Sudoku

Michael Beukman

October 13, 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.
In addition to this, we will do theoretical analysis of the algorithm and attempt to confirm this using experimental 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 3x3 grids contain each digit 1 to n^2 only once. [1]

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

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 a possible solutions, but stopping the execution of one path as soon as the problem becomes invalid. It is a modification of exhaustive search, where we attempt to disregard invalid states and their

children.

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.)

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.

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, $n = 3$)

Algorithm 1: Backtracking Algorithm for Solving Sudoku

```
1  function Solve (index, board);
   Input  : An integer 0 ≤ index ≤ n^4 and board, a Sudoku board
   Output: A solved Sudoku Board
2  if index = n^4 then
3  |  return board;
4  column ← index  mod n^2
5  row ← ⌊index / n^2⌋
6  if board[row][column] is filled then
7  |  return Solve(index + 1, board) // already filled, so move on.
8  for i ← 1 to n^2 do
9  |  if isValidMove(board, row, column, i) then
10 |  |  board.makeMove(row, column, i)
11 |  |  answer ← 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 $0 \leq m \leq n^2$)

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 we the first answer we obtain is valid. This causes us not to have to solve the whole puzzle before we see that it is invalid. The complexity in this case is: $O(m)$ for the number of basic operations
$O(n^4)$ 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 $O(m)$ basic operations.
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 (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.

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 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 $O(n^{2m})$ □

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 $O(n^m)$ 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 $O(n^{2m})$, but we hypothesize that the constant is much less than the worst case.

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 an iMac, with 4 i5 Intel cores, as well as an ubuntu machine with 4 i7 cores. We did not compare results across machines.

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 a 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.

3.3 List of Experiments

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 4x4 case and the 16x16 case, we just used boards that were filled by using the backtracking algorithm above, and gradually cleared cells.
For the 9x9 case, the above strategy caused us to recurse unnecessarily, so we had to add a few more restrictions.
In the following graphs, we plot number of empty cells (x-axis) vs the three metrics we discussed earlier.

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 set 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).

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, but removed cells in a random order.

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.

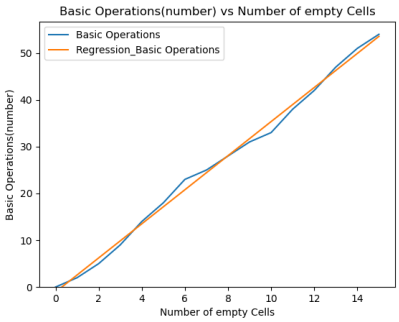
When generating realistic sudoku boards, we used the same method as the average case, and removed cells until there was more than one solution. Then we stopped and added the last cell back in, ensuring unique solutions for our boards.

4 Presentation of results

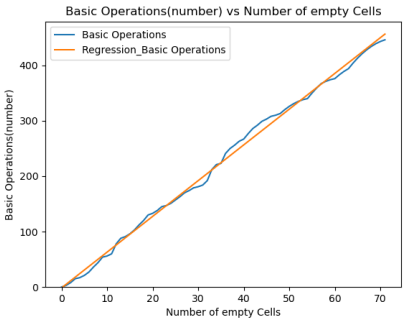
In the following graphs, we plot number of empty cells (x-axis) vs the three metrics we discussed earlier. We also show the line of best fit that was found. If not stated otherwise, the x-axis is the number of empty cells, where the y-axis is either Time (ms), Number of Basic operations or Recursion count.

