Sudoku

Michael Beukman

October 24, 2020

Contents

1 Aims

The goal of this study is to investigate the Sudoku problem, using theoretical and experimental techniques.
We will implement an algorithm to solve the Sudoku puzzle by using the backtracking technique.
We will discuss the algorithm, the theory behind it and attempt to confirm the theory using empirical analysis.

2 Summary of Theory

2.1 Sudoku

Sudoku is a puzzle that is played on a $n^2 \times n^2$ grid, where the objective is to fill in each cell, while keeping to the rules that all rows, columns and highlighted $n \times n$ grids contain each digit 1 to n^2 exactly once. The most standard Sudoku puzzle has a size of 9×9. [1]

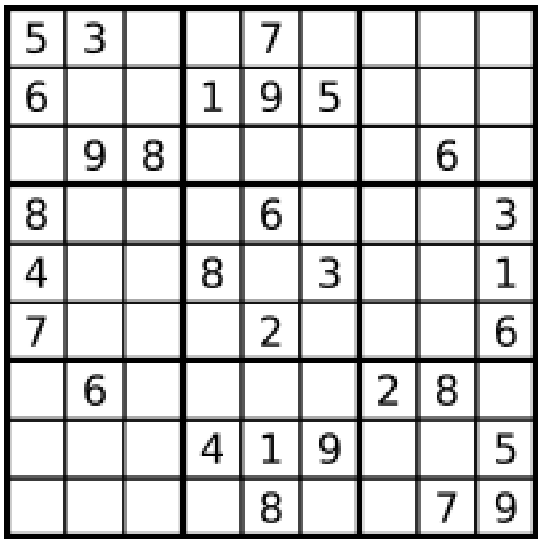


Figure 1: A suduko board [2]

2.2 Backtracking

Backtracking is a programming technique that is used to find solutions to problems, by incrementally building up potential solutions, but stopping the execution of one path as soon as the potential solution becomes invalid. It is a modification of exhaustive search, where we attempt to disregard invalid states and their children to drastically reduce the size of the search space.

We can think of backtracking as starting at the top of a tree, and traversing down a path. If a leaf is encountered where the solution is invalid, the algorithm goes back up the path it took, until it reaches a node which has a child that hasn't been visited yet. It then repeats this procedure until it finds a solution. [3] To make this procedure more efficient, we add checks that prune the tree, leaving us with drastically less states to explore.
Some of these pruning steps may be (among others) [3]:

1. Stop searching, and immediately backtrack if it is determined that the current state can never lead to a valid solution (in sudoku for example, this is when one of the constraints are not satisfied. Since we only add numbers, and never remove them, once there are two of the same number in the same area (row, column, grid) we know it will never reach a valid solution.)

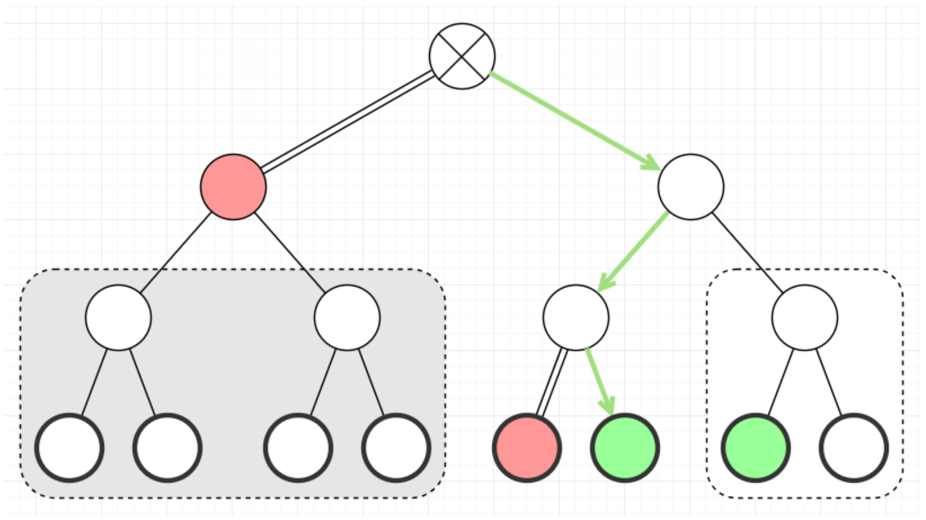


Figure 2: An example of a search tree [4]

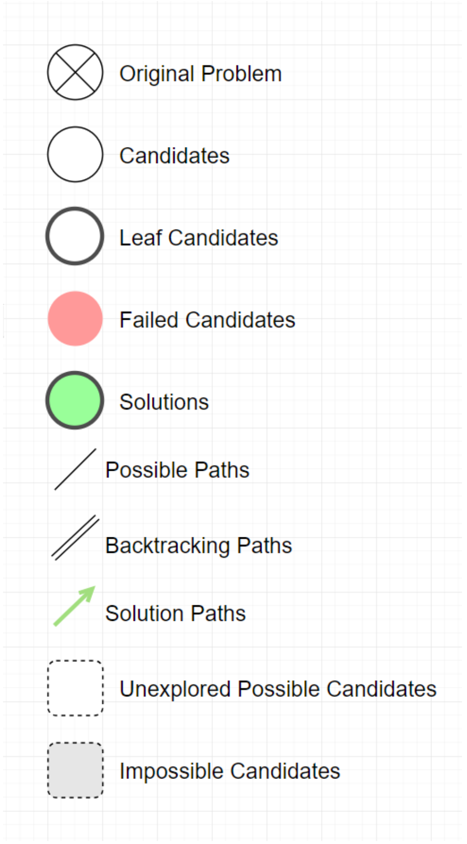


Figure 3: The legend for the above figure [4]

In the above figure, we can see clearly what the backtracking algorithm does and why it is effective. From the original position, the algorithm chooses to go down the left path. It determines that the red node is invalid for some reason. It then backtracks to the root node again and chooses the right path this time. It continues this until it reaches a valid solution, again backtracking on reaching the read leaf as that was invalid.
Now, we can see that the algorithm completely ignored the subtree rooted at the red node on the left, as all of those nodes will be invalid because their parent is invalid. If we consider a tree that has more levels than this one, it is clear that there is the potential for large swaths of the search space to be ignored, without even being examined directly, because they are definitely invalid.

A notable example of backtracking is a simple pathfinding technique. Consider a maze, in which an agent can move in the 4 cardinal directions, except if there is a wall blocking the way. A simple solution would be to always take the top, left, right and then bottom ways. As soon as one reaches a square where the agent cannot continue forward(and only can go back), the agent backtracks along it's path, until it reaches a position where a move is possible.
This is done until the final solution is found.

In the figure below, we can see an example. The goal is to reach the yellow circle from the green one. The numbers indicate the order in which the moves were done. The black lines are the wrong paths, red lines are the backtracking steps and the green lines indicate the correct path.

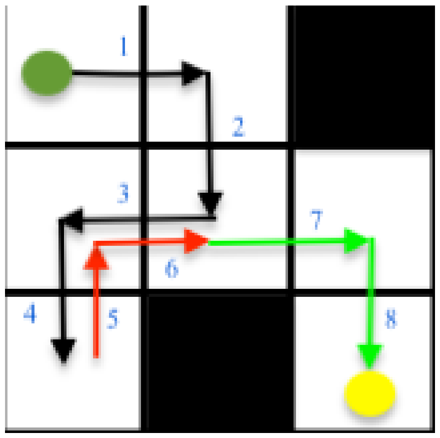


Figure 4: A simple maze

2.3 Our Algorithm

Here we propose an algorithm, based on backtracking, to solve the sudoku puzzle. The value n below is the square root of the length of the board (for example in a 9×9 board, $n = 3$)

The intuition behind our algorithm is this: For each empty cell, we try to fill in the numbers 1 to n^2 in order. For each number, we put it in the empty position, and continue with the process, assuming that the cell we just filled in is correct. If we reach a state where the board is invalid, we simply backtrack until we reach a previous valid state, and try the next number.
This is done until a valid solution is found.

