UNIVERSITY OF THE
WITWATERSRAND,
JOHANNESBURG

University of Witwatersrand
Johannesburg, South Africa

Advanced Analysis of Algorithms Assignment

Rylan Perumal, Tamlin Love, Jia Hao Huo

1396469, 1438243, 1475365

Analysis of the Backtracking Algorithm in the solving of Sudoku problems

---

Semester 2, 2018

1 Introduction

Backtracking is a very important family of algorithms used to solve a variety of problems. It operates by incrementally "guessing" a partial solution to a problem. As soon as a partial solution leads to an incorrect solution, the algorithm "backtracks" to an earlier partial solution and attempts other options, much like a depth first search.
Backtracking algorithms are incredibly powerful tools, as they are guaranteed to find a solution to an appropriate problem, given that the problem is finite and that a solution exists. They are generally much faster than brute force approaches, as they can rule out partial solutions.
In this project, we will apply the backtracking algorithm to solve Sudoku problems. In a Sudoku problem, an $n^2 \times n^2$ grid is divided into n^2 sub-grids of size $n \times n$. Commonly, $n = 3$, and thus we consider only this case. Each space in the grid is given a number, to which the following restrictions apply:

- A number may not appear more than once in any row
- A number may not appear more than once in any column
- A number may not appear more than once in any sub-grid

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

Figure 1: An example of a Sudoku puzzle, where enough clues are given such that the solution is unique [Watanabe, 2013]

We can clearly see that this is the kind of problem in which backtracking algorithms excel. Our strategy is to incrementally fill in the squares, ensuring that the restrictions above are met. If at any stage the algorithm encounters a situation in which no guess is valid, it will backtrack and attempt other guesses.

2 Theoretical Analysis

In this analysis, we will compute the algorithm's complexity with respect to k, the number of 0 entries (i.e. blank entries) in the initial problem. We keep the size of the grid constant at 9×9, and thus our complexity will not vary with respect to the size of the grid.
We will consider stack operations (e.g. *stack.push()* and *stack.pop()*) to be the basic operations. This is equivalent to selecting changes in the state of the grid (i.e. our guesses) as our basic operation.
We begin by providing a brief pseudocode representation of our implementation of the backtracking algorithm.

Algorithm 1 Backtracking Sudoku Solver

```
1:  procedure SOLVE(sudoku)
2:      stack ← []
3:      num ← 1
4:      i, j ← 0
5:      while not isComplete(sudoku) do
6:          if j mod 9 = 0 and j ≠ 0 then
7:              i ← i + 1
8:              j ← 0
9:          current ← sudoku[i][j]
10:         if current.num = 0 then
11:             if checkRCS(i, j,num,sudoku) = false and num < 10 then
12:                 current.num ← num
13:                 stack.push(current)
14:                 num ← 0
15:                 j ← j + 1
16:             else
17:                 if num > 10 then
18:                     if stack.notEmpty()=true then
19:                         current ← stack.pop()
20:                         i ← current.i
21:                         j ← current.j
22:                         num ← current.num
23:                         current.num ← 0
24:                     else
25:                         break
26:         else
27:             j ← j + 1
28:             num ← 0
29:         num ← num+1
30:     return sudoku
```

In the above algorithm, the *isComplete(sudoku)* function checks the grid, *sudoku*, for any 0 entries, returning true if there are no 0's present and false otherwise. The *checkRCS(i,j,num,sudoku)* returns true if *num* occurs in the current row, the current column or the current square. Otherwise, it returns false. Both these functions do not depend on the number of zero entries (k) and

thus are $\mathcal{O}(1)$. We can therefore safely ignore them for the remainder of this analysis.
Our first consideration must be of the Best Case. This occurs when our initial guess (for this algorithm, 1) is correct (for $k > 9$, we extend this case to encompass other early guesses). In this case, the if-statement on line 11 always evaluates as true when it is reached, and thus the value of *num* never exceeds 1. Since the aforementioned if-statement only executes if *current.num*$= 0$, which is only true for k spaces in the grid, the stack operation on line 13 runs only k times, and thus the Best Case complexity is $\mathcal{O}(k)$.
Conversely, the Worst Case occurs when our final guesses are correct, thus forcing the algorithm to perform maximal backtracking. In this case, the algorithm reduces to a brute force approach. The complexity is thus $\mathcal{O}(9^k)$.
The Average Case is not trivial to compute theoretically, and is beyond the scope of this paper. We know for sure that the average case has a complexity between $\Omega(k)$ and $\mathcal{O}(9^k)$. We hypothesise that the average case time complexity of the algorithm is $\mathcal{O}(C^k)$ for some $1 < C \leq 9$, the average number of guesses per backtrack. We will investigate this hypothesis further through empirical analysis.

