Design and Implementation of Algorithms to Solve the Sudoku Puzzle

By

Prem Narayan Yadav- 16CS60R68

Under the supervision of

Professor Abhijit Das

Computer Science and Engineering Department

IIT Kharagpur

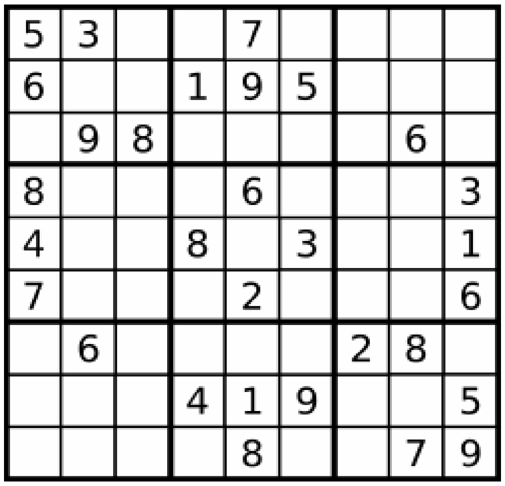
Contents

Abstract

Sudoku is a logical puzzle that has achieved international popularity. There have been a number of computer solvers developed for this puzzle. Various Artificial Intelligence techniques such as constraint propagation, forward checking, minimum remaining heuristic,etc have been developed to solve constraint satisfaction problems. The objective of this project is to optimize the algorithm for solving sudoku puzzles using artificial techniques combined with a heuristic approach to prune the search space of the puzzle.

Introduction

The standard Sudoku Puzzle consists of a 9 X 9 grid, divided into nine 3 X 3 boxes. Each of the 81 squares must be filled in with a number between 1 and 9. For a solution, each row, column and sub-block must contain all of the numbers from 1 to 9.



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

A Sudoku can vary in difficulty depending on various aspects including number of given numbers. The above figure gives an example of a basic Sudoku instance with 30 given numbers.

In general, a sudoku grid can be of size of N^2 * N^2 with sub-blocks of size N * N where for a solution, each row, column and sub-block must contain numbers from 1 to N without repeating. For example a sixteen by sixteen puzzle can be created with sixteen four by four boxes. In a sixteen by sixteen puzzle the rows, columns and boxes are filled with the numbers one to sixteen.

Backtracking search

A simple way to solve the sudoku problems is to use a backtracking algorithm in which we put a value in first unfilled place and check if it is possible, If possible we move forward to the next unfilled process and if it is not possible we track back and change its value and continue. This method is guaranteed to find a solution if there is one, as it will try each possible case. This algorithm is very effective for 4 X 4 puzzles. Unfortunately for 9 X 9 puzzles, there are nine possibilities for each square. This means if there are **N** unfilled places, there are roughly **9 ^ (81 - N)** possible cases which are to be checked. For instance, if we are given 20 numbers in the sudoku grid, then in worst case we have to check for 9 ^ 61 cases which is roughly equal to **1.617 * 10 ^ 58**. Obviously this version of backtracking search is not going to work for 9 X 9 puzzles.

In general, We can see that for a N X N Sudoku grid, we can verify the correctness of the solution in polynomial time but we can’t guarantee a solution in polynomial time, which keeps the problem in a class of **NP-Complete** problems.

Optimization Techniques

Since backtracking algorithm takes much time, we must optimize the algorithm in order to make it faster. Solving sudoku is an important Constraint Satisfaction Problem. Fortunately, several Artificial Intelligence Techniques have been developed for these type of problems. Some of them are :

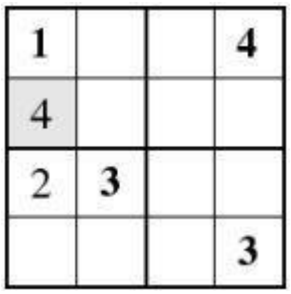
I. Forward Checking

The first improvement on backtracking search is forward checking. Previously in backtracking search, we had to place a value and then check for conflicts. Instead it is easier to just maintain a list of which possible values each square can possibly have given the other numbers that have been assigned. Then when the values are being assigned to that square, only consider the ones that do not directly conflict with the other already placed numbers. For a 9 X 9 puzzle forward checks can be stored in a nine by nine by nine boolean array. Basically each square has its own array of nine boolean values corresponding to each of the numbers that could possibly go in that square. For example, if the third value in the array is set to false, then that square cannot contain a three.