Our algorithm was adapted from [5] to accommodate Sudoku.

Algorithm 1: Backtracking Algorithm for Solving Sudoku

```
1  function Solve ($index, board$);
   Input   : An integer $0 \leq index \leq n^4$ and $board$, a Sudoku board
   Output: A solved Sudoku Board or an InvalidBoard if no solution
           exists
2  if $index = n^4$ then
3      return $board$;
4  $column \leftarrow index \mod n^2$
5  $row \leftarrow \lfloor \frac{index}{n^2} \rfloor$
6  if $board[row][column]$ is filled then
7      return Solve($index + 1$, board) // already filled, so move on.
8  for $i \leftarrow 1$ to $n^2$ do
9      if $isValidMove(board, row, column, i)$ then
10         $board.makeMove(row, column, i)$
11         $answer \leftarrow Solve(index, board)$
12         if $answer.isValid()$ then
13             return $answer$;
14         $board.clearCell(row, column)$ // move is invalid so clear the cell.
15 end
16 return $InvalidBoard$
```

2.3.0.1 Proof of correctness

Proof. Assume the input is a valid sudoku board, with an unique solution.
Suppose further that the board.isValid() routine is correct, i.e. it returns true if the board is a filled sudoku board, satisfying all the constraints, and false otherwise.
Also suppose the isValidMove function is correct, i.e. it returns true if the specified move does not invalidate any constraints and false otherwise.

Suppose, for the sake of contradiction that we return an invalid board.
Now, there are only four places we can return: line 3, line 7, line 13, or line 16.

Line 3 only gets executed when the index is the same as the number of squares on the board. Since we call this algorithm with solve(0, board), it will not call this immediately.

Line 7 simply recurses to the next iteration. It only does this when the current cell is filled. See the next lines for what happens in the following calls.

If we are on line 13, then the answer.isValid() returned true. Thus the board is valid, and we return a valid solution. Our assumption states that we return an invalid board, so this cannot be the case.

If we are on line 16, consider how we got there: For every number 1 to n^2, we try to make a move corresponding to this number, and solve the puzzle again. If we reach line 15, it must mean that in every recursive step, we tried all of the moves, but didn't find a valid solution. (Some constraint was not satisfied. This was either on line 9, if the move was invalid, or line 12, where the returned board was invalid.)

Now, the algorithm attempts to visit all possible states of the board. It only backtracks when the current state is invalid. Since the algorithm always increments the index, just fills empty cells, and never deletes filled cells (except in line 14, where that move was invalid), as soon as we are in a invalid state, there is no way we can continue to get a valid board. It only stops when the index is larger than the number of squares on the board.

This means, out of all the possible boards, none of them were valid (We visited all potential boards, by the above).
This means that the initial board did not have a solution. This is a contradiction to our initial assumption. Thus, our algorithm returns the correct value for all possible valid inputs. □

2.4 Complexity Analysis

We attempt to detail the best case, worst case and average case for our algorithm here. In the following analysis, let:
n^2 be the width of the board
m be the number of empty cells in the board (so $0 \leq m \leq n^4$)

Best Case

In the above algorithm, we see that the best case would be if either the whole board is filled already, or if we only recurse once per function call. The former is not true in general, since when solving a sudoku puzzle, usually one doesn't have the final result.
So, the best real case is if the first answer we obtain is valid. This implies that we do not to have to solve the whole puzzle before we see that the move we made was invalid. The complexity in this case is:

$O(m)$ for the number of calls to the isValid() function.
$O(n^4)$ for the number of recursive calls.

Proof. Suppose that when the algorithm considers a move, only one of two things happen:

1. isValidMove returns false
2. isValidMove returns true and the subsequent answer.isValid returns true.

Under these circumstances, we show that the complexity is as above.
Now, for each value of index, if the square is not filled, we loop over the numbers 1 to n^2. If isValidMove returns false, we continue to the next iteration, and our basic operation is not performed.
If isValidMove does return true, we know that the subsequent answer is valid. We call isValid() once.
Now, the above happens for every recursive call where the square is not filled, so in total, we perform $O(m)$ calls of isValid().
For the number of recursions, we recurse exactly once per recursive call, until index $= n^4$. Thus we recurse $O(n^4)$ times. □

We can also see this by considering the recurrence relation for the number of calls to isValid() (the added one comes from testing once per function call):
$T(m) = T(m-1) + 1 \implies T(m) \in O(m)$

Worst Case

In the worst case, the algorithm will have to traverse the whole solution space before finding a correct solution. Exactly defining a board where this happens is quite difficult, but to simulate it, we can simply turn off the early stopping. This in effect reduces to exhaustive search. It is definitely possible for a normal run to result in the worst case, so we are simply making the experiment easier by turning off early stopping.
If we consider what happens if there is one empty cell, the algorithm will perform n operations (one for each number) When there are two empty cells, for each of the n options for the first cell, the algorithm will have to perform another n operations for the second cell.
This results in $O(n^{2m})$ time complexity (as well as the number of calls to isValid and the number of recursive calls)

Proof. Suppose, in the above algorithm, we do not include the if statement on line 9, and instead always perform the statements in lines 10-14. We do count that as still one basic operation (call to isValid()) though.
Suppose further that we do not return the answer inside the loop if it is valid, but rather after the whole execution of the loop.

In each recursive call that the square is not filled, we perform a loop n^2 times. This is true for every cell. If there were two cells, we would perform $n^2 \times n^2$

operations.
This continues on for each empty cell, and by induction we can see this results in $O(n^{2m})$ $\square$

We can also see this by considering the recurrence relation
$T(m) = n^2 \times T(m-1) \implies T(m) \in O((n^2)^m) = O(n^{2m})$

Average Case

The average case is bit more nuanced. To get an accurate estimate of the complexity, we will consider some simple assumptions: half of the time, the algorithm will solve a puzzle in $O(n^4)$ time. If the other half, it will solve it in $O(n^{2m})$ time. Clearly, then the final complexity is still $O(n^4 + n^{2m}) = O(n^{2m})$ Even if this was more skewed, and 90% (even 99.99%) of the time the algorithm solves it in linear time, the overall complexity will still be $O(n^{2m})$. We hypothesize that the constant is much less than the worst case.

The general complexity of the average case is very difficult to compute accurately, but with these assumptions, we can make a reasonable guess. We will attempt to confirm our results using empirical analysis.

3 Experimental Methodology

3.1 Implementation

Here we discuss some concrete details regarding our implementation. We used the Java programming Language to implement the algorithms. To run the experiments, we used a Dell Vostro 5490 laptop, with 4 i5 Intel cores. When running the experiments, the machine was not being used for any other purpose. We hope that this gives us more accurate data as it is not majorly influenced by the other processes running on the machine.

For data structures, we opted to go for a 2D array to store the current state of the board. We chose this because filling or clearing cells can be done in constant time.

3.2 Measurement Details

We will perform two sets of experiments to help determine if our analysis was correct.

1. Generate many random sudoku boards of various sizes. Using these boards, we will run our solver, and determine how long it takes to solve these boards.

2. Use a small set of boards, where we start off with a fully completed board, and gradually clear cells. On each of these new boards, we evaluate our

solver again. Thus we can judge the performance based on m, the number of empty cells.