4.0.1 Best Case

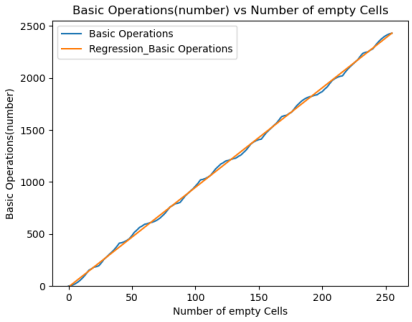
4.0.1.1 Basic Operations



(a) 4x4 board. $y = 3.64 * x + -1.06$

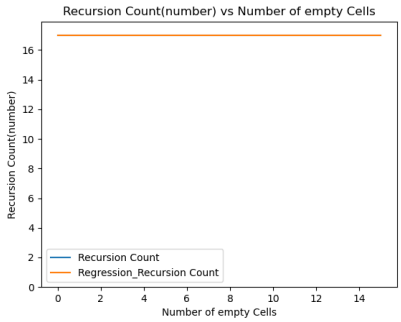


(b) 9x9 board. $y = 6.45 * x + -1.47$

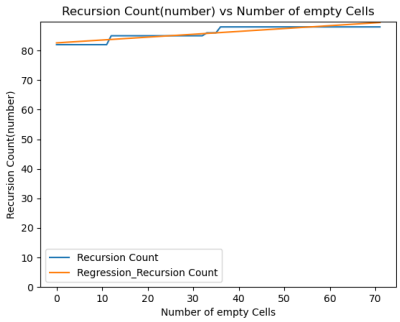


(c) 16x16 board. $y = 9.57 * x + -8.4$

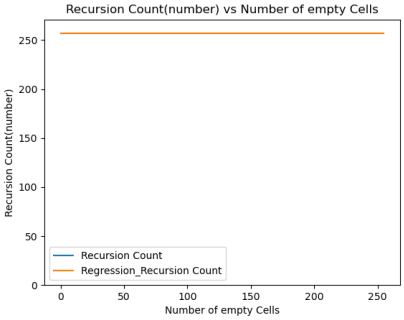
4.0.1.2 Recursion Count



(d) 4x4 board. $y = 0.0 * x + 17.0$

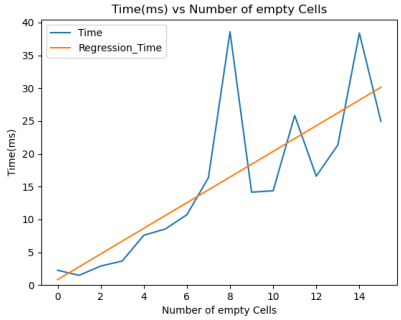


(e) 9x9 board. $y = 0.1 * x + 82.59$

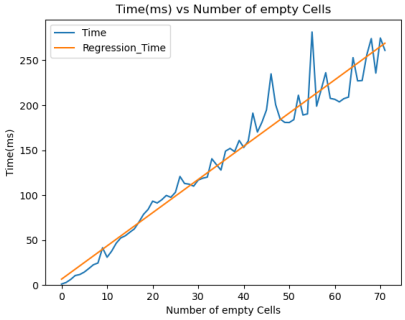


(f) 16x16 board. $y = 0.0 * x + 257.0$

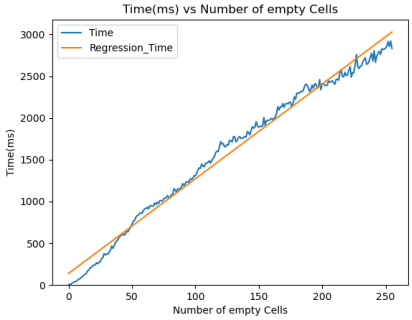
4.0.1.3 Time



(g) 4x4 board. $y = 1.95 * x + 0.83$



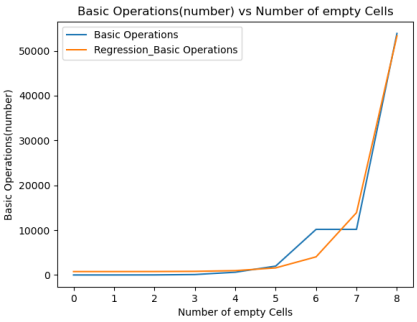
(h) 9x9 board. $y = 3.7 * x + 6.56$



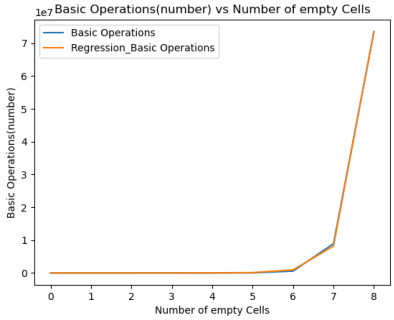
