1. **Aims**

The main aim of this lab report is to review the performance of our group’s implementation of the Backtracking Algorithm on solving sudoku puzzles with varying amounts of zero spaces in the initial puzzle. Conversely, one could also interpret it as reviewing the algorithm’s performance on solving sudoku puzzles with varying amounts of filled spaces or clues in the initial puzzle. In this report we hypothesize that the more open spaces (or the fewer number of clues) there are in the initial sudoku puzzle, then the more computation will be required to solve such a puzzle.

It is thus our goal to verify our theoretical assumptions (which will be expanded upon in subsequent sections) of the algorithm’s performance with the empirical evidence we obtain from running our program on many different sudoku puzzles. Simply put, we want to check that the algorithm has the best, worst and average case complexities that we believe it to have theoretically. If this is not the case, then it is also our goal to determine why it is not.

1. **Summary of Theory**
2. **Time complexity:** O(9^(n\*n)).

For every unassigned index, there are 9 possible options so the time complexity is O(9^(n\*n)). The time complexity remains the same but there will be some early pruning so the time taken will be much less than the naive algorithm but the upper bound time complexity remains the same (Das, 2020).

1. **Space Complexity:** O(n\*n).

To store the output array a matrix is needed (Das, 2020).

1. **Experimental Methodology**

As stated in our aims, our initial assumption is that more open spaces (or fewer clues) in the initial sudoku puzzle will result in more computation needed in order to find the unique solution of a given puzzle. We have utilized 3 metrics to measure the performance of the algorithm, namely *number of comparisons*, *number of changes*, and the amount of *time* taken *to* return a unique solution to a given puzzle. Please see below discussions on our chosen metrics:

1. **Number of comparisons**

This metric measures the number of times the program compares the current value of the puzzle it is on with the rest of the incomplete puzzle. At each iteration, the program checks that the current value does not break the puzzle (ie: it checks to see that the current value of the puzzle is not repeated within that value’s row, column or sub-grid. It is our assumption that puzzles with more empty spaces will need more comparisons than puzzles with less empty spaces.

1. **Number of changes**

This metric measures the number of times the program changes a value it has just compared to a value that will not break the board (since it found that the value at the position is currently on has broken the board). It also measures the number of times the program changes an empty space to a filled space (a space with any positive integer less than 10).

Again, our group assumes that the program will make more changes to puzzles with more empty spaces than it will to puzzles with less empty spaces.

1. **Time**

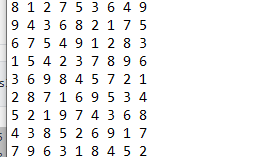
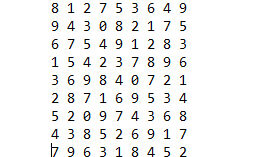
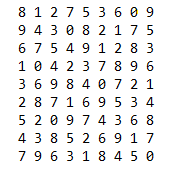
This metric measures the length of time (in milliseconds) it takes for the program to output a unique solution to a given puzzle. It is our assumption that puzzles with more empty spaces will take much longer to complete than puzzles with less empty spaces.

We had initially decided to use a puzzle’s difficulty, instead of the number of open spaces it has, as a metric to measure the performance of our program. We graphed the difficulty of the graph against the number of comparisons, number of changes, and time taken to solve. However, this metric was abandoned for a number of reasons. Firstly, the graphs that were generated using this metric gave very little insight to the performance of or program (if any). Secondly, we found that in some cases, puzzles of differing difficulties gave the same results. This is obviously incorrect, because an easy puzzle should not be getting the same results as a medium puzzle for example. We discovered that a ‘hard’ easy puzzle would yield the same results as an ‘easy’ medium puzzle. Thirdly, defining a puzzle’s difficulty is not east to do. One must not only look at the number of clues to determine that puzzle’s difficulty, but one must also take into account the positions of those clues as well which is not a simple task.

In addition, our group implemented the Backtracking Algorithm on 2 different sets of the same sudoku puzzles. This was to generate as much useful information as possible in order to gain more insights to the empirical performance of our program. In the first implementation our group randomly removed clues in denominations of 3 at each iteration. In the second implementation our group linearly removed clues in denominations of 3 at each iteration. Please see below the discussions on the 2 implementations of the algorithm:

1. **Randomly Remove Clues in Denominations of 3**

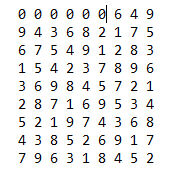
In this implementation we started off with the initial solved puzzle, ran our program on it and recorded and plotted the results. Then we randomly removed 3 clues from that puzzle and repeated that same process. This process was repeated until there were no more clues left in that puzzle (ie: the puzzle was left only with white spaces). Here are some illustrations of the first three iterations of the algorithm of a puzzle taken from sudoku.com (2020).

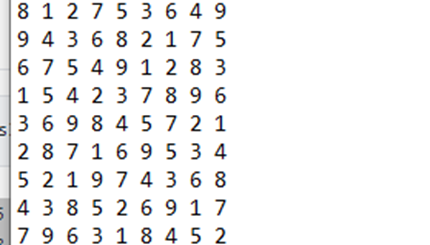
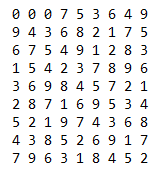


As you can see, the puzzle in the first iteration has no white spaces as that is the solved puzzle. The puzzle in the second iteration has white spaces in positions (1,3), (4, 5) and (6,3). The puzzle in the third iteration has white spaces in positions (1,3), (4, 5), (6,3), (0,7), (3,1) and (8,8). Results of the performance of this implementation of the algorithm will be presented, interpreted and reviewed in subsequent sections.

1. **Linearly Remove Clues in Denominations of 3**

Similarly as the implementation in (a), we started off with the initial solved puzzle, ran our program on it and recorded and plotted the results. Then we linearly removed 3 clues from that puzzle and repeated that same process. That is, we started in position (0,0) and removed the clue in that position, then we went on to remove the next 2 clues on that same row, clues (0,1) and (0,2). This process was repeated until there were no more clues left in that puzzle (ie: the puzzle was left only with white spaces). Here are some illustrations of the first three iterations of the algorithm of a puzzle taken from sudoku.com (2020).





As was the case in implementation (a), the puzzle in the first iteration has no white spaces as that is the solved puzzle. The puzzle in the second iteration has white spaces in positions (0,0), (0, 1) and (0,2). The puzzle in the third iteration has white spaces in positions (0,0), (0, 1), (0,2) (0,3), (0, 4) and (0,5). Results of the performance of this implementation of the algorithm will be presented, interpreted and reviewed in subsequent sections.

**8. References**

* Sudoku.com. 2020. *Play Free Sudoku Online - Solve Daily Web Sudoku Puzzles*. [online] Available at: [https://sudoku.com](https://sudoku.com/) [Accessed 19 October 2020].
* Das, M., 2020. *Sudoku | Backtracking-7 - Geeksforgeeks*. [online] GeeksforGeeks. Available at: <https://www.geeksforgeeks.org/sudoku-backtracking-7> [Accessed 21 October 2020].