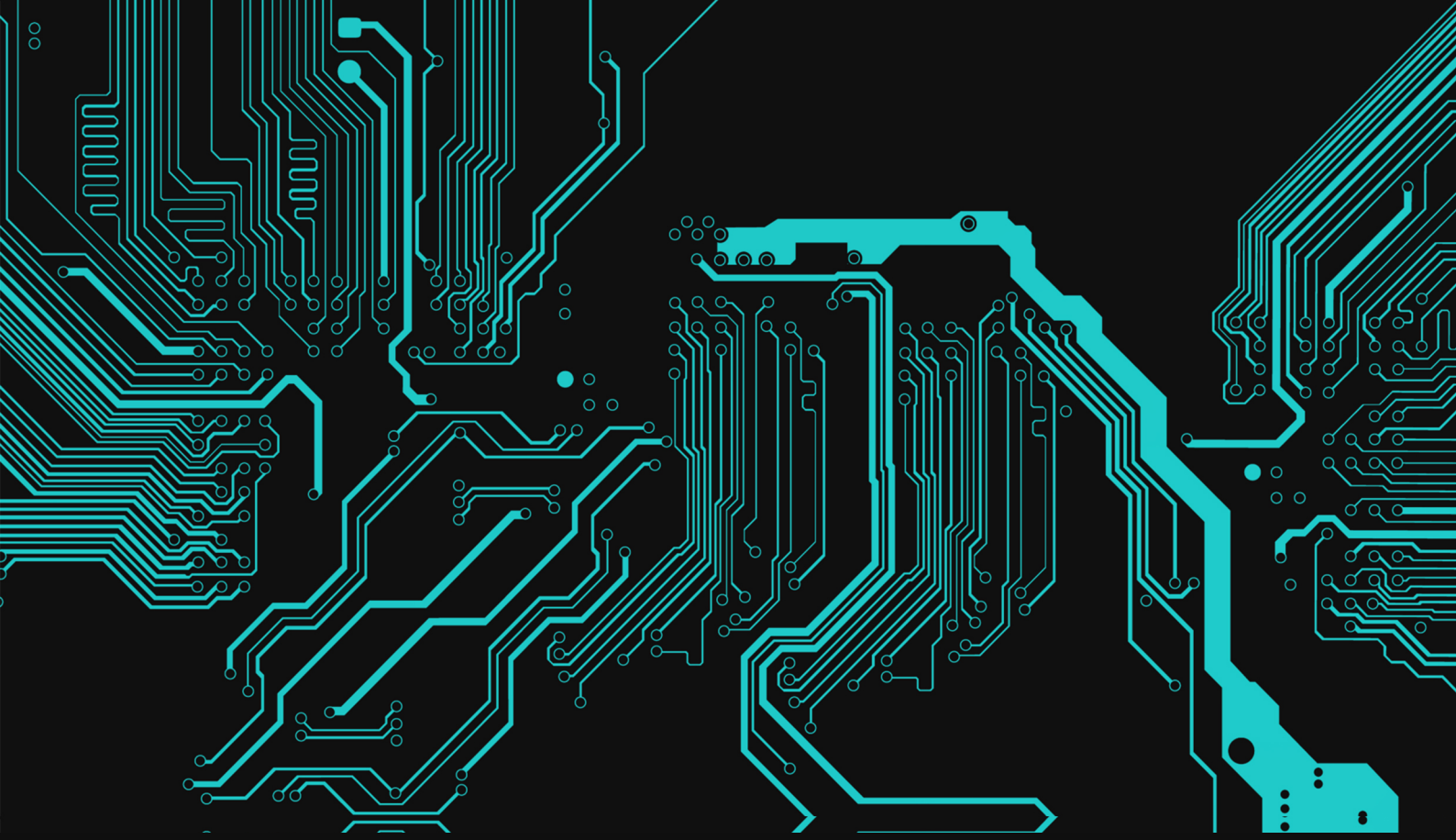
