Lab Report
Backtracking Algorithm: Sudoku puzzle Problem

Authors:
Thandekile Hlatshwayo 686441
Thandokazi Madiya 689181
Lesego Thole 565542

Abstract

The purpose of this experiment was to determine the performance of the sudoku puzzle problem by using the backtracking algorithm. This is done by virtue of comparing the empirical and theoretical analysis of the backtracking algorithm. Of course the results will be affected by a various external or internal factors, such as: data structures, size of input, etc. These extraneous variables will possibly be accounted for when analysing the results.
How this experiment was conducted was through implementing the backtracking algorithm for solving the sudoku puzzle problem. Various partially completed grids will be the input and these will be compared on the basis of time complexity. As said above, these results will be compared to the theoretical analysis of the backtracking algorithm. The outcome of these results will then prove or disprove the hypothesis.

Contents

1 Introduction

1.1 Theoretical Principles

Backtracking is a systematic approach that iterates through all possible available options within a given search space. It's a very general technique which needs to be customized for that particular application or problem. In this case, it needs to customized for our sudoku problem.
In general, we model the solution as a vector a=

$$(a_1, a_2, ...a_n)$$

, where

$$a_i$$

is selected from a finite ordered set

$$S_i$$

. Any

$$a_i$$

that is selected is a possible candidate for the solution at the time; provided that the solution has not been used for that particular space.
Taking the task at hand, being the sudoku problem, solve Sudoku one by one by assigning numbers to empty cells, the empty cells are, in this case is the solution space. Before assigning a number, we check whether it is safe to assign, ie: if the number doesn't already exist in the semi-filled grid. We basically check whether the the number is present in the current row, current column and current 3X3 subgrid. After checking for safety, we assign the number, and recursively check whether this assignment leads to a solution or not. If the assignment doesn't lead to a solution, then we try next number for current empty cell. And if none of number (1 to 9) lead to solution, we return false.

2 Group Members' contribution

Thandekile Hlatshwayo

- Implementing and downloading the SQLite;
- Compiling and preparing the lab report document;
- Research on backtracking;
- Contributed in adding samples in the database;
- Assisted in finding errors in code;
- Experiment analysis and discussion

Thandokazi Madiya

- Programming the ReadIn class
- Including the timers in the algorithm for the experiment
- Conducting the experiment
- Contributed in adding samples in the database
- Drew the graph analysis for interpretation
- Contributed in Report

Lesego Thole

- Programming the backtracking class;
- Contributed in adding samples in the database;
- Did a bulk of the lab report documentation (Experimental procedure/ methodology);
- Assisted in error detection

3 Hypothesis

Each grid (depending on difficulty of the grid) will have a variation of where the clues are given in the grid. The more clues there are in each grid, the less time it takes for the algorithm to run.
The easier levels are more likely to take less time to solve. This is due to the fact that there are less spaces and more clues in the grid.

4 Experimental Procedure/Methodology

4.1 Understanding the theoretical analysis of the Backtracking algorithm

The theoretical analysis of the backtracking algorithm is given above. From this we know that at each recursion level we make at least one choice, so the number of recursion levels is at most the number of choices. The number of recursive calls at a single level is at most the number of possible options to complete the problem. Thus we can bound the total number of recursive calls to the number of choices and the options per choice. Depending on the specific problem being solved, the backtracking algorithm outputs different worst and best cases.
In the case of the Sudoku puzzle problem, using a 2D array as representation of the entire Sudoku grid.

The worst case of the algorithm is

$$O(n^m)$$

where n is the number of possibilities for each square – 9 being the usual number, and m is the number of spaces that are blank (zeros in the array). Practically, the worst case would occur if there a many empty spaces, which means there are more possible numbers to fill in. In this case, there is more room for backtracking. Furthermore, the difficulty, number of filled in spaces and empty spaces (solution spaces) increases the complexity of solving the puzzle. There would be a lot more comparisons, backtracking and possible deletions.

The best case being

$$O(n^m),$$

with m = 1. This algorithm performs a depth-first search throughout the possible solutions. In a more practical manner, the best case would if the sudoku puzzle would be solved.

The average case analysis arises from the worst case and it follows that both are closely related depending on the value of n, usually this holds if our n is less than 10.

The theoretical analysis serves a benchmark for the experiment which we have conducted. It depicts the analysis of various problems in general. In this way, it gives us a guide to whether we have deviated from what we are supposed to test.

4.2 Deciding on what we need to measure

Clearly we want to measure the running time of an appropriate implementation of the backtracking algorithm on input sizes which have been created to reflect the best, worst and average case performance of the above algorithm. In order to study the asymptotic behaviour of the algorithm we must test different arrays with different number of 0's - (m) where m is significantly large or small.

