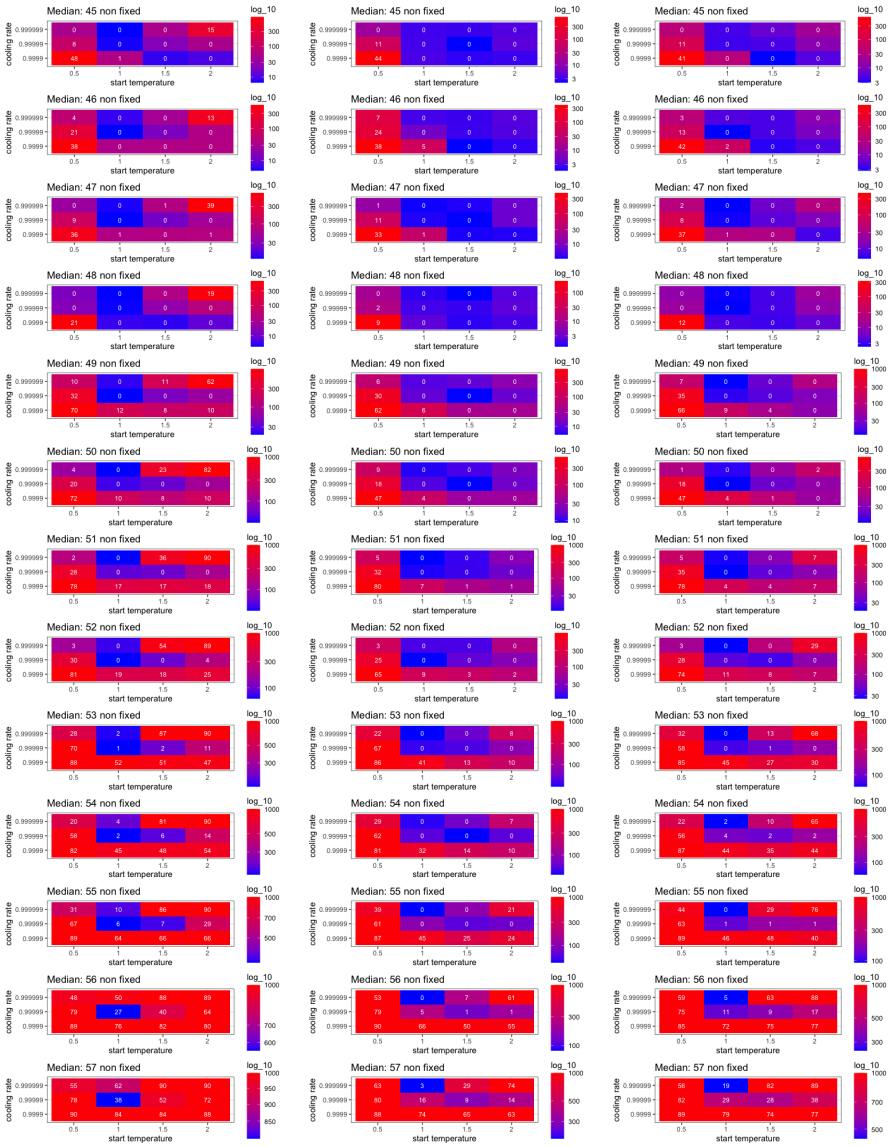

Figure 20: Results from fit of inital parameters search for each group of non fixed cells.