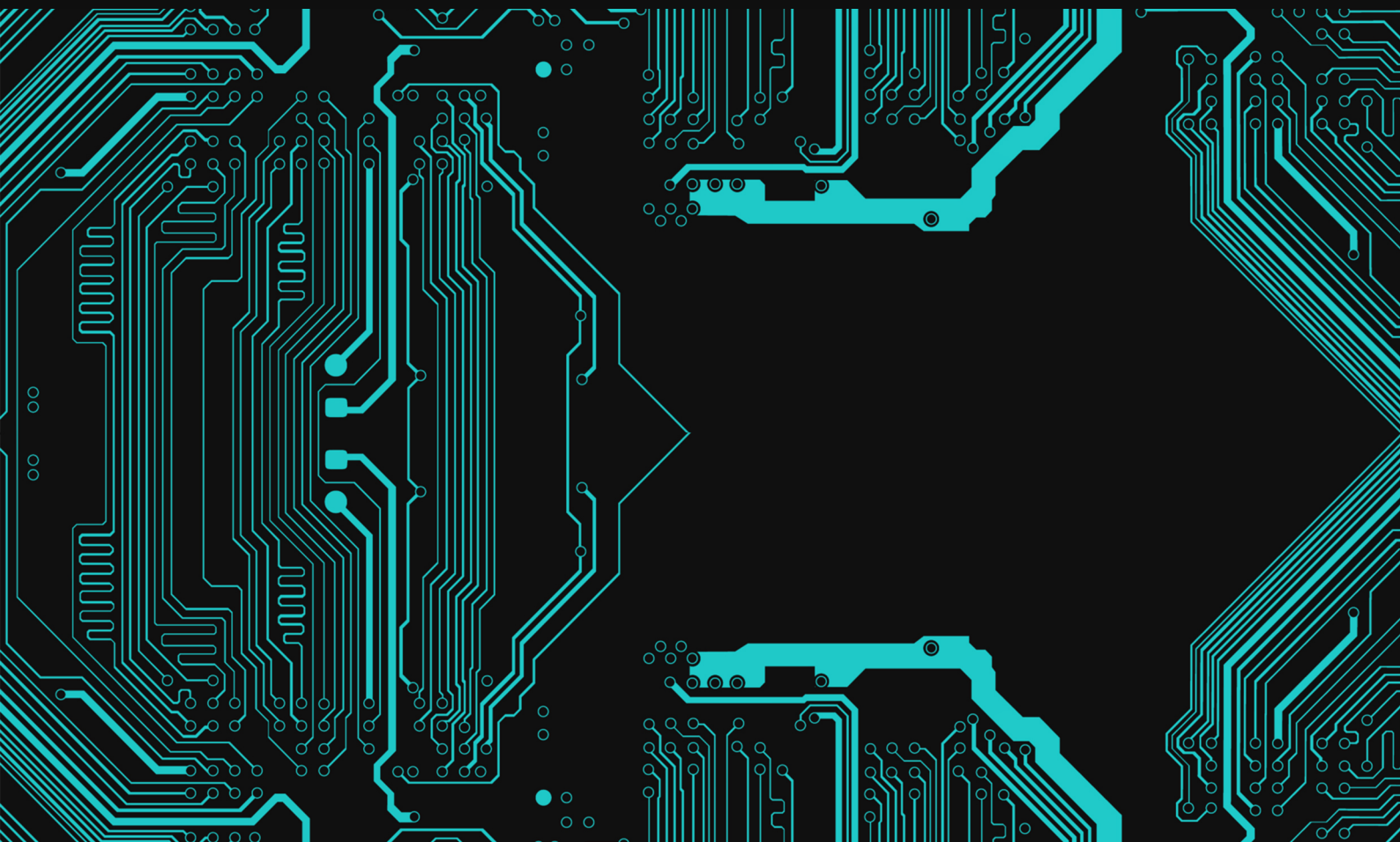


