UNIVERSITY OF THE WITWATERSRAND

UNIVERSITY OF THE
WITWATERSRAND,
JOHANNESBURG

Project Report
Advanced Analysis of Algorithms

Sibusiso Mthethwa (570088) Liam Pulles (855442)
Tshepiso Molobi (748877)

31 October 2016

Contents

1 Introduction

1.1 Aim and Objective

Determining the relative measurement of an algorithm from theory, and comparing them to reality is one decisive idea in computer science. With this we can decide if the algorithm solution if practical or not. Provided that it is indeed practical, we will thus try to identify the points within the problem field that can be successfully used before exhausting the system. Consequently, implementing a Backtracking algorithm using java. The Backtracking algorithm will be used to solve the partially solved diverse Sudoku puzzles reserved in the database. To implement and test the Backtracking algorithm in relation to solving Sudoku puzzles given various number of empty spaces and of course these puzzles are of different arrangements. We will justify the theoretical analysis of the algorithm. A number of experiments will be conducted on the algorithm to measure the performance of the algorithm, and comparisons with the theoretical analysis will be made. With the number of empty spaces, we will do the Emperical analysis of the backtracking algorithm.

1.2 Problem Statement

We want to determine the complexity of our backtracking algorithm. We have done a theoretical analysis of our algorithm, but we need to see if this holds true in reality. An Empirical analysis on our implementation of the backtracking algorithm will be done, to see if the actual complexity conforms to our theoretical complexity.

2 Theoretical Background

2.1 Sudoku Principles

Sudoku is a ralational and insightful puzzle, and to it the obejective is to pack a 9 by 9 grid with numbers such that every column, every row, and every 3 by 3 subgrid has all digits from 1 to 9 appearing precisely once. Each grid has an initial state with some number of cells having digits assigned to them.

2.2 The Backtracking Algorithm

The backtracking algorithm is an algorithm with common qualities with that of the brute force algorithm which is uses a lot less memory. The brute force algorithm will find an empty cell, then find the smallest valid digit from 1 to 9, that is not equal to any digit in the present space, column or row. Provided the digit is valid, the current empty space will be assigned this digit. This algorithm will be recursively called. If no valid digit is found, the algorithm will be called recursively while backtracking if necessary.

3 Theoretical Analysis

A single cell belonging to a Sudoku puzzle contains a unique digit column wise and row wise. Provided a cell with a digit, each cell corresponding to that cell column wise or row wise is prohibited from having the exact number. This restriction we will be verified by a method in relation to how a given Sudoku puzzle is in memory represented. The backtracking algorithm is performed on this logic puzzle, such that if we reach a dead-end while solving the puzzle, then the algorithm will reverse the move and it will try to explore other different paths that have not been traveled until it gets the solution to the puzzle.

In general terms, the algorithm has a complexity that is highly dependent on how complete the given puzzle is, where we consider the number of empty spaces in the Sudoku puzzle. Below are the cases we will look into

Best Case:
The best case scenario is when the path chosen from the beggining to the end is correct, thus the algorithm does not need to backtrack.

Worst Case:
In the worst case scenario all the paths have to be traveled before getting to the final arrangement of the Sudoku elements, as a result the algorithm backtracks at every chosen digit.

3.1 Time Complexity

Worst Case: Let j denote the number of empty cells in a given Sudoku puzzle. We will use undirected acyclic trees to structure our solution explanation. For each empty cell, there exists 9 possibilities. We will have 9 nodes added to the tree, with each new node representing a possible solution to a cell first looked into. These nodes will be connected to our arbitrary root node through an addition of 9 edges. Moving to the second empty cell which will as well have 9 possibilities. We add another level to our tree through adding 9 nodes to all the nodes in level 1. Continuing in this way, we will get to a problem that comes about by a brute force approach. Our backtracking algorithm will take action in way similar to that of the depth first search algorithm upon the constructed tree. The algorithm will go over one branch of the tree up to violating a condition under the Sudoku Fundamentals where it will backtrack and try a different tree branch. Coondidering a tree of n nodes and m edges, the Depth First Search has a complexity of O(n+m). Now looking into our problem,

$$n = 9^j$$

given 9 nodes at every level for each of the nodes. Assuming that all the provided Sudoku puzzles may not have the number clues below 17, then the max number of empty spaces (j) is 81 - 17 = 64. We in the worst case take our upper bound to be

$$O(9^{64})$$

since j can never go beyond 64.

Best Case:
The best case for this algorithm is that it discovers the solution to the problem the very first time through, that it won't have tp backtrack at all. This means that every predicted digit by the algorithm, this being the first guess is well found and thus correct with the algorithm not having to backtrack. This does not by any chance depend on the number of empty spaces. Since the algorithm never backtracks, the first node of the tree is the only node to be iterated through. This results in the complexity

$$n = 1^j$$

, j = 64 is the upper bound.

$$1^{64} = 1$$

, thus we can conclude that O(1) is the complexity in the best case.

4 Experiment

4.1 Experimental Methodology

Begin a Java program that has the functionalities below:

- Allow input and storing of a Sudoku puzzle with various number of empty spaces. These puzzles will read at random from a database.
- Use the Backtracking Algorithm to find a solution to a given Sudoku puzzle.
- Measure the average time it took the Backtracking Algorithm to find a solution Sudoku puzzle.