(i) 16x16 board. $y = 11.33 * x + 139.18$

4.0.2 Worst Case

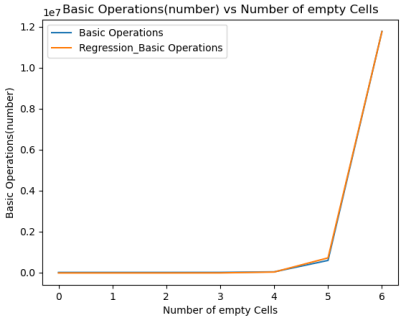
4.0.2.1 Basic Operations



(j) 4x4 board. $y = 0.8 * 4^x + 759.17$

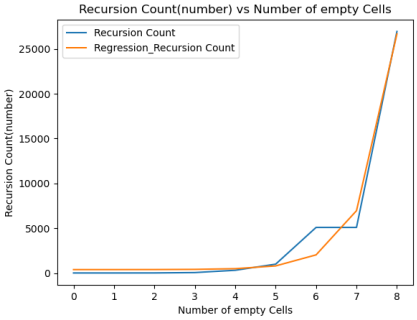


(k) 9x9 board. $y = 1.71 * 9^x + 32157.84$

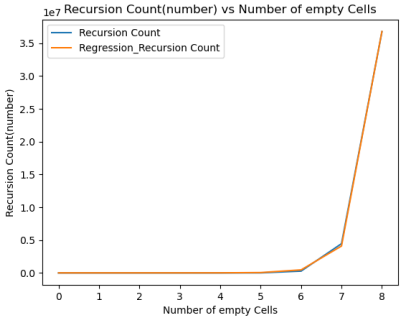


(l) 16x16 board. $y = 0.7 * 16^x + -27216.41$

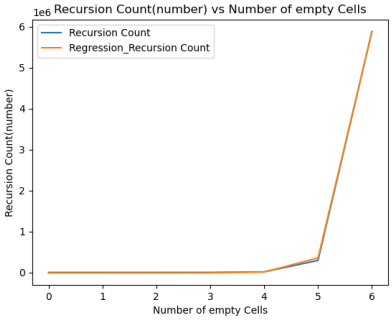
4.0.2.2 Recursion Count



(m) 4x4 board. $y = 0.4 * 4^x + 393.42$

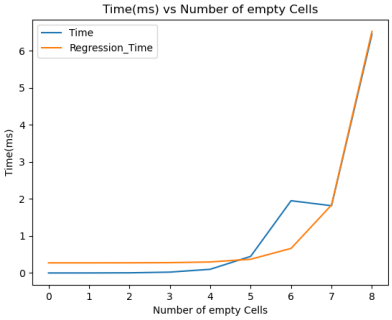


(n) 9x9 board. $y = 0.85 * 9^x + 16157.54$

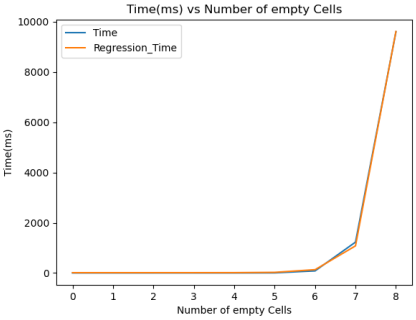


(o) 16x16 board. $y = 0.35 * 16^x + -13353.64$

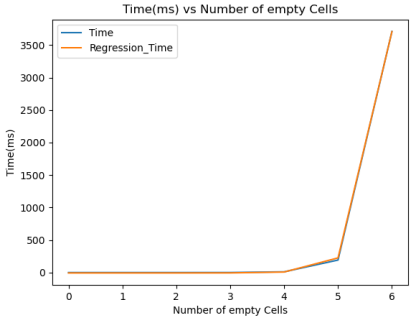
4.0.2.3 Time



(p) 4x4 board. $y = 0.0 * 4^x + 0.27$



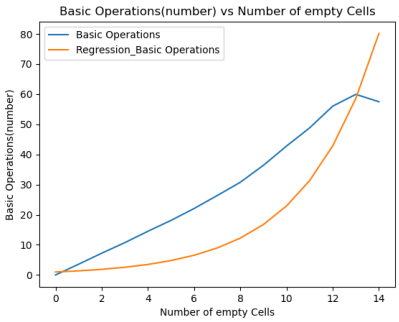
(q) 9x9 board. $y = 0.0 * 9^x + 12.54$