3 Empirical Analysis

3.1 Data Processing

Before we could conduct any meaningful experiments, we required a large dataset of Sudoku problems. We sourced 50 Sudoku problems online, whose solutions exist and are unique [Fontaine, 2012]. We then augmented the size of our dataset by solving each problem and subsequently "unsolving" then by removing random elements. We started off be removing 1 random element up to removing 64 random elements from each solved puzzle. The reason we removed up to 64 elements is that the minimum number of clues required for the solution of the solved puzzle to be unique is 17 [Gary McGuire, 2013]. Thus we have generated 3200 new puzzles to be solved.

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

Figure 2: Example unique solution 1

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

Figure 3: Removing 1 value i.e Order 1

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

Figure 4: Removing 32 values i.e Order 32

4								
								1
			8					
	4		1					
			5	6	4	1		8
1		6						
							6	
	9	5					8	2

Figure 5: Removing 64 values i.e Order 64

3.2 Experiment

Graphing the execution time of the algorithm against the number of zero entries (k), we arrive at the following.

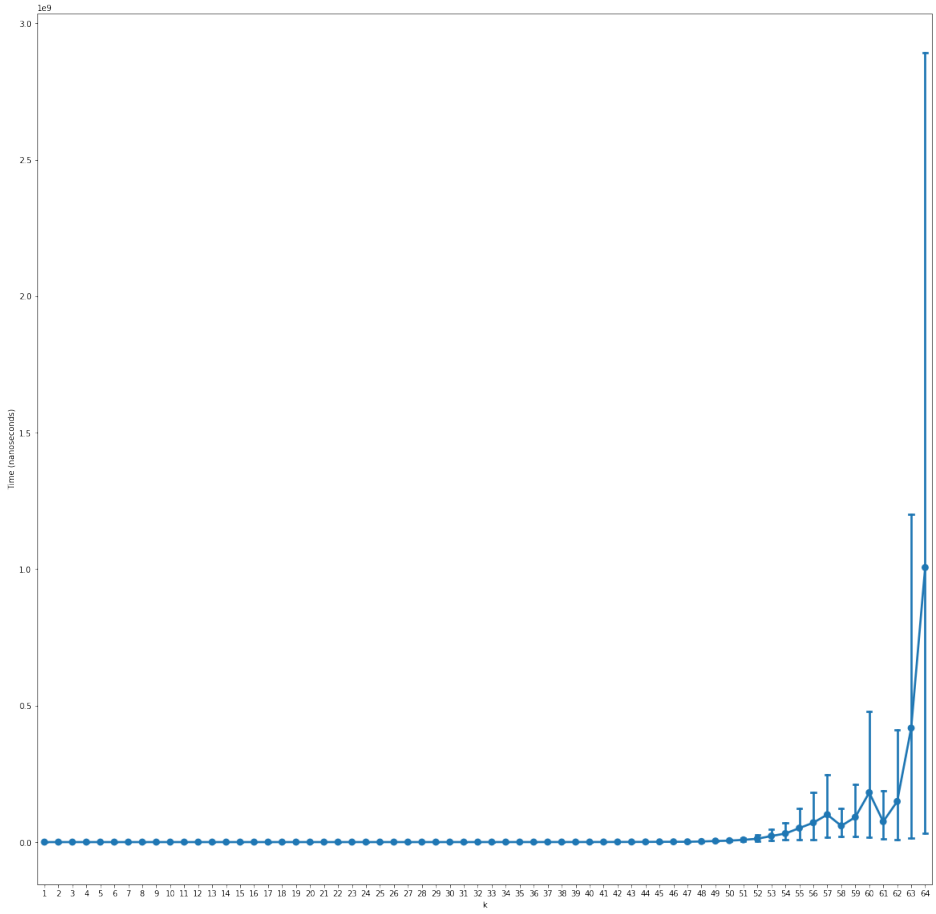

Figure 6: Time (in nanoseconds) vs. Number of 0 entries (k)

The following statistical figures relate to this data-set

Count	3200
Mean	0.03589 seconds
Standard Deviation	0.90084
Minimum	0.00000119 seconds
25%	0.00003279 seconds
50%	0.00008533 seconds
75%	0.00054657 seconds
Maximum	45.85793 seconds

Graphing, instead, the number of stack operations against k, we arrive at the following figure.