AAA Project

A Sudoku Back-tracker

Jadon Manilall 815050
Nivek Ranjith 802119



Contents

Part One

1. Introduction

1.1 What is Sudoku

Something which not many people know is that Sudoku originated in Switzerland, with Leonhard Euler credited as being the man who created this puzzle [1], which is now known the world over, and played by all types of people, not only those who are mathematically inclined.

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

Figure 1.1: Example of a solved Sudoku Grid

A Sudoku game consists of a 9x9 grid of numbers, where each number belongs to the range 1-9. Initially a subset of the grid is revealed and the goal is to fill the remaining grid with valid numbers. The grid is divided into 9 boxes of size 3x3. Sudoku has only one rule and that is that all regions,

[1] http://www.sudokudragon.com/sudokuhistory.htm

that is rows, columns, and boxes, contains the numbers 1-9 distinctly.In order to be regarded as a proper Sudoku puzzle it is also required that a **unique solution** exists, a property which can be determined by solving for all possible solutions.

R Soduku is a *NP* complete problem

1.2 What is Backtracking

Backtracking is a methodical way of trying out various sequences of decisions, until you find one that “works”.
Recursion is the key in backtracking programming. As the name suggests we backtrack to find the solution. We start with one possible move out of many available moves and try to solve the problem; if we are able to solve the problem with the selected move then we are done with the solution, otherwise, we will have to "backtrack" and select some other move and try to solve it using that move. And so on...
An idea of the general flow of a back tracking algorithm is demonstrated in the following image...

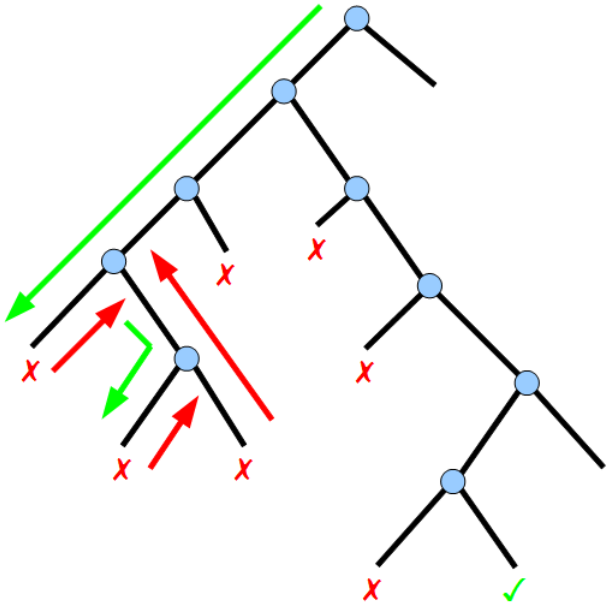


Figure 1.2: Flow of backtracking algorithms

Backtracking is an important tool for solving types of problems known as constraint satisfaction problems. These problems include crossword puzzles, verbal arithmetic, Sudoku, Mazes and many other puzzle type problems[2]

1.3 Aims

In this project, we aim to implement a backtracking algorithm to a partially solved Sudoku grid to find a unique solution.
For the remainder of this project we will be concerned with a typical 9x9 Sudoku grid, and consider different numbers of clues present in the grid to simulate difficulty.

[2]https://en.wikipedia.org/wiki/Backtracking

1.4 Terminology

"Grid" - A 9x9 2D Array of integers that holds the current state of the Sudoku puzzle
"Box" - A 3x3 Sub-grid of the Grid. Each box must have distinct integers from 1 to 9
"Empty Cell" - A cell in the partially solved Sudoku grid that has not been assigned a value
"Clue" - A cell in the partially solved Sudoku grid that HAS been assigned a value

II

Part Two

2. Analysis of Back Tracking Algorithm

2.1 Problem Statement

A Sudoku puzzle has many cases with regards to its positions of numbers as well as the number of clues provided. This experiment will be testing the performance, This includes the Time taken to solve a puzzle and the number of Backtracks needed, of the Recursive Backtracking Algorithm by solving different cases of 9x9 Sudoku puzzles.

2.1.1 Experimental Design

This experiment will consist of the following:

- A Recursive Back-tracking Algorithm implemented in Java 7(Eclipse).
- A Text File called "sudoku.txt" , which will contain our database of partially completed Sudoku puzzles. A '0' in any grid marks an empty cell
- Dell Laptop (i5 Processor, 8gb RAM, 2.6 Ghz)

2.1.2 Hypothesis

- The performance of the Back-tracking Algorithm will decrease as the number of empty cells increases.
- The number of Backtracking needed will increase as the number of empty cells increases.

2.2 Solution Technique

Algorithm 1 Sudoku Back-tracker

00: **input**: Partially solved grid, count
01: **output**: true/false

```
1:  procedure SOLVE
2:      int[] temp ← FindEmpty()
3:      if temp! = null then return false
4:          int row ← temp[0]
5:          int col ← temp[1]
6:          for <i=1,i<N,i++> do
7:              if isSafe(row,col,i) == True then
8:                  grid[row][col] ← i
9:                  if solve() == True then return true
10:                 grid[row][col] ← 0
11:                 count++
12:     else return true
```

The Pseudo-code was derived from the solve() function in our solvePuzzle.java class

2.3 About the code - How does it work?

The Algorithm presented above is our approach to implementing a backtracking algorithm to solve a partially solved 9x9 Sudoku grid.

