Sudoku Backtracking Solver

CSCI 5454 – Spring 2016

Sneha Parmar

Course Project Report

1. Introduction :

This report presents the algorithm used to solve the famous mathematical puzzle Sudoku. Sudoku, which is also called Number Place is a logic based, combinatorial number placement puzzle. The solver used uses the method of backtracking. The report discusses this backtracking algorithm, some related work describing approaches other than backtracking, correctness of the algorithm, time complexity analysis with results and future work.

1.1 Problem Definition

The idea behind the most common version of a Sudoku puzzle i.e. a 9x9 grid is that in the solved Sudoku puzzle a 9x9 matrix having 9 boxes of size 3x3, following constraints must be satisfied:

a. Every row should have unique numbers from 1 to 9 (each number should appear only once and at least once without being repeated)
b. Every column should similarly have unique numbers ranging from 1 to 9
c. Every 3x3 box should similarly have unique numbers ranging from 1 to 9

The given puzzle generally has a few missing cells from the completed grid such that there exists a solution and generally the solution is unique. The cells which are filled are referred to as hints and cannot be modified in any way as they are *fixed* and constitute the puzzle.

The algorithm implemented tries to solve this problem of finding what values should go into each of the empty cells using the hints provided so that the resulting grid satisfies the above three constraints.

Naturally, this problem could be extended to *n by n* grids where n should have an integer square root. The sizes of the puzzles analyzed in the implementation range from 4 x 4 to 16 x 16 sizes of grids.

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

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

Example of a Sudoku puzzle and its solution [1]

In general, it could be defined as a puzzle game made up of a board of $n^2 \times n^2$ cells, divided in n×n boxes, containing whole numbers in the range 1 to n^2. The goal being to fill the cells so that each row, column and box contain all the numbers ranging from 1 to n^2 with the restriction that there must be exactly one number each for each row, column and box.

1.2 Related work

Various methods have been applied to solve Sudoku previously such as genetic algorithms, simulated annealing, particle swarm optimization, harmony search. [2] combines human intuition and optimization to solve Sudoku problems. It uses a set of heuristic moves, incorporating human expertise, to solve Sudoku puzzles. [3] presents a method to solve the puzzle using Boolean algebra. [4] describes some of the brute force methods used for solving Sudoku puzzles.

The Sudoku puzzle can also be formulated as a integer linear program where every cell has 9 variables (for a 9x9 grid) such that the constraints would be that one and only one of these should be 1 which can be formulated as the sum of these which should then equal 1. There would then be in addition to 81 such constraints (1 for each cell), additional constraints for columns and rows – 9 for every column and 9 for every row. These combined could be fed to a solver for ILP to obtain the solution.

The backtracking algorithm discussed in the next section describes and approach where the solver tries to iterate through all possible solutions and backtracks when it finds the constraints are not satisfied at any point.

2. Backtracking Algorithm:

The approach to solving the Sudoku puzzle is implemented by backtracking. Backtracking algorithm can be adapted to solve the Sudoku that iterates over all the possible solutions for the given Sudoku grid. If the solutions i.e. choices assigned to the cells do not lead to the solution of Sudoku, the algorithm discards the solutions and rollbacks to the original solutions and retries again. Hence it is called backtracking.

Every empty cell in the grid is assigned a list of candidate values i.e. the values which the cell could contain given the current state of the grid so that it would still satisfy the 3 constraints. A candidate at the least index in this set (which is also the least valued candidate) is picked from the set of candidate values for the cell and the solver moves to the next empty cell to find its candidate values. Before picking from the candidate values, the candidate set needs to be generated using the constraints. The solver moves from empty cell to another in this manner till in encounters a cell with 0 candidate values. Here the backtracking needs to happen and this is achieved by recursion in the implementation.

