AAA Project Report

Solving Sudoku using the backtracking algorithm

800107 - Daniel Ng See Cheong
839521 - Uzair Moolla

October 31, 2016

Contents

1 Introduction

Sudoku is a numerical based logic puzzle game. The idea is to fill in an $n \times n$ grid with numbers in a specific manner. The most commonly known grid usually consists of square blocks with 3 rows and 3 columns. These individual blocks are then arranged in a similar manner again, with 3 blocks along each row and 3 blocks along each column, producing a 9×9 matrix. An example of this described matrix can be seen in Figure 1. Each individual block needs to be filled in with one of these numbers: $1, 2, 3, 4, 5, 6, 7, 8, 9$.

The rules for filling in the 9×9 sudoku grid are as follows:

- Within each 3×3 block we can only have a single occurrence of each number.
- Along each row of 9 blocks we can only have a single occurrence of each number.
- Along each column of 9 blocks we can only have a single occurrence of each number.

An example of a Sudoku Puzzle completed using the rules stated above can be seen in Figure 2.

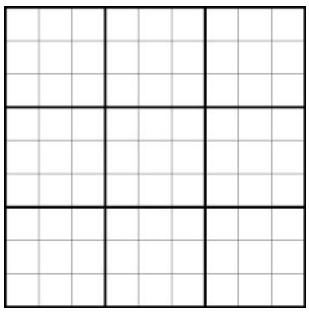

Figure 1: Sample of a 9×9 Blank Sudoku Board

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

Figure 2: Sample of a 9×9 Completed Sudoku Board

Solving sudoku puzzles are a fun exercise for many. However we would like to find a suitable algorithm which can be used to solve sudoku puzzles correctly as well as efficiently. There are various possible methods which can be used to solve sudoku puzzles. We will be focussing on the *Backtracking Algorithm* in our experiments.

2 Aims

We are going to use the backtracking algorithm to solve sudoku puzzles of size 9×9. We will attempt to solve these sudoku puzzles accurately and in as little time as possible. Once we have the algorithm working correctly we will perform both theoretical as well as empirical analysis on it. We aim to find the best and worst case complexities of the algorithm being used. These results will be obtained through both methods of analyses (empirical as well as theoretical) and correlations between the results will be obtained.

We will write code that will randomly generate unsolved puzzles using a random puzzle generation algorithm. These generated puzzles will either have a unique solution, multiple solutions or no solution. These puzzles will be used for all analysis done.

3 Summary of Theory

3.1 Backtracking Algorithm

In general backtracking algorithms are mainly used to solve NP-complete computational problems. Simply put, the backtracking algorithm starts with the selection of one possible move out of all available moves and will try solving the problem with the selected move. If the selected move allows us to find a correct final solution to the problem, the solution is given as an output (it is printed out). However, if the selected move does not lead to a correct final solution to the problem, we backtrack. The backtracking is done to the point where the move was selected and we then choose an alternative move at this point. This process is repeated until the correct solution to the problem is found. If none of the available moves work in finding the correct solution to the problem it is sufficient to say there is no solution to the problem.
Recursion is the most appropriate method to implement this type of algorithm.

4 Pseudo code

Algorithm 1 Check if a Move is Valid

```
1: function validMove(number , row, column)
2: Where:
3: number is the number to be inserted
4: row is the row of current position
5: col is the column of the current position
6: n is the length of the square matrix representing the sudoku puzzle grid
7: board is the n × n square matrix representing the sudoku puzzle grid
8:
9:     for i from 1 to n do
10:        if number is equal to board[row][i] then
11:            return false
12:    for i from 1 to n do
13:        if number is equal to board[i][col] then
14:            return false
15:
16: initialise rowmin to row/blocksize × blocksize
17:
18: initialise colmin to column/blocksize × blocksize
19:
20: initialise rowmax to (rowmin + blocksize)
21: initialise colmax to (colmin + blocksize)
22:
23:    for i from rowmin to rowmax  do
24:        for j from colmin to colmax do
25:            if number = board[i][j] then
26:                return false
27:    return true
```

Algorithm 2 Check if a Position is Open

```
1: function openPosition
2: n is the length of the square matrix representing the sudoku puzzle grid
3: board is the n × n square matrix representing the sudoku puzzle grid
4:
5:     initialise temp to be an array of length 2
6:     for i from 1 to n do
7:         for j from 1 to n do
8:             if board[i][j] is empty then
9:                 temp[0] = i
10:                temp[1] = j
11:                return temp
12:    return NULL
```