In each of the above, we will measure a few things:

1. The time it takes to solve the board.
2. The number of basic operations it requires. For this purpose, we will use the board.isValid() call as our basic operation. The reason behind this is that this check usually takes $O(n^4)$ time to complete, but it can stop early. So instead of measuring the sub operations, which will result in results that aren't very stable, we used this as the basic operation. Also, when measuring the performance relative to the number of empty cells m, the size of the board is in fact constant.
3. The number of recursive calls of the solve() function

For the first metric, we simply use Java's built in $System.nanoTime()$ function. We call this before the algorithm runs, and just after. We then take the difference in times.
For the other two metrics, we simply keep a few counters that we increment when the operation in question is performed. This should not affect our complexity analysis at all, since incrementing a counter is done in $O(1)$ time.

Now, the reasons we chose these metrics are as follows:

1. Measuring the time directly provides a good indication of how the algorithm does as a whole. It is dependant on the specific machine it was tested on, but we mostly care about the growth rates instead of the actual values.
2. Using the basic operations give us an indication of how much work is being done by the algorithm.
3. The number of recursion calls inform us about the number of times the algorithm actually backtracks, which we will see very clearly in the best case and worst case graphs below.

We believe that using these 3 metrics together provide us a good indication of the algorithm's performance.
With three metrics, the reader can focus on the one that is most important to them.

To ensure reliable results, we run each of our experiments 1000 times and report the average of those results. This is done to mitigate the effect of outliers in any one run.

3.3 List of Experiments

In the first 3 experiments below, we measure the complexity as a function of the number of empty cells m. We do this for 4×4, 9×9 and 16×16 boards. In these graphs, we plot number of empty cells (x-axis) vs the three metrics we discussed earlier.
For the fourth experiment we measure the complexity as a function of the board size, and merely note the number of empty cells. We mainly plot the distribution of our metrics.
Finally, using the data obtained above, we compile some graphs that show data from different board sizes and different cases on one axis. This is done to directly compare the different board sizes and cases.

3.3.1 Best Case

In this experiment, we attempt to solve a sudoku puzzle where the algorithm only recurses the minimal amount of times, and the first 'choice' it makes is correct. This means that if it enters into a recursive function, it never returns an invalid board.
For the 4×4 case and the 16×16 case, we just used boards that were filled by using the backtracking algorithm above, and gradually cleared cells.
For the 9×9 case, the above strategy caused us to recurse unnecessarily, so we had to add a few more restrictions. This was still not perfect, but good enough for our purposes.

3.3.2 Worst Case

In this experiment, we simply turn off early stopping, simulating the case where the algorithm encounters a board that will force it to recurse up to the end of the tree before determining that the path is incorrect. We did this because it is difficult to construct boards that force the above condition. It is a possible scenario, and is made easier by turning off early stopping.
Here we only ran across a small number of empty cells, as any more took prohibitively long.

3.3.3 Average Case

In this experiment, we generate a few random boards, and measure the performance of the algorithm as a function of the number of empty cells. There was no specific restrictions on the boards.

3.3.4 Boards

In this experiment, we pre-generate some number of boards that have a unique solution, and test the algorithm on these boards. This leads to slightly more realistic measurements, as previously the boards might have had more than one solution (when the number of empty squares was increased).

We generated 100 different boards for the 4×4 and 9×9 cases and only 10 boards for the 16×16 case. This was done in the interest of time.

3.3.5 Board Generation

All of the experiments above relied on some sort of board generation. To do this, we used the above algorithm, and added some slight modifications.

For the best case, we just ran the algorithm on an empty board, and removed cells from the result. We removed cells in order, starting with the last one, and moving towards the first one.

For the worst case generated the boards in the same way as above.

For the average case, we generated a board similarly to the above, but instead of trying moves in order (e.g. $1, 2, \ldots, n^2$), we randomly shuffled this range. We also removed cells in a random order.

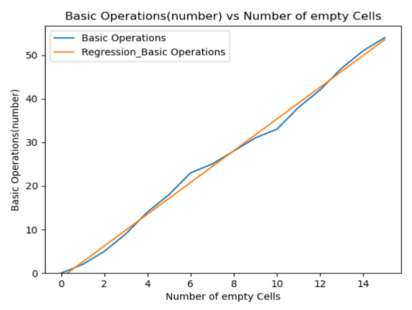
When generating realistic sudoku boards, we used the same method as the average case, and removed cells until there was more than one solution. Then we stopped and added the last cell back in, ensuring unique solutions for our boards.

4 Presentation of results

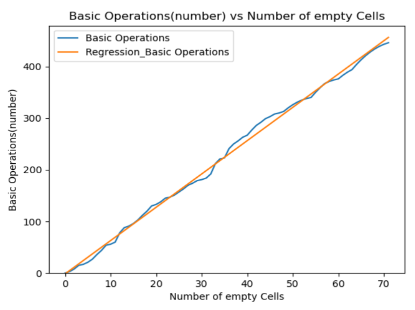
In the following graphs, we plot number of empty cells (x-axis) vs the three metrics we discussed earlier. We also show the line of best fit that was found.
If not stated otherwise, the x-axis is the number of empty cells, where the y-axis is either Time (ms), Number of Basic operations or Recursion count. Where applicable, the blue line is the actual data and the orange line is the line of best fit.
In the captions, we also show the equation of the line of best fit. y is the metric under consideration in that graph and x is the x-axis of that graph (usually the number of empty cells).

For the Board graphs, we show the distribution of the different metrics in bar charts and box plots.
We also combine the board data to show graphs of our metrics (the average) against the number of cells in the board.