The main idea behind this backtracking is that the solver needs to realize a wrong choice of candidate element has been made earlier in the process of solving the grid which has caused an empty cell to have a candidate set of size zero. The solver backtracks by 1, in that it tries to redo the candidate selection for the previous cell. In the process it eliminates the earlier choice made from the candidate set to avoid repeating the mistake. The solver then moves to the cell which initially had 0 candidates. Here it may have solved the problem if the source of the problem was just the one cell behind. However if that is not the case, it may need to backtrack further back. Thus the algorithm performs a depth-first search through the possible solutions where each level of the graph represents the choices for a single cell and the depth of the graph is the number of cells that need to be filled. With a branching factor of *n* and a depth of *m*, finding a solution in the graph has a worst-case performance of O(*n* ^ *m*) as discussed later.

Steps:

1. Generate, for each cell, a list of candidate values by starting with the set of all possible values and eliminating those which appear in the same row, column and box as the cell being examined.
2. Choose one empty cell. If none are available, the puzzle is solved.
3. If the cell has no candidate values, the puzzle is unsolvable.
4. For each candidate value in that cell, place the value in the cell and try to recursively solve the puzzle.

Solver Pseudocode:

bool Solve (grid[][], row, column ,set) //set initially contains all numbers from 1 to 9

if (reached the last element (9,9 for a 9x9 grid)) : the puzzle is solved, return true

*If (grid[row][col]= **fixed**): move to the next cell.*

for 1 to set.size():

eliminate elements from set which are present in 'row', 'column' and current box

if(set.size!=0)

{

grid[row][col]= set[0]; //decide a choice or guess and

move to the next cell ; //this is done by incrementing column value if row is not exhausted or incrementing row value is column is not exhausted

} else

return false; //a wrong choice has been made previously, so go back to the previous cell that was recently filled

A similar algorithm has been also applied in the implementation for the Sudoku grid generator. The difference being that the grid supplied is initially empty and the number selected from the set is selected randomly rather than the number at the least index. This solved grid is then converted to a puzzle by randomly emptying some cells. The rand() function in C is used with varying distributions to make the problem harder or simpler and to find different cases for analysis. So the pseudocode for the generator is :

Generator Pseudocode:

createGrid (grid[][], row, column ,set) //set initially contains all numbers from 1 to 9

if (reached the last element (9,9 for a 9x9 grid)) : the puzzle is created, return true

for 1 to set.size():

eliminate elements from set which are present in 'row', 'column' and current box

if(set.size!=0)

{

grid[row][col]= set[rand(0,set.size())]; //decide a random value to place from the values available by generating a random index

move to the next cell ; //this is done by incrementing column value if row is not exhausted or incrementing row value is column is not exhausted

}

else

return false; //a wrong choice has been made previously, so go back to the previous cell that was recently filled

createPuzzle(grid[][]) //grid[][] is a global

for all Elements from 0 to 81:

bool select = (rand() % 100 > pd) // here pd is the probability distribution which sets the difficulty of the puzzle which can be ideally a value from 30 to 60

if(select)

*set this Element/ cell as **fixed** i.e. set it as a clue*

else

make the element empty

Note that the above implementation when used is used with probability distributions which give unique solutions to the grids generated. For testing grids which need to have very few hints yet have unique solutions, grids from the web [5] were used.

2.1 Correctness

The backtracking algorithm, by its working considers all possible options for every cell which it chooses values for. The algorithm runs through every empty cell and considers all options possible at every cell. It only chooses the options given the three constraints i.e. the number should not be repeated in either – row, column, or subgrid(box). It ensures it only chooses the restricted options by eliminating the values present from either row, column or box from the set of numbers it is considering. If it finds that this set is empty it means that the grid cannot be solved. So it has to backtrack to change an earlier choice it has made. When backtracking through an earlier choice it has made it considers the other option which is viable for the earlier cell similarly. Thus is explores all possible options while deciding values for every empty cell in the grid. Since it indirectly brute forces through all possible options while ensuring that no constraint of Sudoku is violated, if it fills the grid in this way, it is bound to arrive thus at the correct solution if such a solution exists.