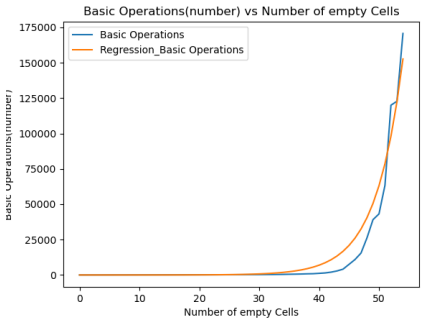
(r) 16x16 board. $y = 0.0 * 16^x + -7.1$

4.0.3 Average Case

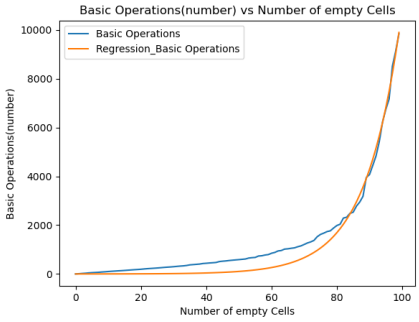
4.0.3.1 Basic Operations



(s) 4x4 board. $y = exp(0.31 * x)$

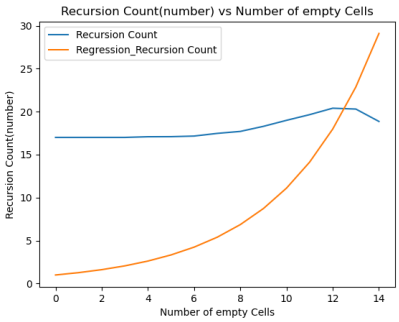


(t) 9x9 board. $y = exp(0.22 * x)$

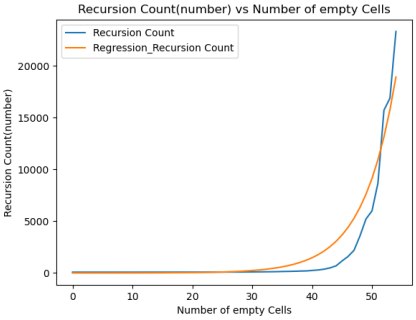


(u) 16x16 board. $y = exp(0.09 * x)$

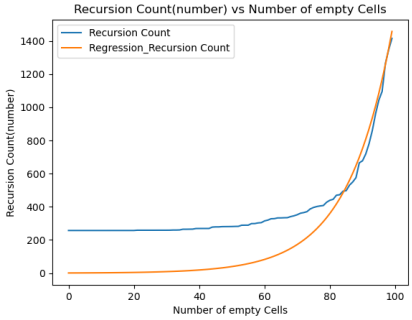
4.0.3.2 Recursion Count



(v) 4x4 board. $y = exp(0.24 * x)$

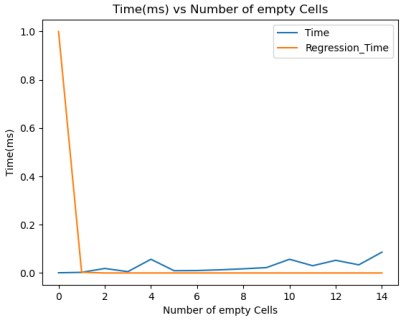


(w) 9x9 board. $y = exp(0.18 * x)$

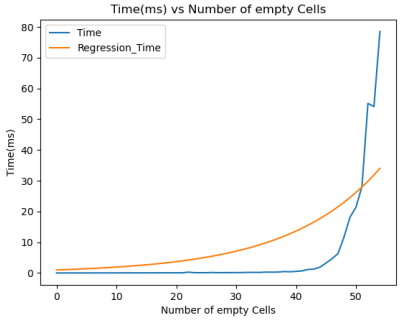


(x) 16x16 board. $y = exp(0.07 * x)$

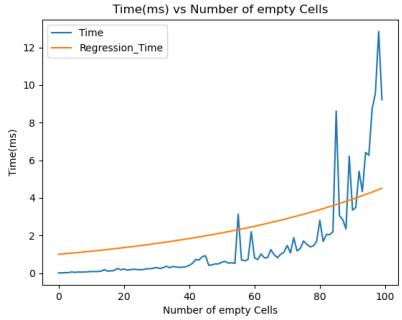
4.0.3.3 Time



(y) 4x4 board. $y = exp(-5.9 * x)$



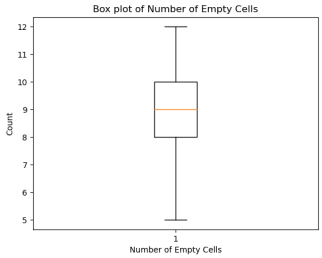