Input to the algorithm is a 2D array of integers and a count variable to keep track of the number of backtracking operations performed for a particular input grid.

2.3.1 Line by line...

First, Line 2 finds any available empty cells in the current grid and returns their positions to the temp array. It accomplishes this by calling the FindEmpty() function.

Line 3 enables the backtracking. It checks if temp is not null(ie. There are still empty cells in the current grid), if it is, that means we still have work to do... If not then we are done(ie. return true line 12)

Lines 4 & 5 assign the row and column indexes of the empty cell, stored in the temp array, to the row and col integer variables respectively

The for loop, on line 6, allow us to try every value between 1 and 9 for any given empty cell

Line 7 Checks if it is safe to assign the value **i** to the current row and column,(ie. If all the rules of the game permit)

Line 8 simply assigns the value of **i** to the current grid at position row and col.

Line 9 Line enables the recursion, it Recalls solve, and if there is no unassigned cells left in the current grid, meaning temp will be null and thus it will break out of the procedure and return True. If not, then Line 10 is executed which sets the value of the grid at row and column back to 0. This means we that the value **i** that we assigned to row and col does not work so we need to try the next one...

Line 11 Simply increments the count variable to keeps count of the number of backtracking operations performed on a specific grid.

Line 12 Returns true if the condition on line 3 fails, that is, if there are no more empty cells in the grid.

2.4 Theoretical Analysis

Let us now analyze this algorithm. To determine the amount of work the algorithm does we must first identify a basic operation which will give us a good idea of the amount of work done and then we must determine how many times this basic operation is executed. In this algorithm, although there is a lot going on we will define the basic operation as incrementing the count variable as this will give us a good approximation of how much of work our algorithm does irrespective of the underlying hardware and directly reflects the number of times our backtracking algorithm actually back tracks for any given valid input.

2.4.1 Worst case

The scenarios for the worst is a Sudoku puzzle that is to be solved **uniquely** which contains 64 empty cells(17 clues)

Line 3: This if statement checks if there any empty cells available, thus $O(1)$

Line 6: This for loop runs N times, where N is the number of rows/columns.

Line 7: The isSafe() function within the for loop in Line 6 calls 3 other functions all with the complexity of N , thus adding up to $N+N+N=3N$. Including the outer if statement: $N+N+N+1=3N+1$

Line 12: This If statement returns true if the Sudoku puzzle is solved , Thus a complexity of $O(1)$

Line 3: This returns false, if the Sudoku puzzle is not completely solved , thus $O(1)$.

Since this is a recursive algorithm , when the code reaches Line 9 ,recursion occurs. The worst case is for the algorithm to recur $O(N^2)$ times.
Therefore if we let $T(N)$ be the function representing the number of basic operations our algorithm performs for input N, We can derive complexity for the worst case as follows:

$$\begin{aligned} T(N) &= (1*N(3N+1+1+1))^{N^2} \\ &= (1*N(3N+3))^{N^2} \\ &= (3N^2+3N)^{N^2} \\ &\approx O(N^{2N^2}) \end{aligned} \tag{2.1}$$

Thus, the worst case is $O(N^{2N^2})$

2.4.2 Best Case:

The scenario for the best case is an already solved Sudoku puzzle i.e. Number of empty cells are zero.

Line 3: This if statement checks if there are any empty cells available, since the best case has no empty cells available , it would go to Line 12 and return true, thus the complexity for the best case is $O(1)$

2.5 The Implementation

Our solution consists 2 Java classes called getPuzzle.java and SolvePuzzle.java as well as a database of partially solved sudoku puzzles stored in a text file called sudoku.txt

2.5.1 Data structures used

A 2D Array of Integers is used to store and manipulate the grid.
We chose this data structure as it was optimal in our case. Since the size of the grids are known and we're accessing the data in the grid so frequently it helps that it takes constant time ie. $O(1)$ to access an element using an array, Where as with a linked list or an ArrayList it would take around $O(N)$
So by using an array of integers we have the advantage of $O(1)$ access time.

2.5.2 Sudoku.txt

Sudoku.txt contains a database of 54 Partially solved sudoku puzzles of varying difficulties.