The correctness can also be proved by contradiction. Let us assume that there is some solution (s1,s2,s3…si…sm) where m is the number of empty cells in the puzzle. Let us assume this algorithm is not able to find this solution. This is possible if there is a decision point si, where the algorithm backtracks without exploring all the possible candidates of the set at si. But the design of the algorithm is such that it backtracks only if the set of candidates is empty at si. If

that is not the case i.e. if the set is not empty it always explores all the options in the candidate set beginning with the least valued one before it backtracks. Hence if there is some solution it is bound to find it by its exhaustive exploration of the options or candidates.

Alternatively, once it has exhausted all possible candidates in the set for every cell, it could return false when the grid cannot be solved. This is similar to the backtracking analogy of DFS in a tree.

3. Analysis

3.1 Time Complexity

Given a Sudoku grid of nxn with m empty spaces, using the backtracking algorithm it will need to explore 'n' options at 'm' empty spaces in the worst case. Thus the complexity could be n*n*n…. m times = n^m.

Considering 'm' as the current cell to be filled and m-1 be the previous cell to be filled. The recurrence relation could be given by:

T(m) <= n*T(m-1), T(1) = n.

This follows from the fact that for the mth cell, there are n options (of which it may need to go through every option or a limited set), but this is combined with the options which can be obtained till the (m-1)th cell. There is an additional component to this in that the algorithm needs to check at every iteration the cells present in every row, column and the current box. Since the sizes of all of these are 'n' it would be O(n) + O(n) + O(n) = 3*O(n) for every cell. This can be ignored here since this is linear.

For the best case, the algorithm would not need to backtrack. Here different types of cases could be considered. The simplest case would be when m = 1 and it just needs to iterate over every element and if a solution exists it would find that without having to backtrack at any step.

The complexity here is thus O(n^2)

For another best case where it needs to iterate over cells where there may be more than one blank cell, the complexity would be O(n^3) where

T(m) = T(m-1) + O(n^2)

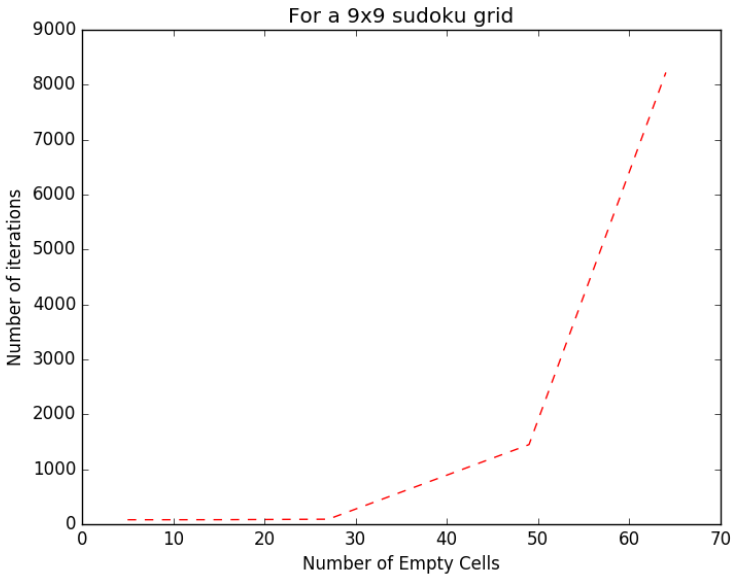
3.2 Space Complexity

The space needed is for the recursive calls and the set used in the implementation which can contain up to 9 numbers. This part is a constant. So only the recursion is considered for the

space complexity. The space needed for the stack is O(m) where m is the number of spaces in the Sudoku grid.