Algorithm 3 Backtracking

```
1: function SOLVE
2: n is the length of the square matrix representing the sudoku puzzle grid
3: board is the n × n square matrix representing the sudoku puzzle grid
4: openPosition(),validMove(i,row,col) are the functions described above in Algorithm
   1 and Algorithm 2 respectively.
5:
6:     initialise row and col;
7:     initialise temp to openPosition();
8:     if temp is not NULL then
9:         row = temp[0];
10:        col = temp[1];
11:    else
12:        return true
13:    for i from 1 to n do
14:        if validMove(i,row,col) is true then
15:            board[row][col]=i;
16:            if solve() is true then
17:                return true;
18:            set board[row][col] to be empty;
19:    return false;
```

5 Theoretical Analysis

Theoretical analysis will be done for each of the algorithms listed above. Even though the first two algorithms are used in the Backtracking Algorithm (ie Solve calls validMove as well as openPosition) we will be calculating their complexities individually and thereafter include them in the complexity calculation of the Backtracking Algorithm. We are able to do this as the aim of this experiment is to find the complexity of the Backtracking Algorithm being used to solve a sudoku puzzle.

5.1 Algorithm 1: validMove

The algorithm validMove is given the number to be inserted as well as the row and column where it will be inserted in the sudoku puzzle. It then runs through the puzzle being filled in and checks if the three rules of sudoku listed above are not violated by the insertion of the particular number at that position.

For the theoretical analysis of this algorithm we will use comparisons as the basic operation. There are two main segments in this algorithm, the first simply checks if the number to be inserted in the puzzle already exists in the row or column of the block wherein we want to insert the number. The second segment checks if the number to be inserted exists in the sub-block of the sudoku puzzle as mentioned in the rules of sudoku.

5.1.1 Best Case

In the best case this algorithm would exit at line 11. This would happen when the number in position [r][1] is the same as the number to be inserted into row r. This means the time complexity of Algorithm 1 in the best case is $O(1)$.

5.1.2 Worst case

In the worst case the algorithm would exit at line 27. This would happen when all the rules are valid and the number can be inserted into the position being checked. Therefore the time complexity of this algorithm in the worst case is $O(n)$ for a sudoku board of size n.

5.2 Algorithm 2: openPosition

The algorithm openPosition does not require any inputs. It then runs through the sudoku puzzle being filled in and searches for blocks (positions) which are empty in the puzzle being filled in. The empty blocks are represented by 0 in the puzzle.

For the theoretical analysis of this algorithm we will use comparisons as the basic operation. This algorithm simply loops through all positions in the sudoku puzzle searching for an empty block wherein we can fill a number according to the rules of sudoku.

5.2.1 Best Case

In the best case this algorithm would exit in line 11. The best case complexity will occur when the open position is the first block searched in the puzzle i.e. position [1][1] in the matrix. Therefore the complexity of this algorithm in the best case is $O(1)$.

5.2.2 Worst Case

In the worst case the algorithm has two possibilities. The first is if the board only has one last open position i.e. [n][n] is open for a board of size n. The second is if there are no open positions left on the board. In both cases the complexity is $O(n^2)$. Thus the time complexity of this algorithm in the worst case is $O(n^2)$ for a sudoku board of size n.

5.3 Algorithm 3: Solve

The algorithm Solve does not take in any input. It is the backtracking algorithm we are using to solve sudoku puzzles. The algorithm initialises row and column and finds an open position on the board using the openPosition algorithm. It then tries to fill in a number using the validMove algorithm. If this does not work it backtracks and reattempts to solve the sudoku puzzle. This process is repeated until a final unique solution is found. However, if that is not the case we then conclude that there is no solution to the puzzle.

5.3.1 Best Case

In the best case this algorithm would exit in line 12. The best case complexity will occur when there are no open positions in the puzzle. This implies that the puzzle is already solved. For this we need to take into account the complexity of openPosition in this situation. The board in this situation would be a completely full. Therefore the complexity of this algorithm in the best case is $O(n^2)$ for a sudoku board of size n.

5.3.2 Worst Case

In the worst case The algorithm would have to use the algorithms openPosition as well as validMove on every choice made. We also have to consider the recursion which takes place in line 16. Taking all this into account we see that the algorithm would have a worst case complexity of $O(n^m)$ where m is the number of blank or empty blocks in the initial sudoku puzzle of size $n \times n$. The reason for this is the recursive nature of the backtracking algorithm. It also represents every possible permutation of the puzzle.

6 Experimental Methodology

For our experiments we have written code that will generate a sudoku puzzle of a given size n and fill in p random positions within the puzzle with random numbers between 1 and n. The number of positions filled in, p, can be changed as it is taken in as an input to the generator algorithm. We will then run our backtracking algorithm, *Solve*, on this generated puzzle and the output, in our case, is the time in μ seconds. This process will be repeated 1000 times for each random puzzle generated. An average over the 1000 runs is used to produce a final time for solving a puzzle where the number of filled in positions p is varied. The variable p will be between 1 and 81 in our experiments, because the size of the sudoku puzzle being used is 9×9.