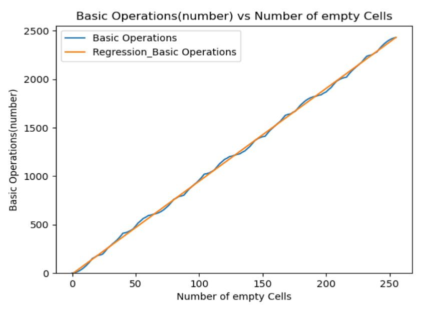
4.1 Best Case



(a) 4 x 4 board. $y = 3.6412 \times x - 1.0588$

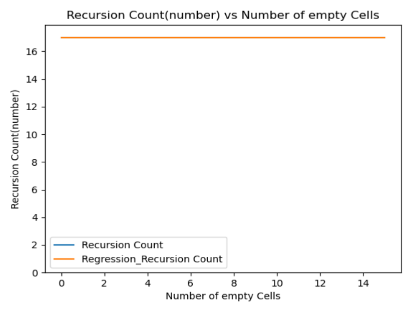


(b) 9 x 9 board. $y = 6.448 \times x - 1.4741$

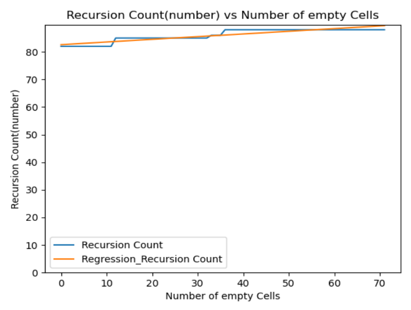


(c) 16 x 16 board. $y = 9.5659 \times x - 8.4047$

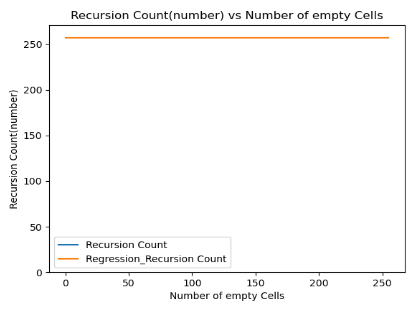
Figure 5: Basic Operations



(a) 4 x 4 board. $y = 0.0 \times x + 17.0$

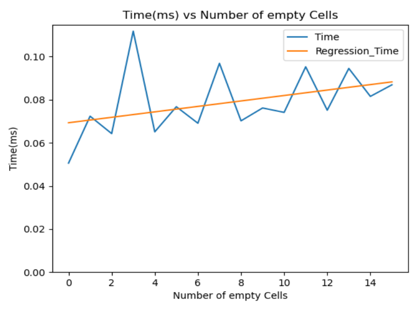


(b) 9 x 9 board. $y = 0.0971 \times x + 82.5947$

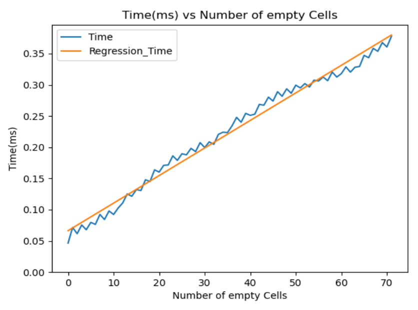


(c) 16 x 16 board. $y = 0.0 \times x + 257.0$

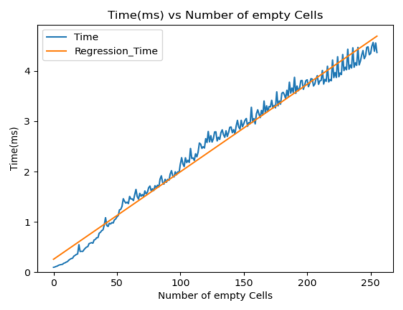
Figure 6: Recursion Count



(a) 4 x 4 board. $y = 0.0013 \times x + 0.0693$



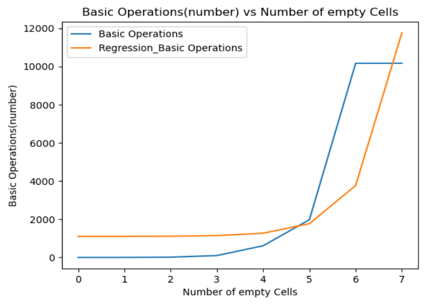
(b) 9 x 9 board. $y = 0.0044 \times x + 0.0664$

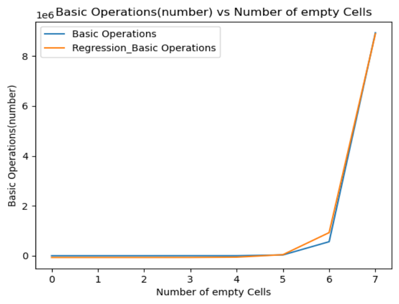


(c) 16 x 16 board. $y = 0.0174 \times x + 0.256$

Figure 7: Time

4.2 Worst Case


(a) 4 x 4 board. $y = 6.51\text{e-}01 \times 4^x + 1.10\text{e}03$


(b) 9 x 9 board. $y = 1.87\text{e}00 \times 9^x - 6.93\text{e}04$

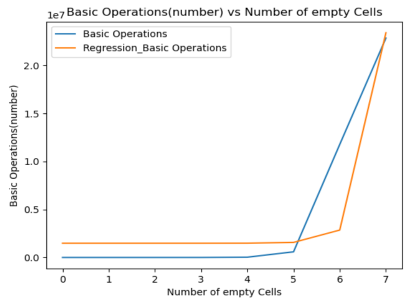
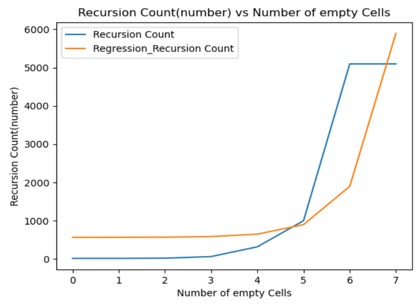
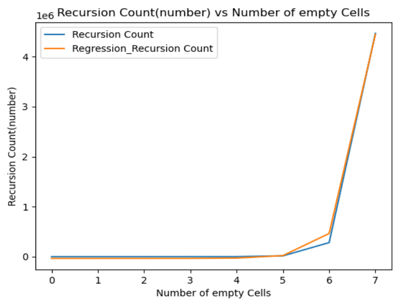

(c) 16 x 16 board. $y = 8.17\text{e-}02 \times 16^x + 1.48\text{e}06$

Figure 8: Basic Operations


(a) 4 x 4 board. $y = 3.25\text{e-}01 \times 4^x + 5.66\text{e}02$


(b) 9 x 9 board. $y = 9.37\text{e-}01 \times 9^x - 3.46\text{e}04$

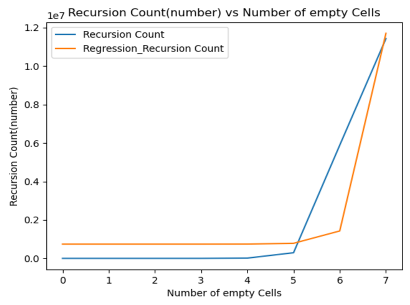
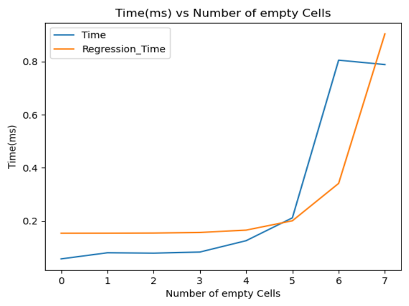
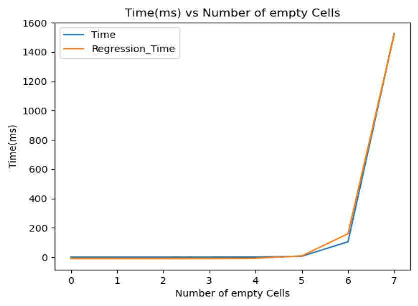

(c) 16 x 16 board. $y = 4.08\text{e-}02 \times 16^x + 7.41\text{e}05$

Figure 9: Recursion Count


(a) 4 x 4 board. $y = 4.59\text{e-}05 \times 4^x + 1.53\text{e-}01$


(b) 9 x 9 board. $y = 3.20\text{e-}04 \times 9^x - 1.05\text{e}01$

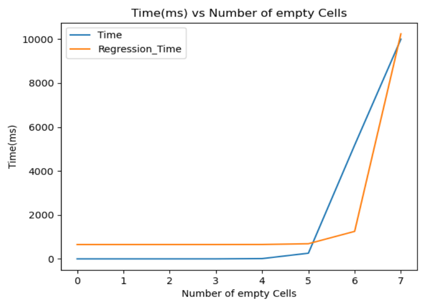
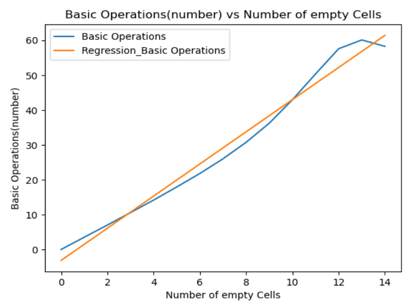
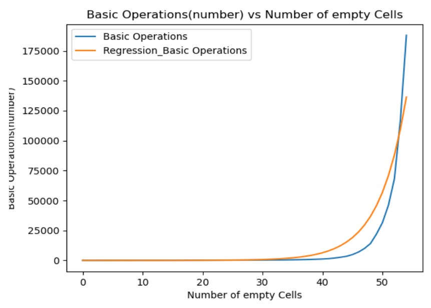