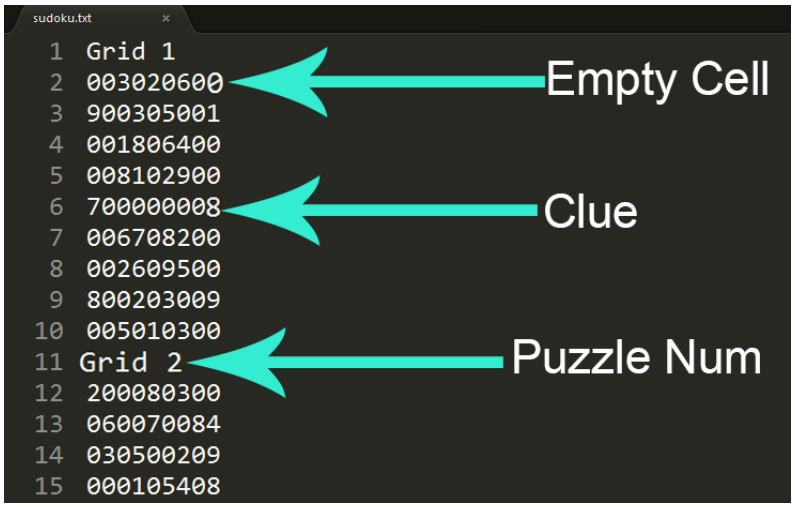


Figure 2.1: Screenshot of sudoku.txt

2.5.3 getPuzzle.java

getPuzzle.java was written as a helper class for the SolvePuzzle class. getPuzzle, takes in a path to sudoku.txt and an integer specifying a puzzle number (between 1 and 54) and returns a 2d String array representing the puzzle (shown in fig2.1)
This string can then be converted to an 2d integer array.

2.5.4 SolvePuzzle.java

This class is used to solve the grid. It contains 7functions to help with this task.
First, the findEmpty function scans through the grid looking for any empty cells.

When the solve function is called on a grid, it then calls the isSafe function, Which in turn calls 3 other functions; checkRow, checkCol and checkGrid. These functions take in a row, col and number and return a boolean indicating weather it is is safe (ie. if the rules of sudoku allow) to insert the number in the grid at position row and col.
The last function is the print function, this functions just takes in a grid prints it out to the console.

- **Sample input :**
  Console : Choose puzzle between 1 & 54
  User: 54
- **Sample output:**
  Empty cells : 64.0
  Clues : 17.0
  Percentage partially solved: 20.98765432098765

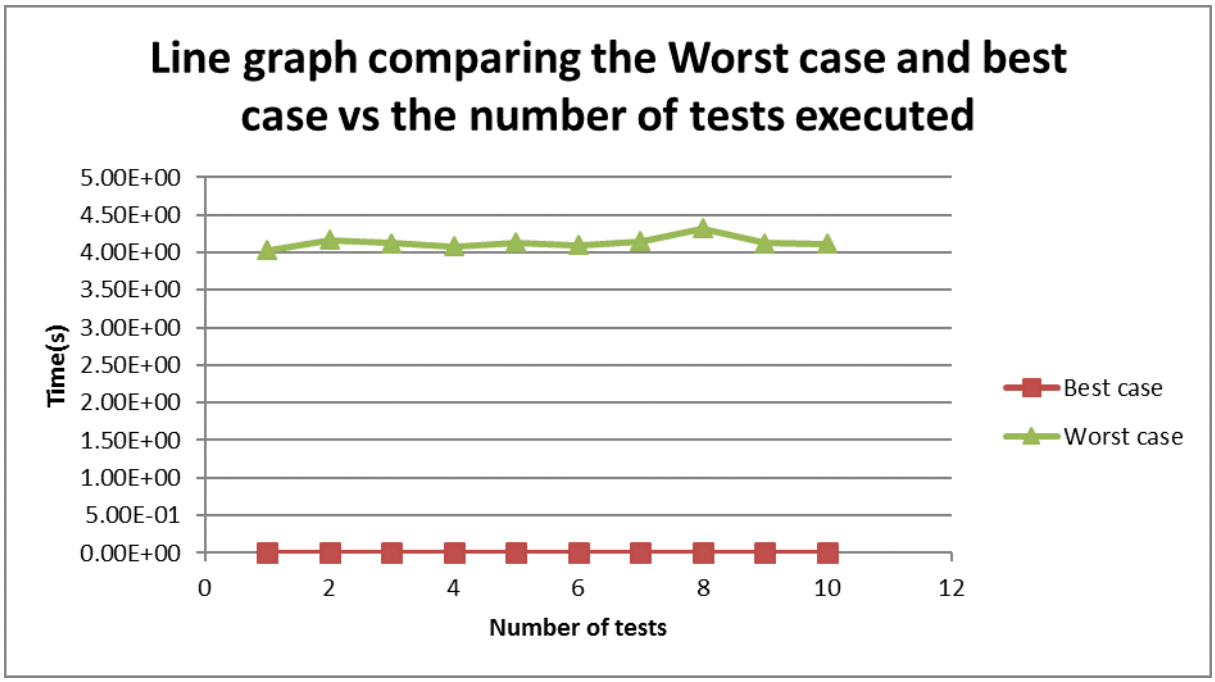
$$\begin{bmatrix} 0 & 0 & 0 & 7 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 3 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 6 \\ 0 & 0 & 0 & 5 & 0 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 4 & 1 & 8 \\ 0 & 0 & 0 & 0 & 8 & 1 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 & 5 & 0 \\ 0 & 4 & 0 & 0 & 0 & 0 & 3 & 0 & 0 \end{bmatrix}$$