The reason for testing our algorithm in this manner is to find pattern in the run times. We can expect to see something of the form n^{81-p}. We also have to note that uniform randomness and the nature of sudoku solutions may affect this.

7 Presentation of results

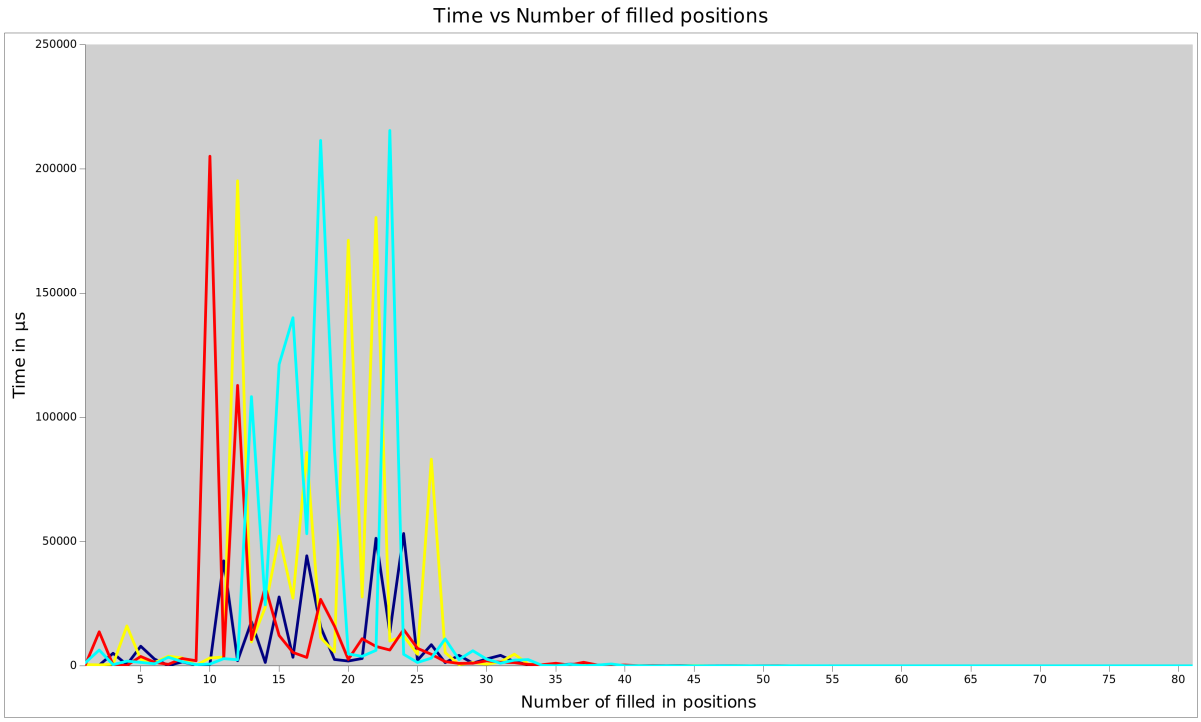

Figure 3: Results of empirical analysis

8 Interpretation of results

In Figure 3 each plotted line represents a single test. Each test was performed as explained in our Experimental Methodology. From the results we see that for f between 1 and 10 the sudoku puzzle is solved in a relatively short amount of time. A possible reason for this is that as the puzzles are randomly generated there may be multiple solutions to each puzzle.

Also in Figure 3 We see that for f between 10 and 30 solving the sudoku puzzle takes the greatest amount of time. The reason for this is that, as some mathematicians theorise, a sudoku puzzle has a unique solution when a certain number of blocks have been filled in. This would imply that at these points the algorithm searches for the unique solution backtracking the most. This would happen often as the backtracking would occur regardless of whether a unique solution is found or not. This would be the worst case time complexity of our algorithm.

The best case complexity occurs when f is greater than 45. The reason for this is that the number of open positions start decreasing at this point and thus the algorithm has much less work to do.

9 Conclusion

We set out to determine the time complexity of the backtracking algorithm we designed to solve a sudoku puzzle of size 9×9. The algorithm does find solutions to every solvable sudoku puzzle given to it as an input. We have also been able to determine the best as well as the worst case complexities both theoretically and empirically. The complexities we have obtained through empirical analysis do align with those obtained in theoretical analysis. All in all this research has been a success as we have been successful in completing the aims as set out when we began this project.

10 Statement of Effort

All work was split as follows:

Student	**Amount of work done**
Daniel	50%
Uzair	50%

11 References

1. http://algorithms.tutorialhorizon.com/introduction-to-backtracking-programming/
2. https://en.wikipedia.org/wiki/Backtracking
3. https://www.quora.com/What-is-backtracking-in-algorithms
4. www.websudoku.com/
5. https://en.wikipedia.org/wiki/Sudoku
6. https://www.sudokuoftheday.com/howtostart/
7. https://www.sharelatex.com/