(c) 16 x 16 board. $y = 3.57\text{e-}05 \times 16^x + 6.53\text{e}02$

Figure 10: Time

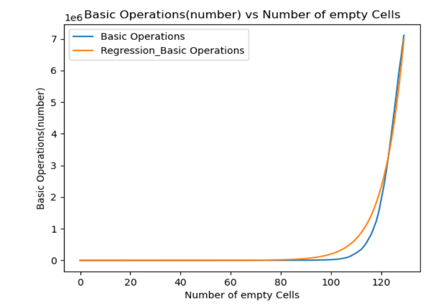
4.3 Average Case



(a) 4 x 4 board. $y = 4.6139 \times x - 3.135$

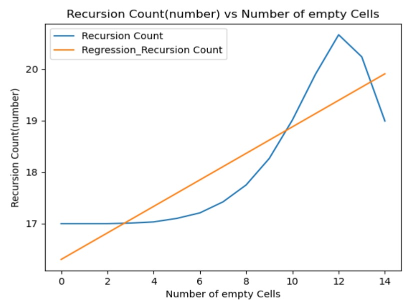


(b) 9 x 9 board. $y = exp(0.219 \times x)$

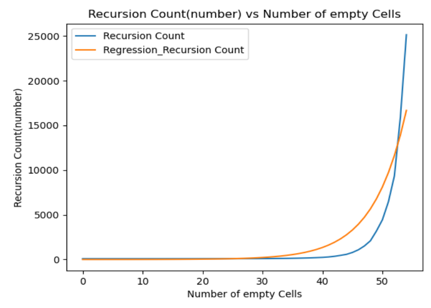


(c) 16 x 16 board. $y = exp(0.1222 \times x)$

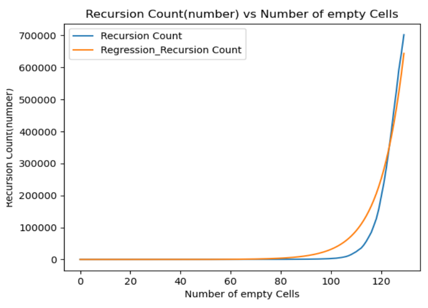
Figure 11: Basic Operations



(a) 4 x 4 board. $y = 0.2572 \times x + 16.3059$

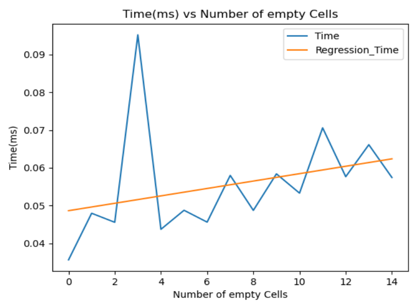


(b) 9 x 9 board. $y = exp(0.18 \times x)$

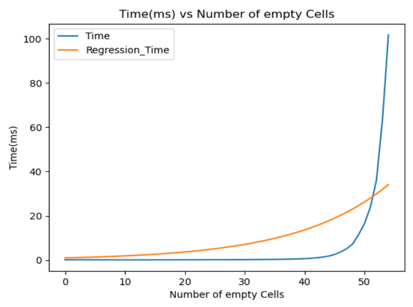


(c) 16 x 16 board. $y = exp(0.1037 \times x)$

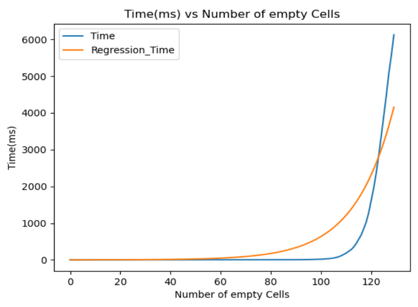
Figure 12: Recursion Count



(a) 4 x 4 board. $y = 0.001 \times x + 0.0486$



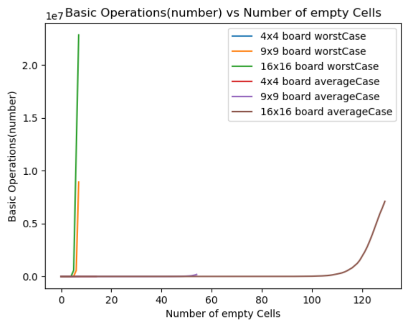
(b) 9 x 9 board. $y = exp(0.0654 \times x)$



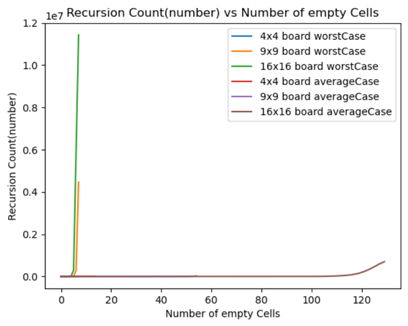
(c) 16 x 16 board. $y = exp(0.0646 \times x)$