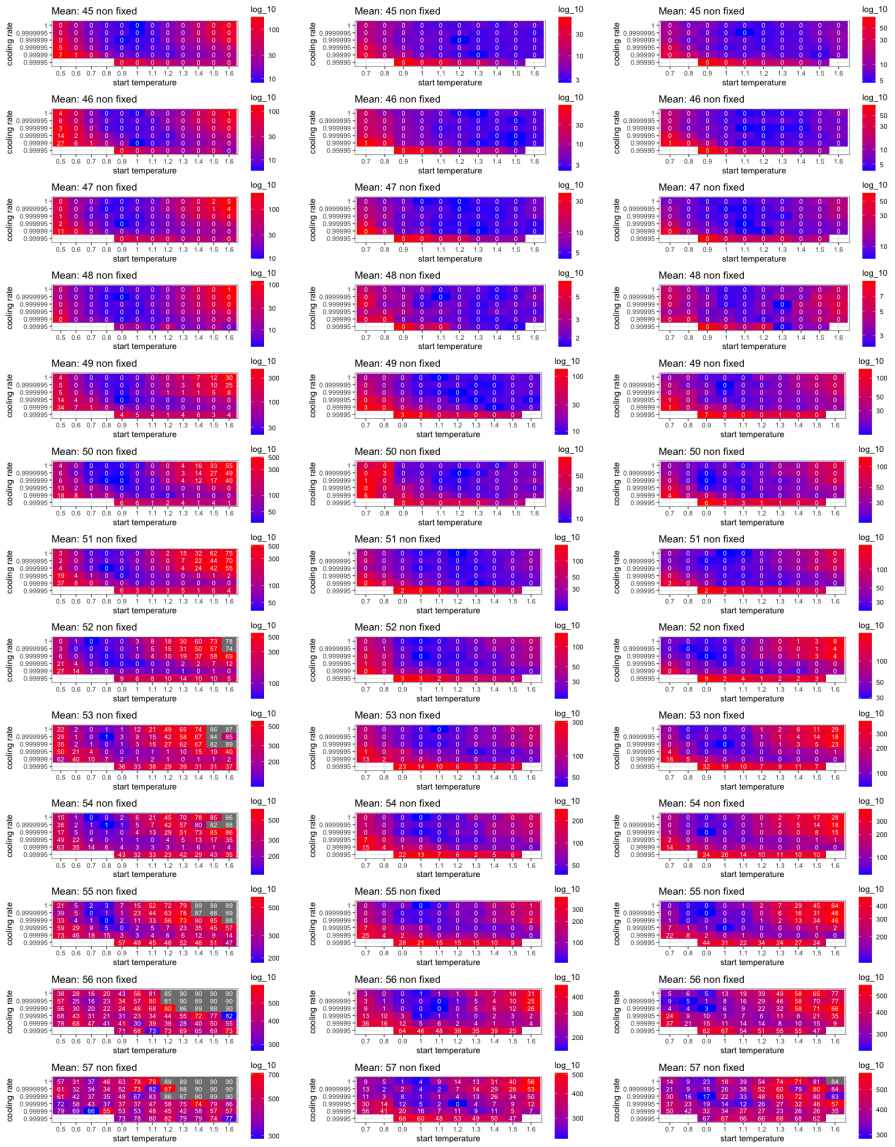

Figure 21: Results from fitting of fine parameter search for each group of non fixed cells

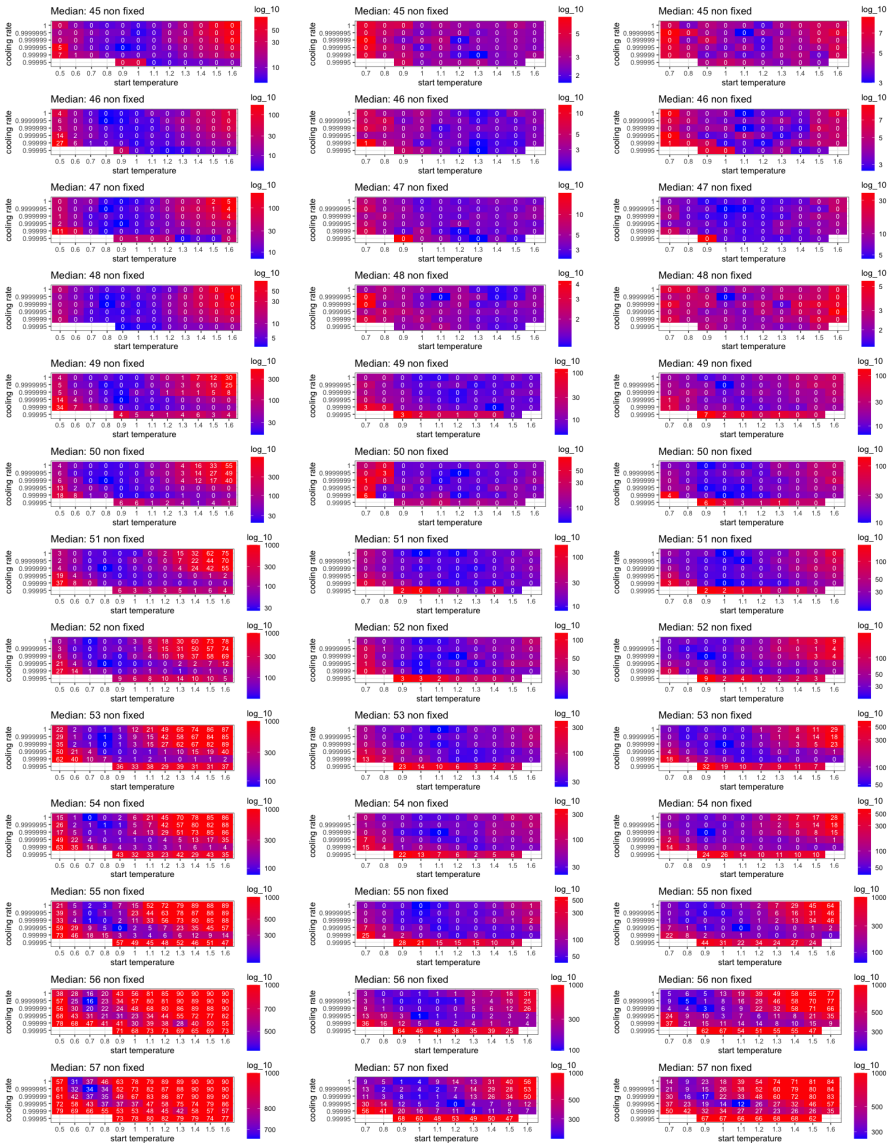

Figure 22: Results from fitting of fine parameter search for each group of non fixed cells