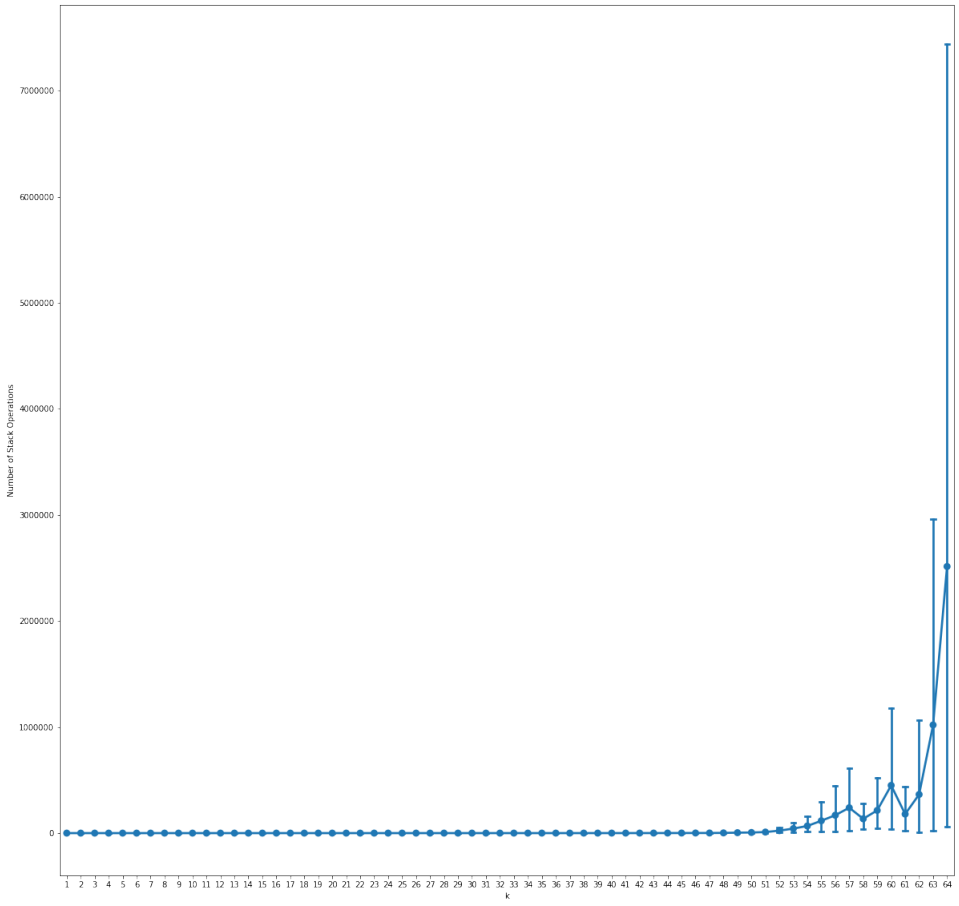

Figure 7: Number of stack operations vs. Number of 0 entries (k)

The following statistical figures relate to this data-set

Count	3200
Mean	86941.75 operations
Standard Deviation	2260370
Minimum	1 operation
25%	17 operations
50%	53 operations
75%	477.5 operations
Maximum	115201800 operations

4 Discussion

Our first remark when comparing figures 6 and 7 is that they have incredibly similar shapes. Thus the number of stack operations closely correlates with the time taken for the algorithm to execute. This indicates that our choice of basic operation at the beginning of Section 2 was pertinent.
Our second observation is to note that the shape of these graphs match up closely with the shapes of functions of the form C^x (see figure 8), for some constant C. Our algorithm is bounded above by the worst case, $\mathcal{O}(9^k)$. It therefore makes intuitive sense that the average case of our algorithm would be some $\mathcal{O}(C^k)$ for $1 < C \leq 9$, supporting our theoretical analysis.

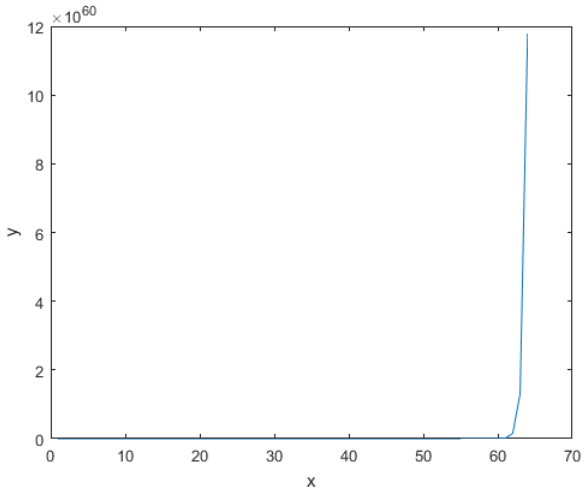

Figure 8: Graph of $y = 9^x$

Unfortunately, however, the difficulty of a Sudoku problem can have massive effects on the results for large k (See the vertical variation for larger k values in figures 6 and 7). Quantifying the difficulty is not trivial, and depends on one's definition [Watanabe, 2013]. Nevertheless, if one considers mean values for each k, then for large data-sets the difficulties of the individual problems should average each other out. In that case, our discussion above remains valid.

5 Conclusion

From our theoretical and empirical analysis, we conclude that the average complexity of our backtracking Sudoku-solving algorithm is $\mathcal{O}(C^k)$ where $1 < C \leq 9$. Further experimentation is needed to determine the value of C.

References

[Fontaine, 2012] Fontaine, D. (2012). Solving every sudoku puzzle. https://github.com/dimitri/sudoku.

[Gary McGuire, 2013] Gary McGuire, Bastian Tugemann, G. C. (2013). Solving the sudoku minimum number of clues problem. https://arxiv.org/abs/1201.0749.

[Watanabe, 2013] Watanabe, H. (2013). *Difficult Sudoku Puzzles Created by Replica Exchange Monte Carlo Method*. Arxiv - Cornell University. https://arxiv.org/abs/1303.1886.