Figure 13: Time

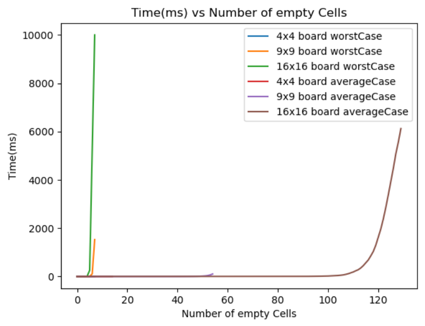
4.4 Combining the Graphs



(a) Basic Operations

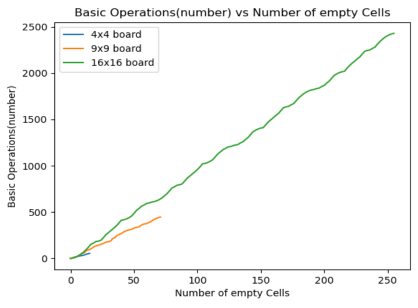


(b) Recursion Count

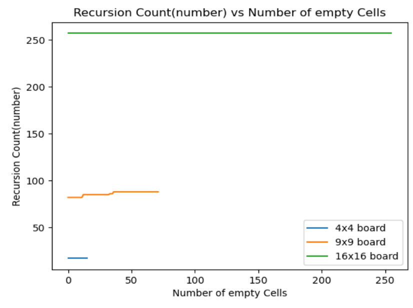


(c) Time

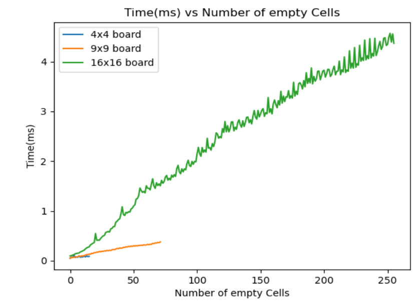
4.5 Split by case



(d) Best Case

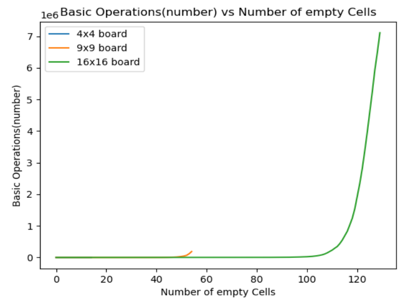


(e) Best Case

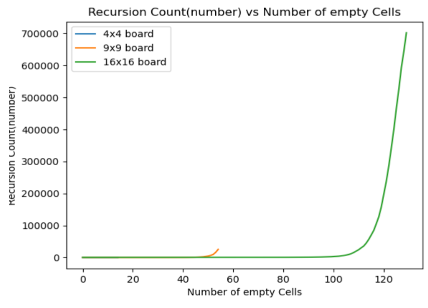


(f) Best Case

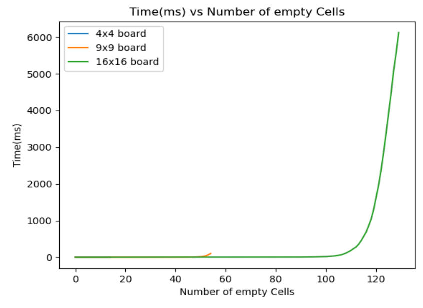
Figure 14: Best Case



(a) Average Case

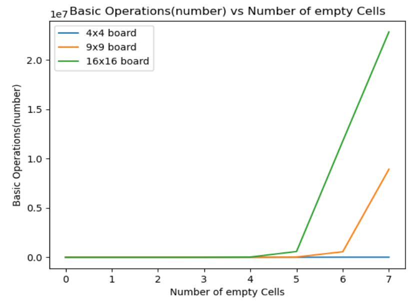


(b) Average Case

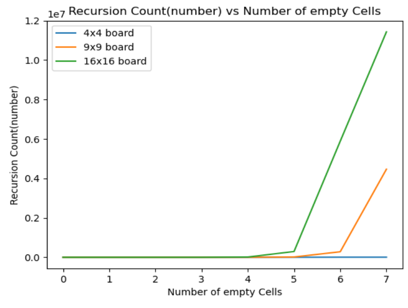


(c) Average Case

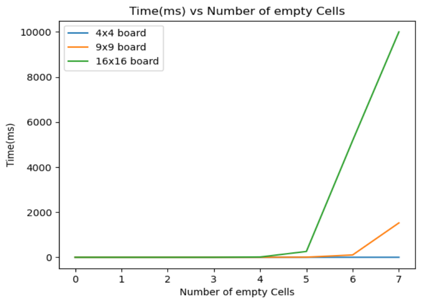
Figure 15: Average Case



(a) Worst Case



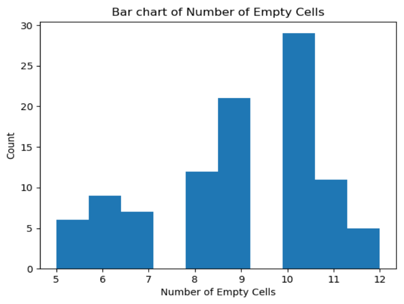
(b) Worst Case



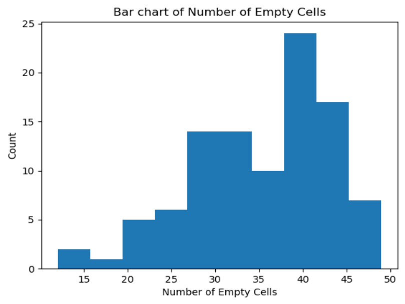
(c) Worst Case

Figure 16: Worst Case

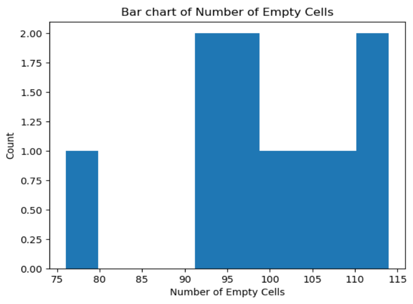
4.6 Boards



(a) 4 x 4 board.

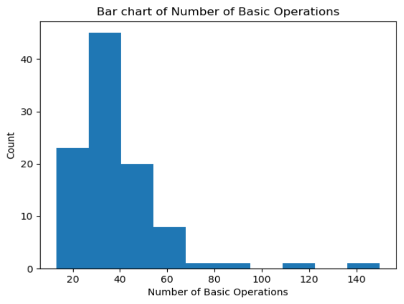


(b) 9 x 9 board.

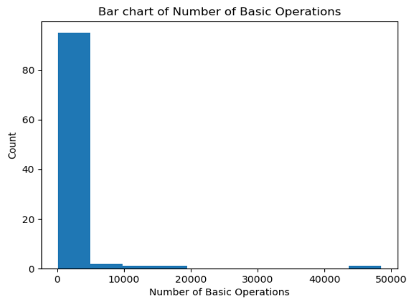


(c) 16 x 16 board.

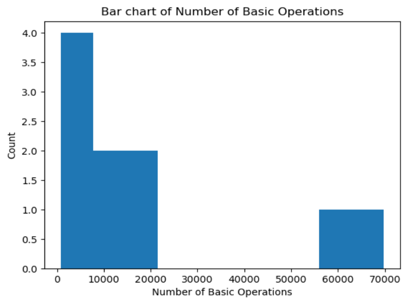
Figure 17: Number of Empty Cells



(a) 4 x 4 board.

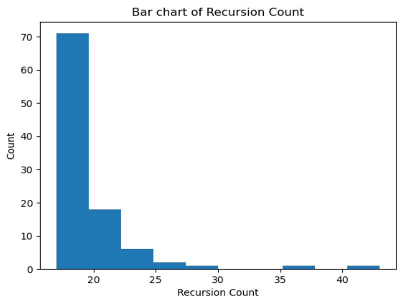


(b) 9 x 9 board.

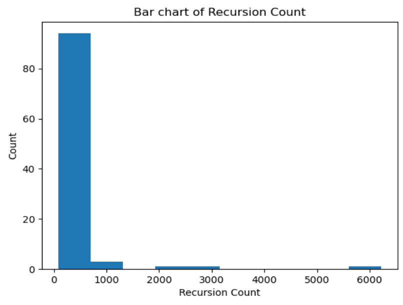


(c) 16 x 16 board.

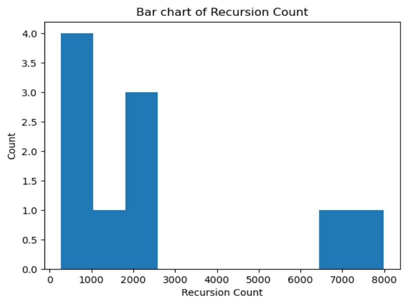
Figure 18: Number of Basic Operations



(a) 4 x 4 board.

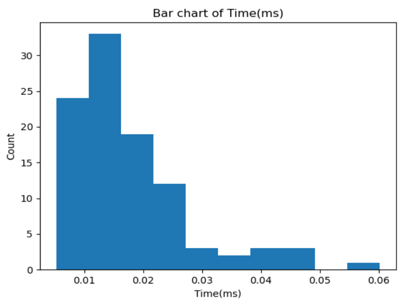


(b) 9 x 9 board.

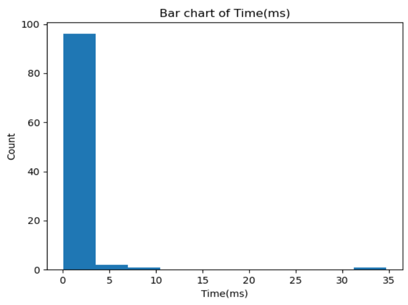


(c) 16 x 16 board.

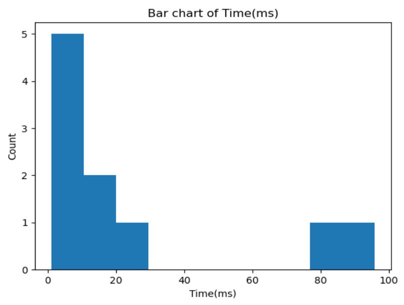
Figure 19: Recursion Count



(a) 4 x 4 board.

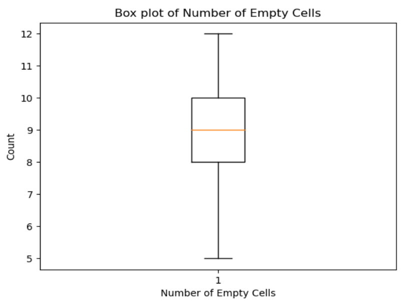


(b) 9 x 9 board.