  Solving...

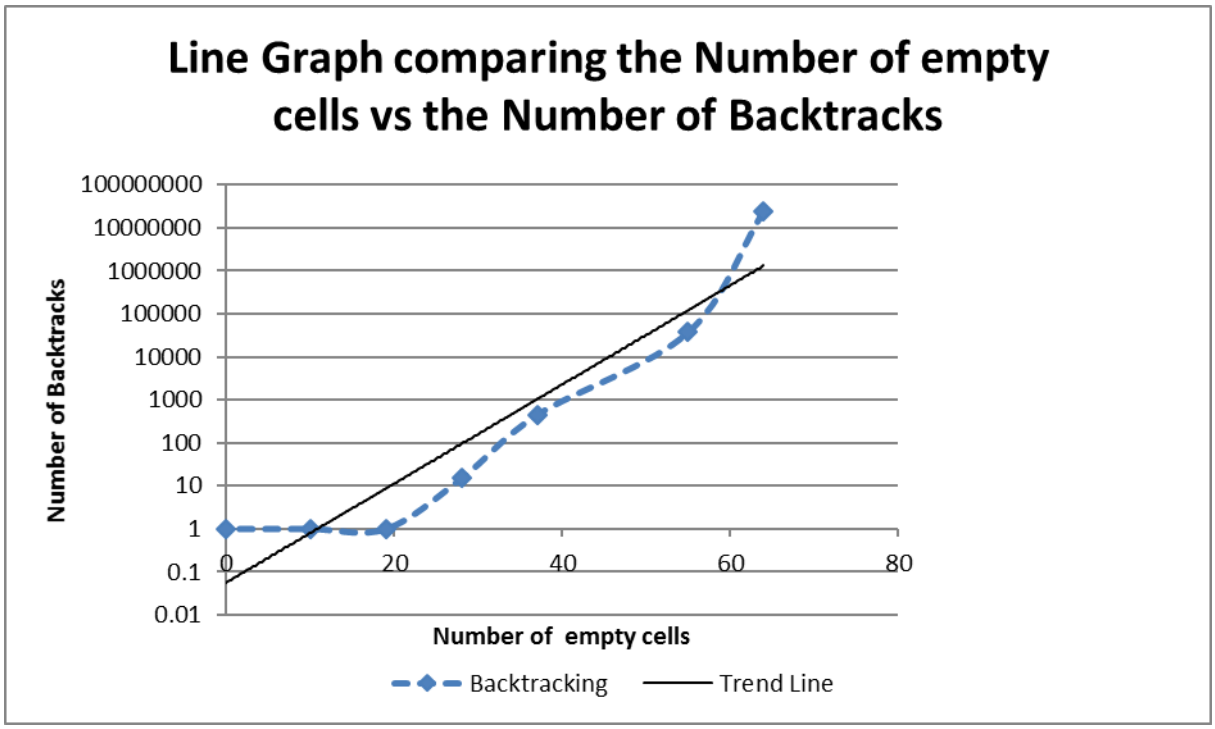
$$\begin{bmatrix} 2 & 6 & 4 & 7 & 1 & 5 & 8 & 3 & 9 \\ 1 & 3 & 7 & 8 & 9 & 2 & 6 & 4 & 5 \\ 5 & 9 & 8 & 4 & 3 & 6 & 2 & 7 & 1 \\ 4 & 2 & 3 & 1 & 7 & 8 & 5 & 9 & 6 \\ 8 & 1 & 6 & 5 & 4 & 9 & 7 & 2 & 3 \\ 7 & 5 & 9 & 6 & 2 & 3 & 4 & 1 & 8 \\ 3 & 7 & 5 & 2 & 8 & 1 & 9 & 6 & 4 \\ 9 & 8 & 2 & 3 & 6 & 4 & 1 & 5 & 7 \\ 6 & 4 & 1 & 9 & 5 & 7 & 3 & 8 & 2 \end{bmatrix}$$