The best case is easy because we just have to make sure that the number of blank spaces is m = 0 in the square, meaning that the square is solved. The worst case is relatively easy as well, bearing in mind the level of difficulty, and the most possible value of m. For the average case performance the theoretical analysis tells us that we should measure the running time for the algorithm to find the average growth.

4.3 Deciding on appropriate hardware and programming language

This could seem trivial, but there are a couple of factors we need to consider before selecting an appropriate programming language and hardware. We are more interested in the rate of growth of the performance of the algorithm than the actual running times displayed. There could be instances when we are actually interested in the running times on a particular machine. So the factors considered here were

Availability – doing the experiment on a machine with the required language is advantageous and one which is easily accessible to us. Also, doing the experiment on a machine that is required by the lecturer.

Ease of use – follows from point above

Familiarity – it would be advantageous to work with structures that are more familiar to the programmer, to make for better testing.

Specifics of the programming language that hinder measurements – how would the limitations of the selected programming language affect the results ? For instance, java's automatic garbage collection.

The type of database we're using – it's implementation with accordance to the programming language to be used, bearing in mind that the quickest access to data is most efficient.

Implementation of relevant data structures - the data structures to be used depend on the programming language, also considering which is the most efficient.

4.4 Deciding on appropriate Data Structures

In order to decide on what data structures to use we need to have a good understanding of the algorithm and how the choice of data structure could affect the performance of the algorithm. In some instances the difference might not really be important. In this instance, we are measuring the performance of the backtracking algorithm when implemented to solve the Sudoku problem. Saving samples of input as 2D arrays of integers proves to be most efficient for the recursion that will be called. These 2D arrays are stored as row elements (a string separated by a semicolon) in a table inside the SQL database. Each entry of each row is separated by a comma.

4.5 Implement the Algorithm

Once the decisions about hardware and programming have been made we would then implement the algorithm in the chosen language on the chosen machine. Even at this level we still have decisions to make. Clearly these decisions are at the detailed level but they could still have impacts on our measurements and we should be aware of them. Firstly, we distinguish between the level of difficulty of our different input sizes, then run the algorithm on these different input sizes to find unique solutions. Secondly a graph will be plotted to depict the behavior of the algorithm and we lastly analyse the results.

4.6 Implement some form of timing device

In this experiment we want to measure the performance of running the backtracking algorithm on the Sudoku puzzle problem – how long it takes to output a solution, if the solution exists. What we're not considering however is the time it takes to access the sample input, the overall setup, waiting time etc. Another concern is what other factors will affect the overall running time of the algorithm and how they will do so. Therefore the timing device used, in our case being the built-in system time, the above needs to be paid attention to.

Factors like the machine that is used to do the testing, the operating system and the network over which this is to be done prove to also be slightly crucial. Certainly connecting to the database in quick run-time is required, although this is not part of the algorithm itself and the timing device should definitely not consider this. To prove the reliability of the algorithm, implementing the algorithm on the same size input numerous times could also be helpful as this would prove to be more precise.

4.7 Creating the data

Creating the data for this experiment means manually adding sample input in the database in SQL and thus calling it from our code. This allows for the testing of our various cases.

Best case - here our sample input will be of a grid that is already solved.

Worst case - here our sample input should consider the placement of integers in the grid, bearing in mind the number of empty blank spaces and the level of difficulty.

Average case - this will be the general result of running the algorithm on various input sizes.

5 Results

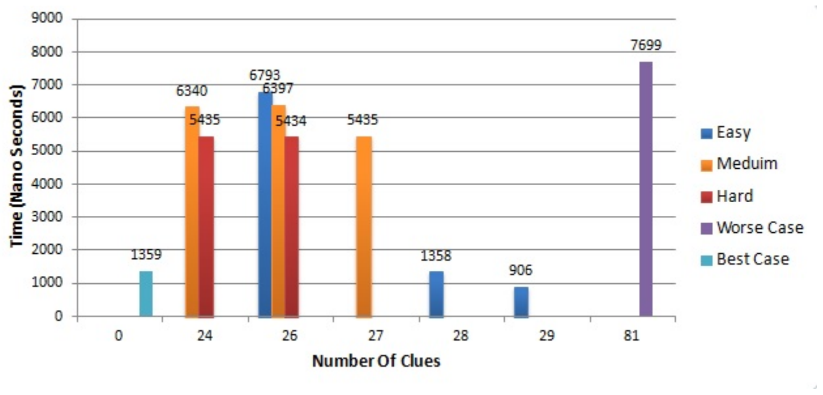

Figure 1:

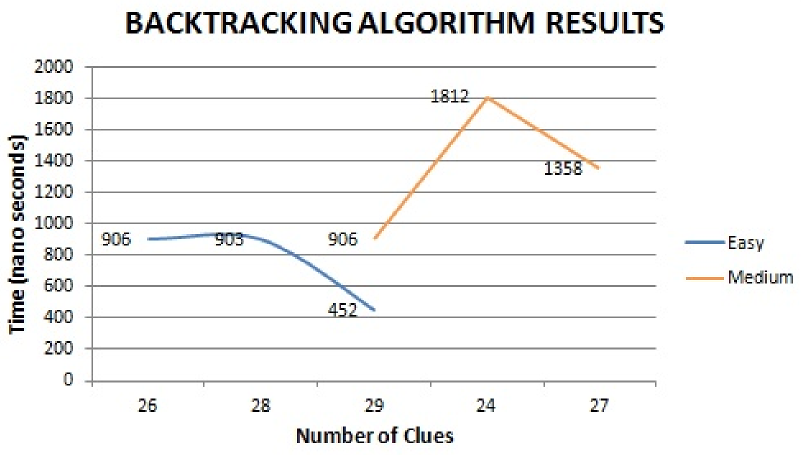

Figure 2:

5.1 Analysis

Figure 1 The number of ”clues axis” shows the amount of numbers (from 1 to 9) filled in the grid and the ”time axis” shows the amount of time taken to solve the sudoku puzzle.

Based on the results of the above bar graph; the "easy" grids (in blue) take longer to solve the solution. The "medium" grid (in orange) takes less time and even less for the "hard" grids (in red), with time values in nanoseconds as follows: 6793; 6597 and 5434, respectively for the above mentioned levels of difficulty.
Also, the worst case is when the sudoku grid is solved and completed; meaning we will make a lot more comparisons. The algorithm runs through each and every cell and subgrid and finally the entire grid to realise that the puzzle is actually complete.
The best case is when the grid is empty, ie: all the cells are empty. This means that there are no comparisons and one can input any number they wish and basically create their own solution. As a results, there will be less time taken on the solution because there will be no comparisons from the onset.
The average case varies from input sample to input sample. It can take

Figure 2 depicts two curves, each based on the level of difficulty of the sudoku samples from the database. Each curve represents the time it took for various sudoku grids to be solved. The blue curve represents the easy-type sample input, while the orange one represents the medium-type. From the graph it is clear that as the number of clues we originally have increases the time it takes to solve the grid decreases, with the major drops in the curves depicting just how crucial the number of clues is.
We see this with the "easy" grids, where it starts of at a constant slope, meaning that the grids run at a constant rate. As the numbers are being inputted, the slope declines, meaning it now takes less time. This could be due to a number of factors: the number selected for a solution happened to be the correct solution for that given solution space as the positioning of the number is convenient; also, now that there are less numbers to input, it is easier to detect the next move and next number to input.

6 Interpretation and Discussion

How we will interpret these results is through examining the time that it took for solving the sudoku puzzle which translates to a curved graph showing the growth rate and a bar graph of the growth to see whether there is a positive or negative relationship between our dependent and independent variables.
Looking at **figure 1**, it suffices to show that the more amount of inputs we have, the more time it will take to solve the puzzle. This we see with the "easy" to "hard" grids. which decrease in number of clues.
Our best case and worst case has changed. The best case, judging from the experiment, shows that the an empty grid (solution space) yields the least amount of time to solve. The worst case is then a situation when the grid is fully solved, with consequently more comparisons.
Figure 2 The blue curve (easy level of difficulty) declines as the number of clues increase, the factors contributing to that are explained above.

For the "hard" grid, it initially takes long to solve the sudoku problem and it further increases as the number of clues increase **(note: there is an error in the scaling of the number of clues on the x-axis)**. so, the the harder the difficulty level, the less time it takes to solve the sudoku puzzle.

From the analysis it is evident that the level of difficulty or rather the amount of clues given play a major role in the time complexity of the algorithm. This is because with more clues given, on average, more comparisons have to be made in the backtracking algorithm thus more time is taken to solve the grid and less clues given requires less comparisons have to be made in the backtracking and thus less time is taken to solve the grid.

7 Conclusion

In this section we state whether the hypothesis and the theoretical analysis agree or this agree with the hypothesis. It is evident that the more clues initially included in the grid, the more time it takes for the backtracking algorithm to run. Thus we reject the hypothesis.

8 References and Acknowledgements

- YouTube video lecture - COMP300E: Programming Challenges by Prof. Skiena Stevens. Department of Computer Science and Engineering. held at: Hong Kong University of Science and technology.
- stackoverflow.com

- https://cle.wits.ac.za/access/content/group/COMS3000$_{A}lgorithmsandArtificialIntelligence$ $/Assignment/Empirical/20Analysis_{2}015.pdfhttp : //www.puzzles.ca/sudoku.html$