(z) 9x9 board. $y = exp(0.07 * x)$



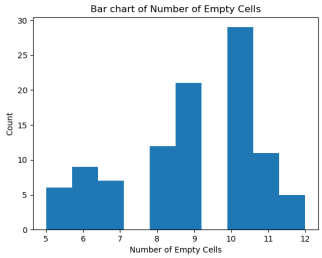
(aa) 16x16 board. $y = exp(0.02 * x)$

4.0.4 Boards

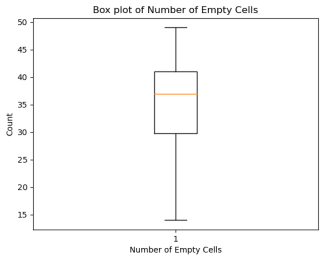
4.0.4.1 Number of Empty Cells



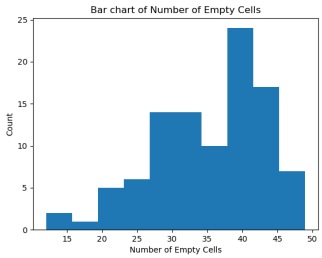
(ab) 4x4 board.



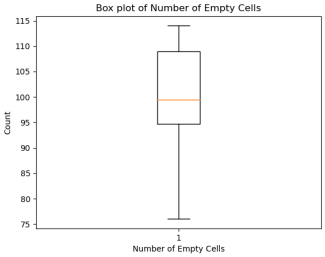
(ac) 4x4 board.



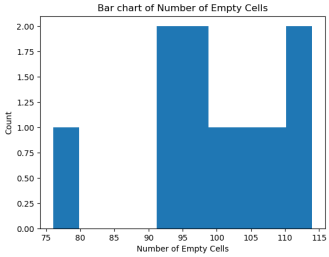
(ad) 9x9 board.



(ae) 9x9 board.

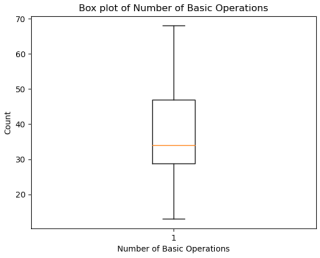


(af) 16x16 board.

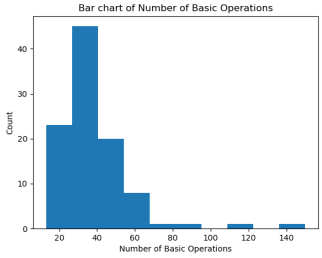


(ag) 16x16 board.

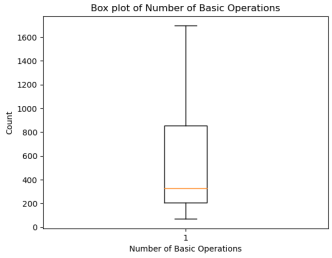
4.0.4.2 Number of Basic Operations



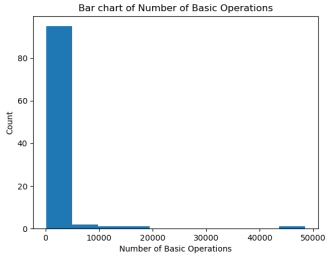
(ah) 4x4 board.



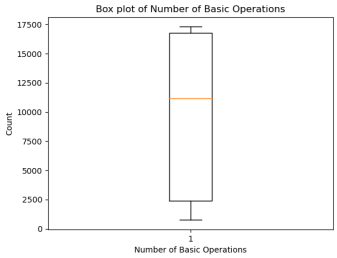
(ai) 4x4 board.



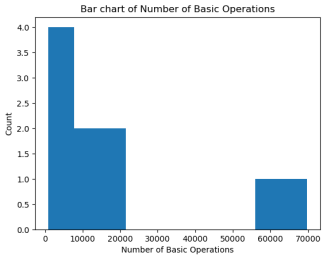
(aj) 9x9 board.



(ak) 9x9 board.

