AAA Project

Liron Mizrahi 708810 | Daniel da Silva 738215 | Jan Badenhorst student no.

September 2016

1 Aims

The problem proposed is to solve Sudoku's using the Backtracking Algorithm and to verify the theoretical analysis of the algorithm using empirical analysis.

2 Summary of Theory

A Backtracking algorithm is a type of brute force algorithm that incrementally builds candidates to the solution. If a candidate is found to be invalid, the algorithm 'backtracks' and deletes it. This only happens if the algorithm determines that the candidate cannot possibly lead to a solution.

3 Experimental Methodology

3.1 Understanding the theoretical analysis

The time complexity for this algorithm can be seen by working backwards from a single blank square. If there is only one blank square, then in the worst case there are n possibilities that must be worked through. If there are two blank squares, then there are n possibilities for the first square and n possibilities for the second square corresponding to each of the possibilities for the first square. If there are three blank squares, then there are n possibilities for the first blank. Each of those possibilities will lead to a puzzle with two blank squares that have n^2 possibilities. The algorithm performs a depth first search through all the possible solutions. So carrying on in this way, the worst case complexity will end up being $O(n^m)$, where n is the number of possibilities for each square and m is the number of blank squares.

3.2 Deciding on Appropriate Hardware and Programming Language

We are more interested in the rate of growth of the performance of the algorithm than the actual running times and as such the specific hardware used will not be a relevant factor so long as the tests are run on the same hardware. However, the programming language that has been chosen is C++, as it is much easier and more reliable to time the algorithm with the OpenMP library, rather than Java's timing implementations. Also, Java would be easier to use for this sort of problem but the Garbage Collector may change results drastically and lead to incorrect results.

3.3 Deciding on Appropriate Data Structures

The data structure used is a three dimensional array. This is to store multiple sudokus (6 from the number of sudokus per text file). Prior to being passed to the backtracking algorithm, however, a sudoku is copied into a new two dimensional array. This is to reduce the memory access times required to reference the three dimensional array.

3.4 Implement the Algorithm

```
1: Solve(intmatrix[][])
2: int row
3: int column

4: if there are no empty squares then
5:   return true
6: end if

7: for value from 0 to 9 do
8:   if there are no numbers in same row, column and 3x3 square as value
     then
9:     matrix[row][column] ← value
10:    if Solve(matrix) then
11:      return true
12:    end if
13:    matrix[row][column] ← 0
14:  end if
15: end for

16: return false
17: EndSolve
```

The algorithm above takes in a sudoku as a 2D matrix. The if statement on line 4 checks if there are any empty blocks in the matrix, if there are then *row* and *column* are assigned to the corresponding block. If there are no empty blocks then the sudoku is complete.

The for loop on line 10 will iterate through the values 1 to 9 for the corresponding empty block. The if statement underneath checks if the current value can be placed in the block without breaking any of the rules of the sudoku. If it can be placed there then $matrix[row][column]$ is updated to the value.

The if statement on line 10 will recursively call the Solve function for *matrix* with the newly inserted variable. If the any function call reaches line 13 then it means the current setup has failed and that the latest value must be backtracked. This line will reset the current block to 0. Line 16 is the line that invokes backtracking. When a $false$ value is returned then the previous block that was updated must be tried again for the other values it has not tried yet. If every function call returns true then the sudoku has been solved. If the first function call returns false then the sudoku has no solution.

3.5 Create the Data

Data was generated using the website https://kjell.haxx.se/sudoku/. A field of 6 sudokus was generated for each number of given numbers from 17 up to 35. Another website (https://regex101.com/) was then used to convert the sudokus into the required format.

A metric was designed to approximate the difficulty of the puzzles using the formula:

$$\frac{number\_empty\_cells}{max\_number\_completed\_cells\_(block/row/column)}$$

This metric is useful in approximating the difficulty of the puzzle and was found to agree with the difficulty levels quoted by the website for each puzzle. However, our main analysis uses the number of empty cells vs time. This metric is just an attempt at evaluating what makes a sudoku difficult.

4 Presentation of Results

4.1 Number of empty cells

Firstly we examine the graphs of time vs number of empty cells.

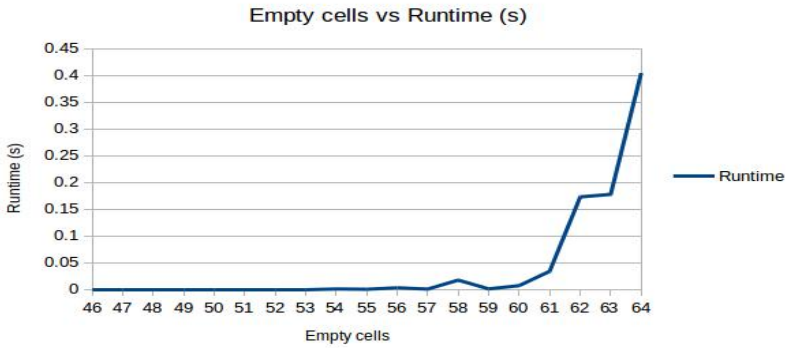

Figure 1: Aggregated data Empty cells vs time

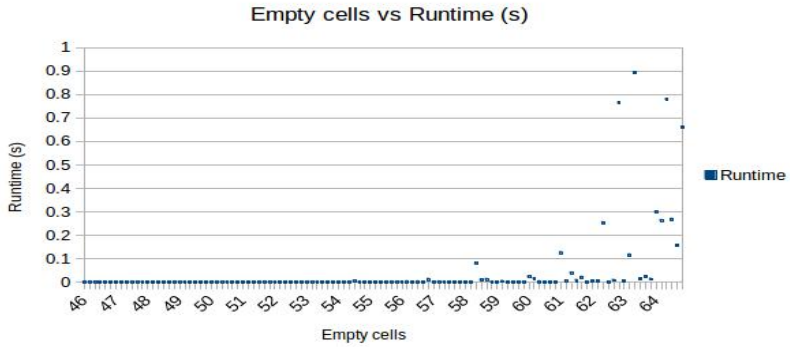

Figure 2: Scatter plot data Empty cells vs time

4.2 Difficulty metric

We now examine the test metric graphs to evaluate how good of an approximation it is for difficulty.