  Time taken :4.141131979 (sec)
  Number of backtracks: 24396642

2.6 Results and analysis

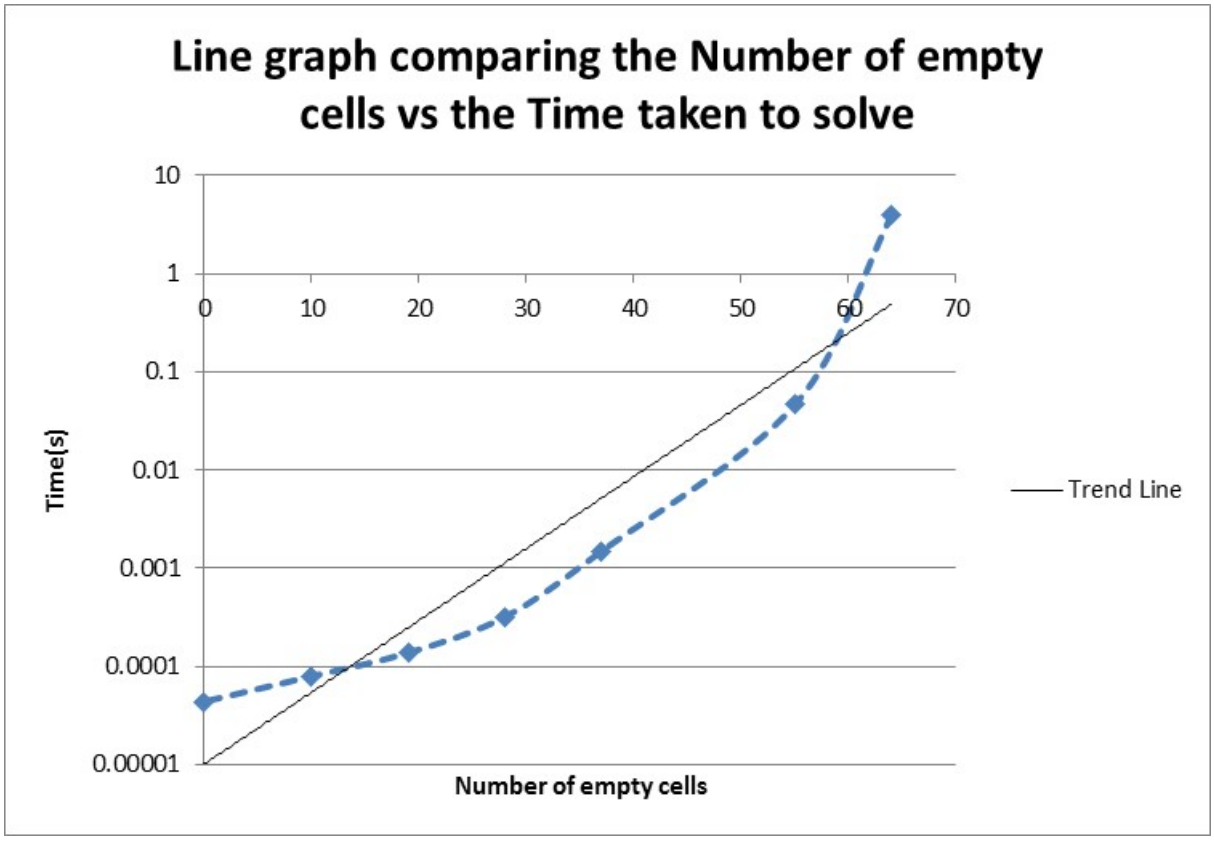


The graph above compares different scenarios of both the best and worst case. As you can see the best case(Already solved grid) runs much faster than the worst case(64 empty cells in a grid).



The graph above compared the number of empty cells with the number of backtracks needed to

solve a Sudoku puzzle. It is clear in the graph that as the number of empty cells increases the the number of backtracks needed increases exponentially.