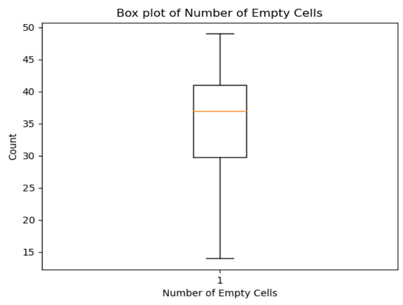


(c) 16 x 16 board.

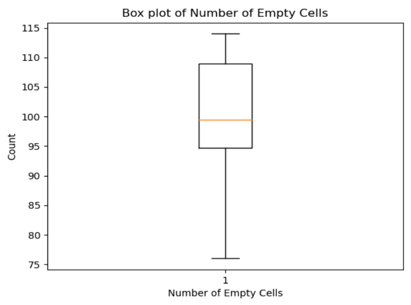
Figure 20: Time



(a) 4 x 4 board.

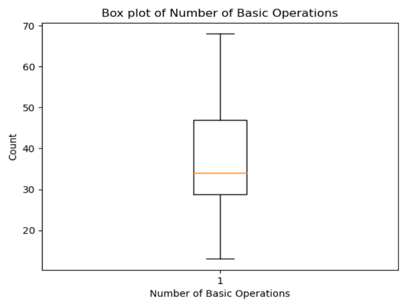


(b) 9 x 9 board.

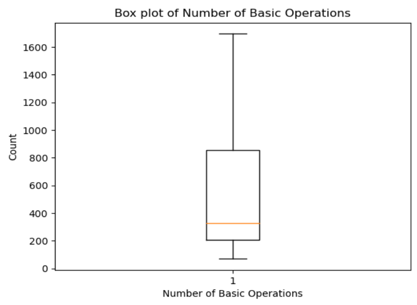


(c) 16 x 16 board.

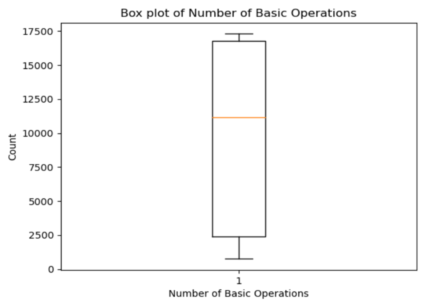
Figure 21: Number of Empty Cells



(a) 4 x 4 board.

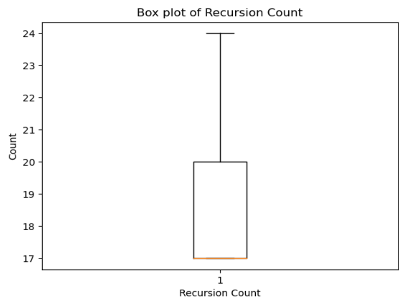


(b) 9 x 9 board.

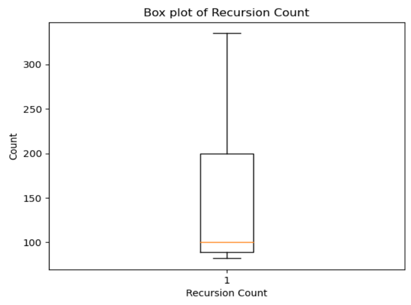


(c) 16 x 16 board.

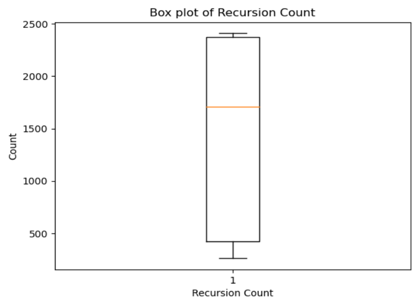
Figure 22: Number of Basic Operations



(a) 4 x 4 board.

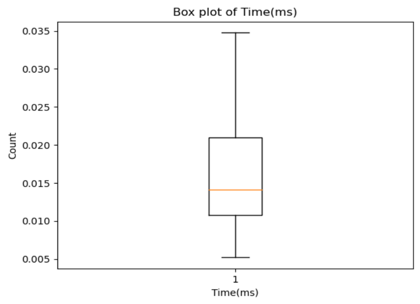


(b) 9 x 9 board.

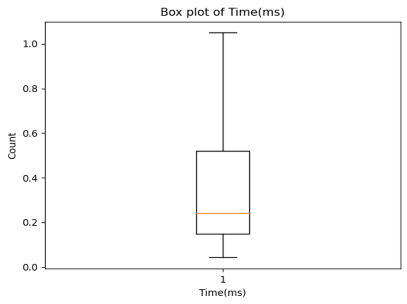


(c) 16 x 16 board.

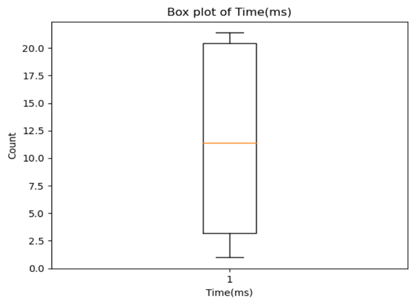
Figure 23: Recursion Count



(a) 4 x 4 board.



(b) 9 x 9 board.

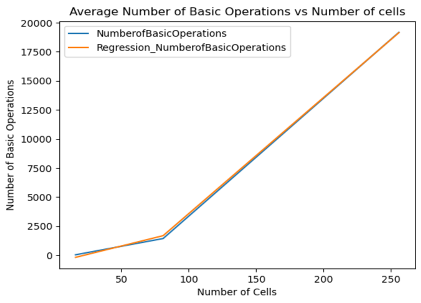


(c) 16 x 16 board.

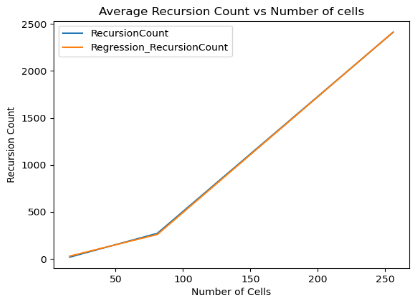
Figure 24: Time

4.6.0.1 Single graphs

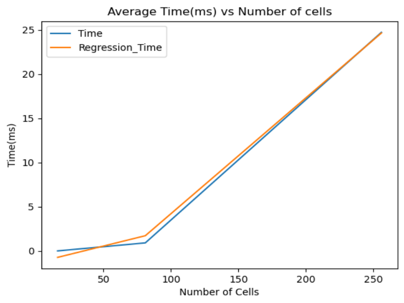
Here we combine the above data to show the relationship between the number of cells and the average of our metrics. This was obtained by averaging our metrics per board size and plotting that vs the number of cells (e.g. 9×9 has 81 cells)



(a) $y = 0.2961 * x^2 - 259.2126$



(b) $y = 0.0365 * x^2 + 20.8049$



(c) $y = 0.0004 * x^2 - 0.8143$

4.7 Statistical analysis

We mostly used linear regression, and fit the following curves:
$y = ax + b$ for the best case
$y = a * (n^2)^x$ for the worst case
$y = exp(ax)$ for the average case
$y = ax^2$ for the combined board graph, where $x = n^4$ is the number of cells in a board.
For example, in the worst case, our data consisted of y and n^{2x} and we fit that using linear regression.

The average case required us to fit a non linear curve, which we accomplished using the python package Scipy [6].
We could not fit a curve of the form n^{2x} using linear regression for the average case, as the numbers were too large and started to overflow.
Since we didn't use linear regression in the above step, we don't have values for the correlation coefficient. We have to rely on the standard deviation to judge our correctness.