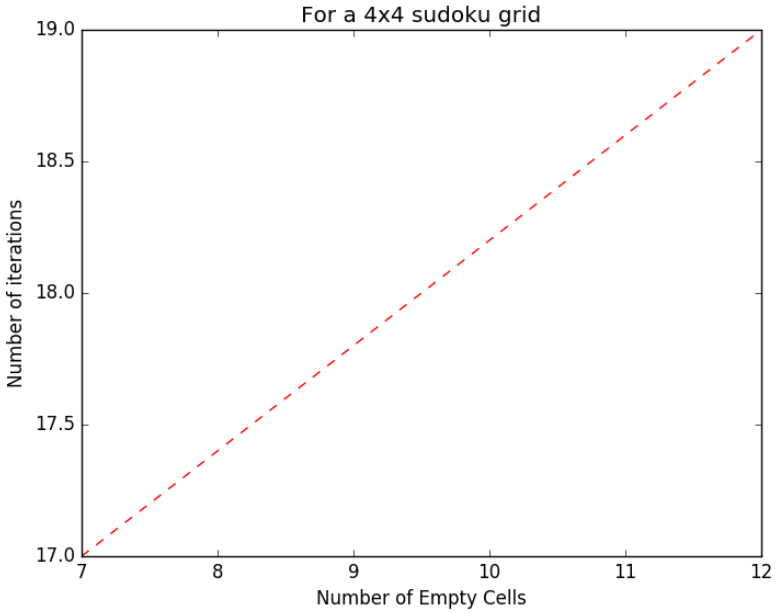
3.3 Results

The random input to the Sudoku solver is the grid generated using the Sudoku grid generator for most inputs. For inputs with fewer clues, grids were used from the web. The Sudoku grid generator can be configured to give Sudoku puzzles with different number of empty cells. IT uses a random number generator with different probability distributions to test with different number of cells. The distributions used for the analysis of the 3x3 grid are 5%, 10%, 20%, 35% 60% and 75% where this % value indicates roughly how much percent of the grid will be empty. It is expected that the time required to solve a sparse grid should be more from the time complexity analysis. The results are recorded over multiple runs for every configuration i.e. around 10 runs are performed and these results are averaged out to get the points plotted. The following is a plot of the number of iterations. Note that number of iterations of the recursive function is used here since the time taken for solving is almost negligible and hence this number of iterations is considered to be indicative of the time taken.



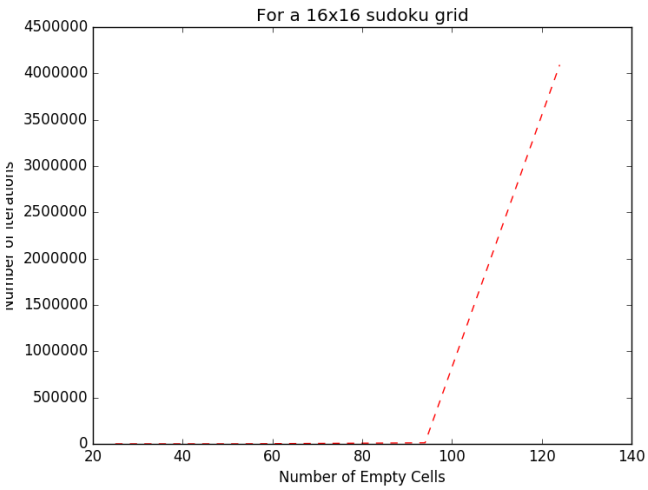
It is observed how the time required has increased for higher number of blank cells. In the individual readings for the grid with more than 75% variation, it was observed that for the worst case where the algorithm needed to backtrack the most, it took 33077 iterations which brought the average up. The other readings have an average of 3252 iterations (rounded off). It is also observed that the best case for this is 81 iterations for distributions less than or equal to 12%. The average case of having 17 clues is given by the 75% probability distribution with iterations of 8223. Here it is observed how the variation in the position of the empty cells significantly effects the runtime. As with roughly the same number of hints, the least runtime is 544 and the highest is 33077.