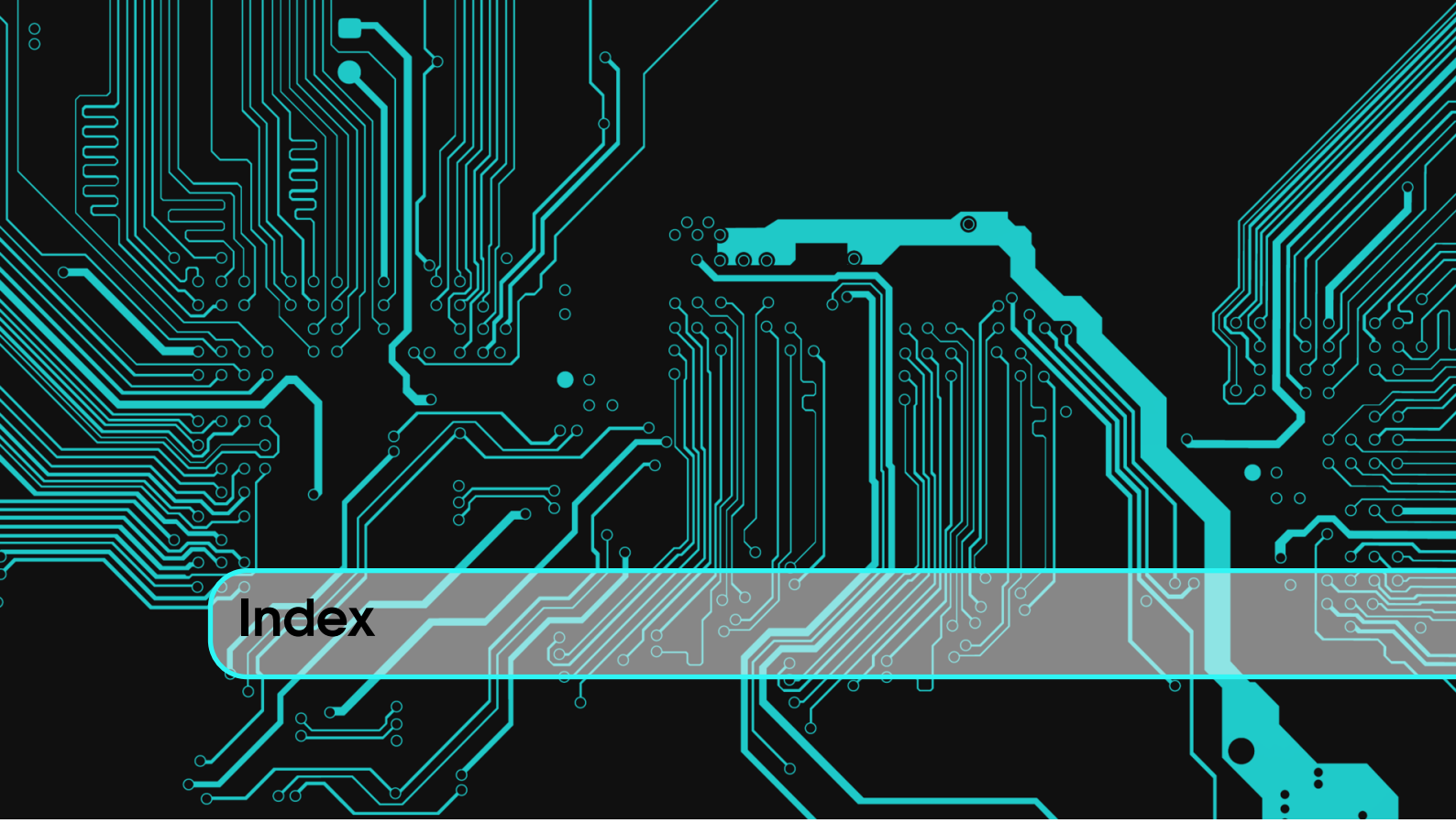
The graph above compared the number of empty cells with the time taken(s) to solve a Sudoku puzzle. It is clear in the graph that as the number of empty cells increases , the time taken to solve the puzzle increases exponentially. This concludes that the theoretical analysis corresponds with the empirical analysis.

III

Part Three

3. Conclusion

From our analysis it is clear that our Hypotheses are correct. However, there were many isolated incidences that did not produce expected results. Upon further research it is concluded that the complexity of a Sudoku puzzle is not always determined strictly by the number of clues/empty cells provided.



Index