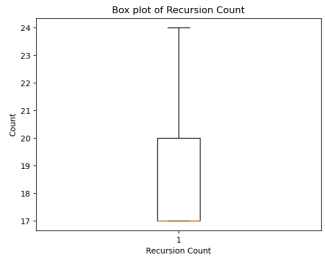


(al) 16x16 board.

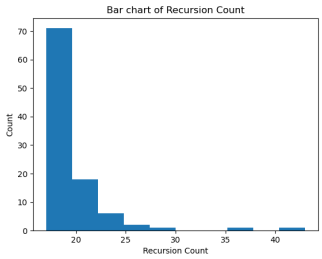


(am) 16x16 board.

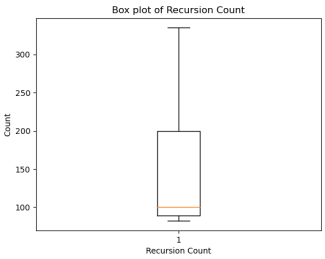
4.0.4.3 Recursion Count



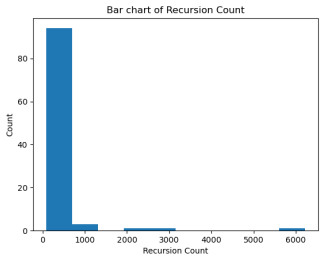
(an) 4x4 board.



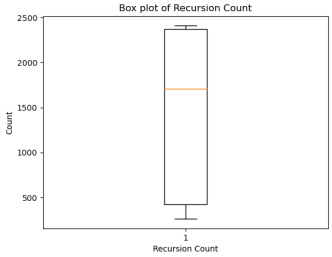
(ao) 4x4 board.



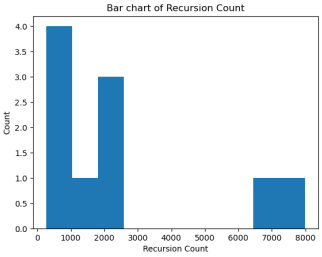
(ap) 9x9 board.



(aq) 9x9 board.

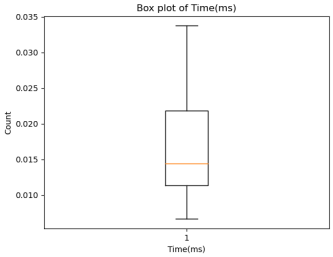


(ar) 16x16 board.

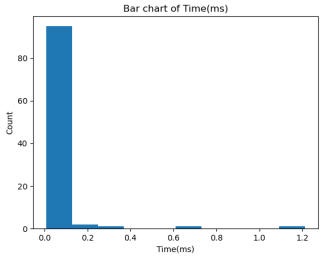


(as) 16x16 board.

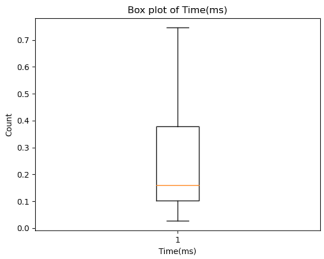
4.0.4.4 Time



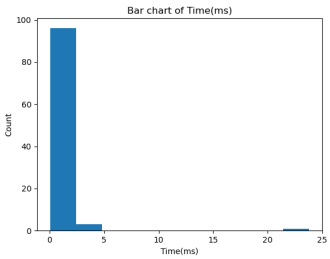
(at) 4x4 board.



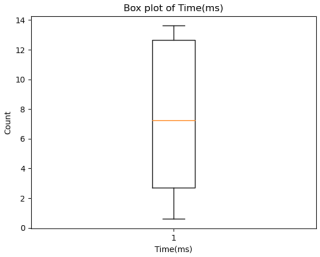
(au) 4x4 board.



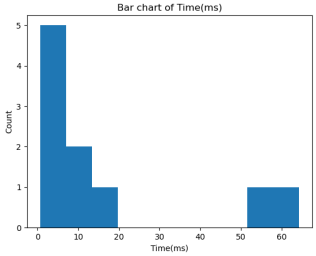
(av) 9x9 board.



(aw) 9x9 board.

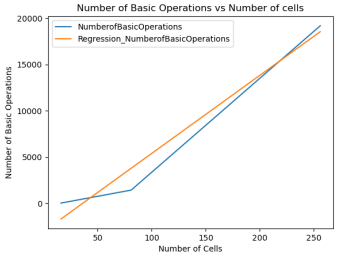


(ax) 16x16 board.