For the 4x4 grid, the probability distribution used in the generator was not varied much because of the small size of the grid, since the generator cannot easily generate grids with unique solutions with a very high p.d.. The p.ds. used were 50% and 75% and they gave approximately similar number of empty cells.



Here it is observed that the number of iterations has not significantly changed. This could be due to the small size of the grid. It is observed that the best case i.e. the minimum iterations it does is 16.

For the 16x16 grid, the probability distributions explored are 10%,20%, 35% and 50%.

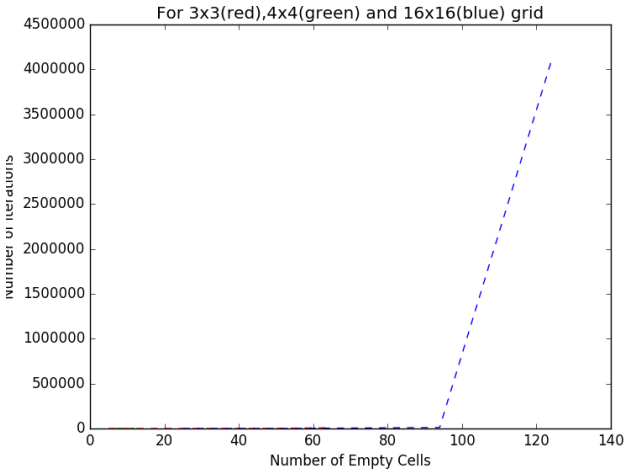


The actual values used to plot the above graph are:

Probability distribution	10%	20%	35%	50%
Average Number of iterations	256	302	11470	40191873
Average number of empty cells	25	53	94	124

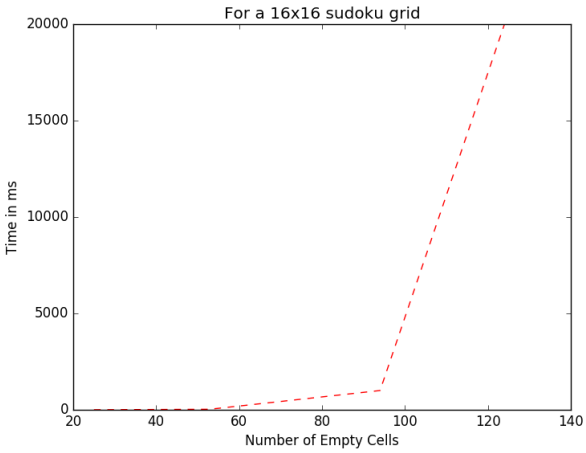
The worst case number of iterations for the 50% distribution was observed at 19170703 iterations for a grid with 129 empty cells and 127 hints. It is observed in general that the performance becomes exponentially worse as the number of empty cells becomes more than the number of filled cells for any size of the Sudoku grid.

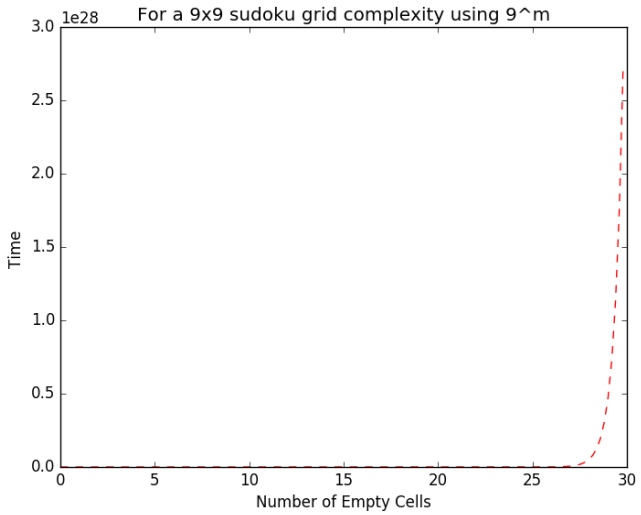
Comparing the performance of the three grids gives:



The 16x16 grid clearly dominates the other two because it has a high ‘n’.