For accuracy's sake the experiment was done multiple times. And the graphs of time vs number of empty spaces was obtained to make the performance of the backtracking algorithm feasible enough, that we can be able to conclude on its complexity.

4.2 Apparatus

- NetBeans IDE - Java Code
- RAM (4 GB)
- Linux Operating System (Arch, latest linux kernel)
- Processor (Core i5 3.8GHz)

5 Results

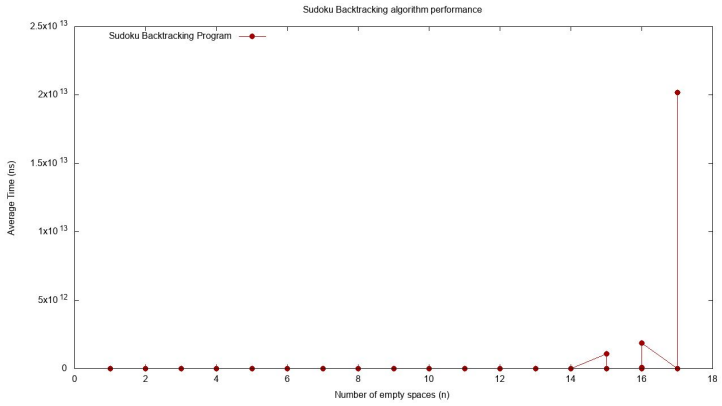
5.1 Presentation of Results

The results presented below are that of the experiments that were conducted and their graphs.

```
run:
1 478806
1 30242
1 30805
1 28148
1 51677
2 45319
2 41418
2 40067
2 40936
2 130196
3 150665
3 78845
3 59195
3 61454
3 72865
4 89454
4 87613
4 51140
4 49494
4 58355
5 58001
5 60389
5 139393
5 58528
5 85880
6 742382
6 51764
6 41114
6 102057
```

```
6 47932
7 53636
7 50123
7 52320
7 55677
7 55687
8 59854
8 60394
8 2335439
8 59505
8 67436
9 73053
9 84447
9 67702
9 78128
9 67042
10 101752
10 74095
10 65269
10 194361
10 60955
11 74254
11 72199
11 72855
11 74290
11 73658
12 84626
12 87492
12 92483
12 84457
```

```
12 81492
13 94575
13 96825
13 13588192536
13 51086
13 10033172
14 27633
14 27054
14 21171250
14 26559
14 37947
15 1063553712915
15 30017
15 29143
15 1326547
15 667642263
16 33435
16 688187511
16 31798
16 92129469188
16 1860265946299
17 1120090
17 414897
17 20195797762465
17 56151
```



5.2 Results Analysis

The results that appear in fig:1-30, fig:31-61 and fig:61-85 were gathered from the experiments that were performed. The results show two columns with the first one being the number of empty blocks on the Sudoku board.The second one represents the time taken(in nano seconds) to solve the board.

From the results that were gathered a graph was drawn up to show the complexity of the algorithm as the number of empty blocks increases. The graph depicts what seems to be an exponential growth, which as hypothesized suggests that as the input size of the empty blocks increases, more time is needed to solve the Sudoku puzzles. A polynomial curve was fitted to the graph and shall be discussed further in the following section to Theory section.

5.3 Results in Relation to Theory

This section is dedicated to comparing the complexity of the experiments by the backtracking algorithm to the theoretical complexity of the algorithm that we conducted. The graph shows that the average time of the backtracking in comparison to the the number of the empty spaces. If we were to take into consideration direct cell comparisons as a basic operation, it can be shown that the time complexity is bounded above by O(n^2 n!) in the worst case. In the best case we have complexity of O(n^2). This correlates to the empirical complexity that we hypothesized.

6 Conclusion

In conclusion we can see that the analysis of Sudoku contains many complexities, and that given the enormity of the state space, the number of tests that could be performed in our alloted time was limited.

However, it is clear to see that the performance in the best and worst case of sudoku vary astronomically.

The algorithm could be improved further by sorting possible moves so as moves where only one number for a position is possible are done first. These would be guarenteed moves. And even if only two moves are possible for a position, the amount of backtracking would still be reduced - and the benefits down the line of recursive calls would be enormous. It could increase the complexity of our best and worst case slightly due to the sorting algorithm, but it would reduce the average case substantially. We implemented this program and for the interested reader the neccesary functions can be found in the Optimized.java class, which is not currently used.

Other improvements might include a dynamic programming element, which might be a structure in memory which holds moves already found. The list of moves would have to be removed and added to with recursive calls and bactraking

respectively, but is not out of the realm of possibility and may reduce complexity and increase performance in the long run.

7 Group Work Allotment

The work allotment is (generally) as follows:

Liam Pulles

- Code (80%)
- Test Design and Result generation
- Graph section
- Conclusion section

Sibusiso Mthethwa

- Code (10%)
- Results section

Tshepiso Molobi

- Code (10%)
- Introduction section
- Background theory section
- Theoretical analysis section
- Experiment section