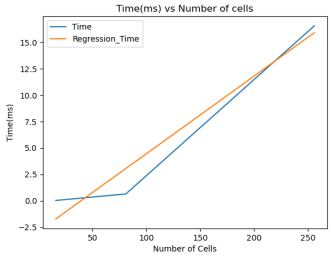
(ay) 16x16 board.

4.0.4.5 Single graphs



(az) $y = 84.22 * x + -3027.35$

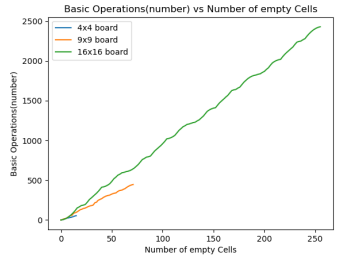
(ba) $y = 10.44 * x + -327.23$



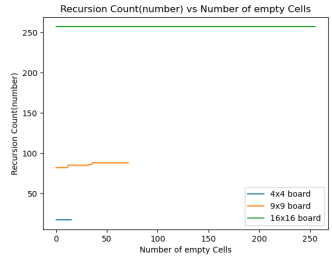
(bb) $y = 0.07 * x + -2.89$

4.0.5 Combining the Graphs

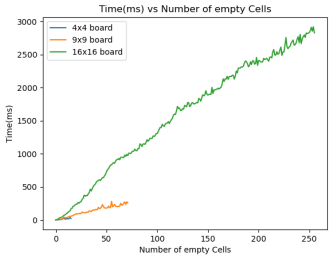
4.0.5.1 Best Case



(bc) Best Case

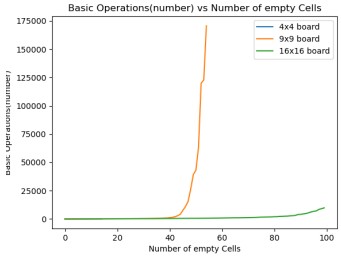


(bd) Best Case

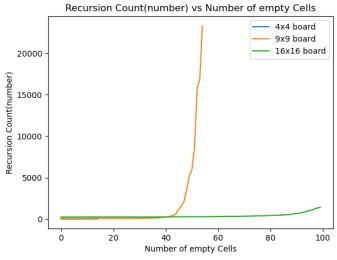


(be) Best Case

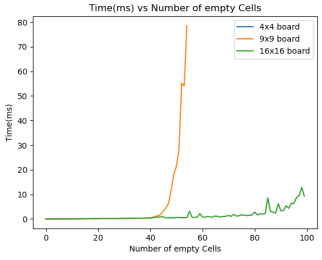
4.0.5.2 Average Case



(bf) Average Case

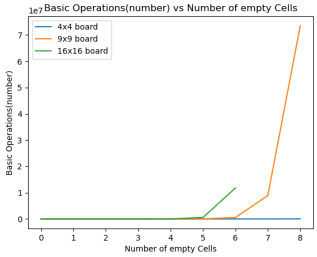


(bg) Average Case

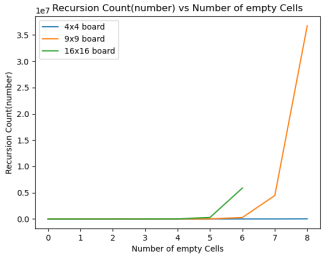


(bh) Average Case

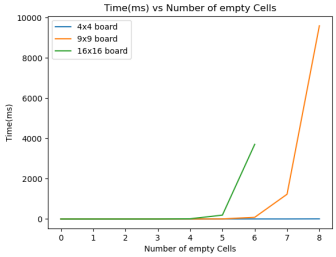
4.0.5.3 Worst Case



(bi) Worst Case



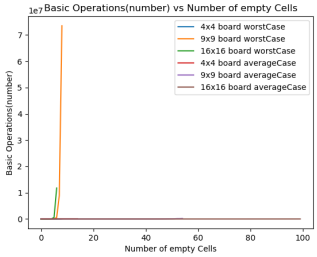
(bj) Worst Case



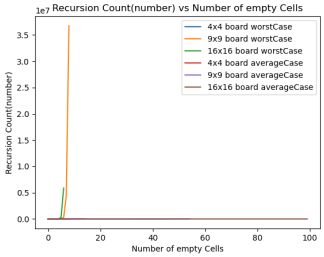
(bk) Worst Case

4.0.6 Combining the Graphs V2

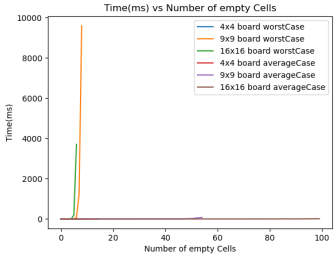
4.0.6.1 On one Axis



(bl) Basic Operations



(bm) Recursion Count



(bn) Time

4.0.7 Statistical analysis

For more information on what statistical techniques we used to fit the curves, as well as the errors we obtained, please view Appendix A.