Our curve fitting techniques and results were satisfactory, as we can see that the correlation coefficient (r) is always quite close to one. The standard deviation (σ) is also quite low in most of our experiments.
We have more detailed information about our curve fitting results in the Appendix.

5 Interpretation of results

Here we discuss our results, and what they can tell us about the algorithm.
We saw that the best case of the algorithm is when there is no unnecessary recursion, which resulted in a linear relationship between the number of empty cells, the time taken and the number of basic operations. In this case, we did see that the number of recursion steps were constant (per board size), which means that the algorithm recursed the same amount each time. This indicates that the algorithm at least checks each cell in the board once.

Visually, our fitted curves look very close to the actual data and our correlation coefficient was close to one. Thus the experimental analysis confirms our theory for the best case.

The worst case was a bit more severe though. In the worst case, the algorithm reduces to exhaustive search. Due to hardware limitations, we could only perform small experiments, as they took very long.
We saw exponential relationships between the number of empty cells and all three our metrics. The correlation coefficient is still quite close to one (except for the 4×4 case). Thus our theory is again confirmed by the experiments.

The average case, the relationships weren't as clear as the above. We see roughly exponential relationships between the number of empty cells and our three metrics. The growth rate of these was much smaller than that of the worst case. This indicates, that even though the algorithm runs in exponential time, it is possible to complete non trivial puzzles, because of the early pruning backtracking entails.
The line of best fit is not as perfect as in the worst case. This is potentially due to the fact that the average case has more variability. With more runs, this might be improved.

Notably, in the 4×4 board case, the graphs we fit were linear instead of exponential. This is potentially because the small board did not provide enough data to accurately fit a curve. Another problem could be that the 4×4 case is quite trivial, so the difference in times between 1 empty cell and 16 is minimal. The 9×9 and 16×16 boards did however exhibit exponential behaviour, so we can regard the 4×4 case as an outlier.

When evaluating the graphs where we combined our different results, we can see the following clearly:
$T(16 \times 16) \geq T(9 \times 9) \geq T(4 \times 4)$
$T(\text{worst case}) \geq T(\text{average case}) \geq T(\text{best case})$
where $T(z)$ represents our different metrics for the case z. We also saw that the worst case takes much longer than the average case (the 16×16 average case with 120 empty cells compares to the 9×9 worst case with only 6 empty cells in figure 4.4(a))

When running the algorithm on a random set of boards with unique solutions, we see that the performance was mostly clustered, which meant that the time to solve each puzzle was more or less the same each time, which is to be expected. There were some outliers that did not follow this pattern.
When fitting the curves to describe the boards, we did not have many points(only 4×4, 9×9 and 16×16 boards). This led to some potentially inaccurate fitting (as there are 3 points, and we fit a parabola), but we can see a general trend of a $O(x^2) = O(n^8)$ curve, where x is the number of cells in the board, which is the same as n^4.
It took too long to successfully generate 25×25 boards, and 1×1 boards are trivial, so getting more data is infeasible.

When evaluating the algorithm on realistic sudoku boards, we see that it performs relatively consistently Even solving 16×16 puzzles was doable in reasonable time.

6 Relate results to theory

From the above experiments we see that the theory was supported by our experiments. Specifically, we found that the best case was indeed $O(m)$, and $O(n^4)$ for time and for the number of basic operations, respectively.

The worst case was also confirmed to be $O(n^{2m})$

For the average case, we saw that the complexity was overall exponential, as predicted, but the fitting was not as perfect as with the above two cases. This possibly was due to not running enough experiments to confirm.
Regarding how certain we can be for the above, the errors obtained when doing regression on our data were mostly very small, making us confident that our results are correct. The average case errors were also small, so we are confident in our results.

7 Conclusion

In conclusion, we see that the theoretical analysis holds for the sudoku puzzle. Our experiments clearly showed that our theoretical analysis does indeed hold true.
Despite the exponential average case, the backtracking algorithm manages to solve most well set up puzzles in reasonable time. This is due to the pruning capabilities of this algorithm, which drastically reduce the search space, making the problem generally tractable.

We have shown that our hypotheses regarding the theoretical complexity were in fact confirmed by our experimental analysis. The study was therefore successful.

References

[1] Terry Stickels with Anthiny Immanuvel. *Sudoku.* Charlesbridge, WaterTown, Massachusetts, 2011.

[2] Tim Stellmach. Sudoku layout.

[3] Robert Sedgewick. *Algorithms.* Addison-Wesley, Reading, Massachusetts, 1983.

[4] Shirsh Zibbu. Sudoku and backtracking.

[5] H.A. Priestley and M.P. Ward. A multipurpose backtracking algorithm. *Journal of Symbolic Computation*, July 1994, 1994.

[6] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

8 Appendix A - Regression Details

Here we have some details regarding our curve fitting procedure, as well as some more detailed information about the errors, as well as the exact functions we tried to fit. In general, our correlation coefficient (when applicable) is close to one, and our σ is quite small.

Case	x	y	Size	a	b	r	σ	equation
best	# of Empty Cells	Basic Operations	4	3.641176	-1.058824	0.997875	0.06355	$y = a * x + b$
best			9	6.448019	-1.474125	0.99832	0.044731	$y = a * x + b$
best			16	9.565919	-8.404669	0.999656	0.015745	$y = a * x + b$
best		Recursion Count	4		17.0			$y = a * x + b$
best			9	0.097096	82.594749	0.906381	0.005409	$y = a * x + b$
best			16		257.0			$y = a * x + b$
best		Time	4	0.001264	0.069322	0.396405	0.000783	$y = a * x + b$
best			9	0.004414	0.066434	0.992914	6.3e-05	$y = a * x + b$
best			16	0.0174	0.256008	0.992008	0.000139	$y = a * x + b$
worst		Basic Operations	4	0.650994	1103.380039	0.814532	0.189282	$y = a * 4^x + b$
worst			9	1.873504	-69299.944998	0.998857	0.036598	$y = a * 9^x + b$
worst			16	0.081689	1481625.690051	0.904845	0.015691	$y = a * 16^x + b$
worst		Recursion Count	4	0.325192	566.023353	0.814491	0.094567	$y = a * 4^x + b$
worst			9	0.936751	-34570.847499	0.998857	0.018299	$y = a * 9^x + b$
worst			16	0.040845	741066.889433	0.904845	0.007846	$y = a * 16^x + b$
worst		Time	4	4.6e-05	0.153223	0.806263	1.4e-05	$y = a * 4^x + b$
worst			9	0.00032	-10.511169	0.999104	6e-06	$y = a * 9^x + b$
worst			16	3.6e-05	652.766879	0.903758	7e-06	$y = a * 16^x + b$
average		Basic Operations	4	4.613932	-3.134992	0.991024	0.172623	$y = a * x + b$
average			9		0.218957		0.000932	$y = exp(b * x)$
average			16		0.122179		0.000104	$y = exp(b * x)$
average		Recursion Count	4	0.257207	16.305883	0.87227	0.039994	$y = a * x + b$
average			9		0.180037		0.001136	$y = exp(b * x)$
average			16		0.103686		0.000154	$y = exp(b * x)$
average		Time	4	0.000982	0.048625	0.3088	0.000839	$y = a * x + b$
average			9		0.065355		0.002829	$y = exp(b * x)$
average			16		0.06459		0.000344	$y = exp(b * x)$
Boards	Number of Cells	Numberof Basic Operations		0.296131	-259.212587	0.999758	0.006522	$y = a * x^2 + b$
Boards		Recursion Count		0.036505	20.804864	0.99996	0.000326	$y = a * x^2 + b$
Boards		Time		0.000389	-0.81434	0.998473	2.2e-05	$y = a * x^2 + b$

Table 1: A table our our regression coefficients and errors