The time based plot for the 16x16 grid is as below –





4. Future work

The solver may achieve a better performance by employing either of the two methods:

1. When moving one empty cells ahead, instead of moving in a row-wise sweeping order, select the cell in the grid with the least number of candidates
2. Select the candidate randomly rather than always selecting the least valued candidate for any cell. For example, if the candidate values are [1,2,4,7,9] instead of selecting 1 and then solving the grid, use some uniform random distribution to select any element from this set i.e. 9 could be the first choice. Such a random distribution would yield better performance which could be observed from analysis.

The backtracking solver thus, has different versions as discussed above which could make it more efficient and these methods can also be analyzed.

5. References:

1. https://en.wikipedia.org/wiki/Sudoku
2. http://sacj.cs.uct.ac.za/index.php/sacj/article/viewArticle/111
3. http://research.ijcaonline.org/volume52/number21/pxc3879024.pdf
4. http://www.cs.bris.ac.uk/Publications/Papers/2000948.pdf
5. https://www.sudoku.ws

Appendix:

Example of the 16x16 grid generated using 50% p.d. and solved by the implementation

5 _ 4 _ _ 13 _ _ 16 8 7 _ 9 _ _ 11

7 _ 16 _ 5 11 12 _ 15 _ 2 _ _ 8 _ 1

6 11 8 1 16 7 _ _ 3 10 12 _ _ 15 _ _

3 _ _ 15 _ _ _ 8 _ _ 5 _ 10 12 16 7

_ 3 _ 9 _ 15 _ _ _ _ 10 _ _ 6 _ _

_ _ 15 11 _ 9 _ 16 _ _ _ 6 3 10 12 8

_ 7 1 _ 10 5 6 3 14 _ 13 8 _ 9 _ _

13 _ _ _ 11 12 8 _ _ 9 3 15 7 1 2 _

10 _ _ _ _ 8 _ 9 4 13 _ 5 _ _ 1 2

14 _ _ 4 _ 6 11 _ _ _ 8 7 15 _ _ 3

15 2 _ 5 _ _ _ _ 9 _ _ _ 14 13 8 6

9 _ _ 6 2 _ _ 13 1 _ _ 12 _ 5 _ _

_ 15 _ 2 3 _ 10 _ _ 6 _ 11 13 _ 7 16

_ 13 10 8 _ _ _ _ 7 _ 16 3 _ _ 6 12

16 _ _ _ 8 2 _ _ 12 _ 4 _ 1 14 15 9

_ _ _ 14 6 _ 7 _ 13 _ 15 2 8 _ 5 10

Solved Grid =

5 12 4 10 15 13 3 6 16 8 7 1 9 2 14 11

7 9 16 13 5 11 12 10 15 4 2 14 6 8 3 1

6 11 8 1 16 7 2 14 3 10 12 9 5 15 4 13

3 14 2 15 4 1 9 8 6 11 5 13 10 12 16 7

8 3 14 9 7 15 1 2 11 12 10 4 16 6 13 5

4 5 15 11 13 9 14 16 2 7 1 6 3 10 12 8

2 7 1 12 10 5 6 3 14 16 13 8 4 9 11 15

13 10 6 16 11 12 8 4 5 9 3 15 7 1 2 14

10 16 11 3 14 8 15 9 4 13 6 5 12 7 1 2

14 1 13 4 12 6 11 5 10 2 8 7 15 16 9 3

15 2 12 5 1 10 4 7 9 3 11 16 14 13 8 6

9 8 7 6 2 3 16 13 1 15 14 12 11 5 10 4

12 15 5 2 3 14 10 1 8 6 9 11 13 4 7 16

1 13 10 8 9 4 5 15 7 14 16 3 2 11 6 12

16 6 3 7 8 2 13 11 12 5 4 10 1 14 15 9

11 4 9 14 6 16 7 12 13 1 15 2 8 3 5 10