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.

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 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.
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 4x4, 9x9 and 16x16 boards). This led to some inaccurate fitting. It took too long to successfully generate $25x25$ boards, and $1x1$ boards are trivial.

When evaluating the algorithm on realistic sudoku boards, we see that it performs relatively consistently Even solving 16x16 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 good 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 results where less conclusive, leading us to not be fully certain that our results are correct.

7 Conclusion

In conclusion, we see that the theoretical analysis holds for the sudoku puzzle, but 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.
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] 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 Acknowledgments

9 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.
We mostly used linear regression, and just fit $(n^2)^x$ vs y in the worst case. The average case required us to fit a non linear curve, which we accomplished using the python package Scipy [4] 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.

Case	x	y	Size	a	b	r	σ	equation
best	# of Empty Cells	Basic Operations	4	3.6412	-1.0588	0.9979	0.0635	$y = a * x + b$
best			9	6.448	-1.4741	0.9983	0.0447	$y = a * x + b$
best			16	9.5659	-8.4047	0.9997	0.0157	$y = a * x + b$
best		Recursion Count	4		17.0			$y = a * x + b$
best			9	0.0971	82.5947	0.9064	0.0054	$y = a * x + b$
best			16		257.0			$y = a * x + b$
best		Time	4	1.9536	0.8255	0.7868	0.4096	$y = a * x + b$
best			9	3.6957	6.5639	0.9778	0.0947	$y = a * x + b$
best			16	11.3266	139.1779	0.9934	0.0819	$y = a * x + b$
worst		Basic Operations	4	0.802	759.1682	0.9889	0.0456	$y = a * 4^x + b$
worst			9	1.7085	32157.8359	0.9999	0.0078	$y = a * 9^x + b$
worst			16	0.7028	-27216.4124	0.9999	0.0039	$y = a * 16^x + b$
worst		Recursion Count	4	0.4009	393.4174	0.9889	0.0228	$y = a * 4^x + b$
worst			9	0.8543	16157.543	0.9999	0.0039	$y = a * 9^x + b$
worst			16	0.3514	-13353.6395	0.9999	0.002	$y = a * 16^x + b$
worst		Time	4	0.0001	0.2742	0.9718	0.0	$y = a * 4^x + b$
worst			9	0.0002	12.544	0.9998	0.0	$y = a * 9^x + b$
worst			16	0.0002	-7.1004	0.9999	0.0	$y = a * 16^x + b$
average		Basic Operations	4		0.3131		0.0097	$y = exp(b * x)$
average			9		0.221		0.0006	$y = exp(b * x)$
average			16		0.0929		0.0002	$y = exp(b * x)$
average		Recursion Count	4		0.2407		0.0207	$y = exp(b * x)$
average			9		0.1824		0.0008	$y = exp(b * x)$
average			16		0.0736		0.0006	$y = exp(b * x)$
average		Time	4		-5.8954		96.9075	$y = exp(b * x)$
average			9		0.0653		0.0023	$y = exp(b * x)$
average			16		0.0152		0.001	$y = exp(b * x)$
Boards	Number of Cells	Numberof Basic Operations	0	84.2221	-3027.3467	0.9802	17.0085	$y = a * x + b$
Boards		Recursion Count	0	10.4401	-327.2307	0.9859	1.774	$y = a * x + b$
Boards		Time	0	0.0734	-2.8852	0.9732	0.0174	$y = a * x + b$

Table 1: A table our our regression coefficients and errors