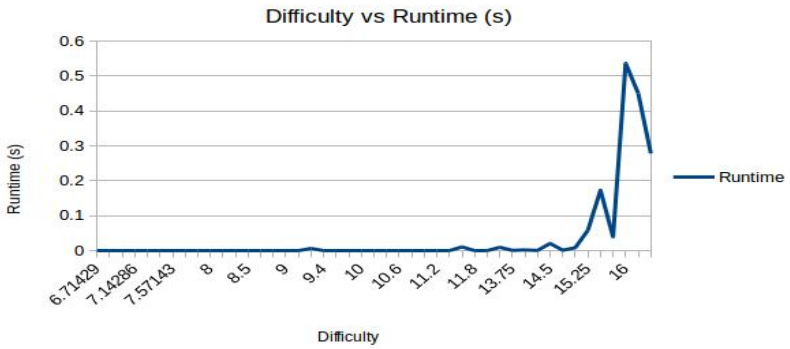

Figure 3: Aggregated data Difficulty vs time

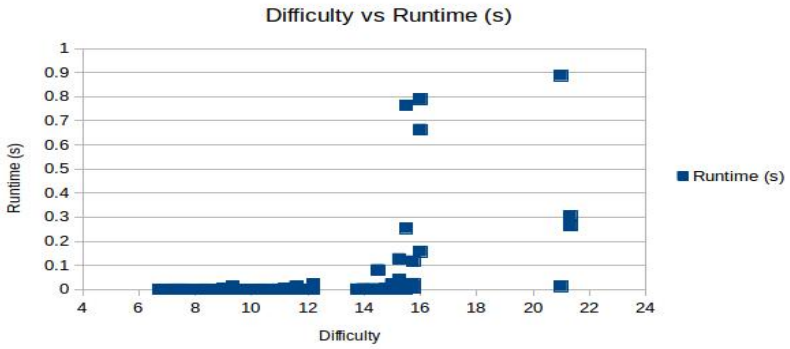

Figure 4: Scatter plot data Difficulty vs time

5 Interpretation of Results

5.1 Number of empty cells

It is clear from figure 1 that there is a sharp increase in runtime from 61 empty cells to 64 empty cells. This growth is exponential and would likely continue for sudokus with a higher number of empty cells. A somewhat small increase in runtime can be seen from 62 to 63 empty cells. We can ascertain that this is variation due to a few outliers as seen in figure 2. The number of sudokus taken into consideration for a given number of cells is only 6, so variations like these are to be expected. There is another variation in figure 1 at 58 empty cells. We can again ascertain that this is due to, in this case, a single outlier as seen in figure 2. There is a lot of variation in this problem as for some sudokus the algorithm may be 'lucky' and guess right multiple times without having to actually backtrack. Given that there are only 6 sudokus per number of empty cells, the relationship is considerably strong. It is also important to note the small runtimes for 46 to 57 empty cells, which provide small changes that cannot be viewed in figure 1 due to the scaling. Refer to figure 5 below for the graph of that region.

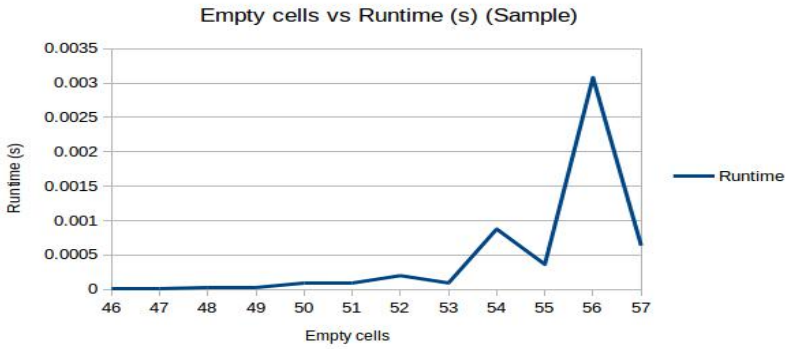

Figure 5: Aggregated data Empty cells vs time for 46-57

Here we see an exponential relation as well, which again, appears as a straight line on the graph in figure 1. This really outlines the magnitude with which the runtime increased in figure 1.

5.2 Difficulty metric

The difficulty metric mostly mirrors the number of empty cells graphs, however there are much larger variations in the graphs. The metric takes into account difficulty a human would have in solving a sudoku as opposed to what the algorithm might struggle with. It does not seem to provide a better approximation than the number of empty cells. Notice the large variation right at the end of the graph (at around 21 difficulty units), this is clearly a sudoku being rated as quite difficult when it should not be considered as such. This is why the metric is not considered in the empirical analysis.

6 Relate results to Theory

7 Conclusion

8 Acknowledgments

Thank you to Kjell Ericson for the test data which is created on demand at https://kjell.haxx.se/sudoku/.

9 References