Maintaining these lists is simple. Whenever a new value x is assigned is assigned, go to every other square in the same row, column and box and mark false in its array for value x. Storing this information can be used in two ways. First, it can be used to verify that no value is ever assigned that directly conflicts with another assigned value. Second, if the array for any square contains all false values, then there is no possible value for that square and the most recently assigned value is wrong. Using forward checking the backtracking search can now solve size 9 X 9 puzzles.

II. Constraint Propagation

Forward checking can only catch conflicts right before they cause a certain branch to fail. It is possible to detect errors even earlier and prune off entire branches. Consider the following 4 X 4 puzzle:



1			4
4			
2	3		
			3

It may seem like a good idea to place a four in the shaded box. It doesn't immediately conflict with any other locations and after placing it all of the squares still have possible values. Next look at the square right below the shaded one, it must be a two. Filling in a two there, however, leaves the lower left square with no possible value. This may not seem like a big deal, it only takes an extra two assignments to realize that the four was wrong, but what if the two isn't the next assignment. If the search is moving from left to right across the rows, it will assign the three empty squares to the right of the shaded one first.

Depending on the possible values that these squares have, there may be up to three layers of branching before reaching the conflict. Each branch must now fail separately before the search realizes that the four was a bad choice. The result is an exponential increase in the time needed for the search to realize that the four was a bad choice. The solution is to assign values that only have one possible choice immediately. This is known as constraint propagation. After each value is assigned by the search algorithm, constraint propagation iterates through the squares assigning values to squares with only one possible value. If a square has no possible values, the algorithm fails and returns to the search which re-selects the last value. If multiple values are assigned by constraint propagation, then they are all repealed at once upon a fail. In the example above after the four is assigned, constraint propagation realizes that the space below the four must be a two. It then notices that the lower left corner has no possible value and fails, returning to the search, which chooses another value for the shaded square.

III. Minimum Remaining Values

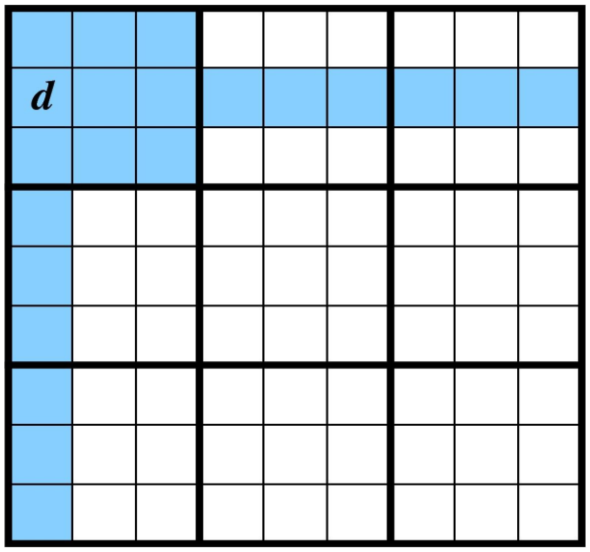
The "Minimum Remaining Values" heuristic is an important heuristic for optimizing solutions of Constraint Satisfaction Problems. According to this heuristic, while choosing a value for a variable, we select the variable which has least number of remaining legal values so that the remaining probability of choosing an incorrect assignment is lesser. For example, there are 3 choices (1 correct + 2 incorrect) for variable x1 and 2 choices (1 correct + 1 incorrect) for variable x2, so if we select x1 for an assignment, the probability of incorrect assignment will be 2/3 while if we select x2 for an assignment, the probability of incorrect assignment will be 1/2 which is lesser.

Heuristics for optimized search

We can use various heuristics to further optimize our algorithm. Here we explain such heuristics which optimize our algorithm for finding a solution for sudoku.

I. Presence Rule

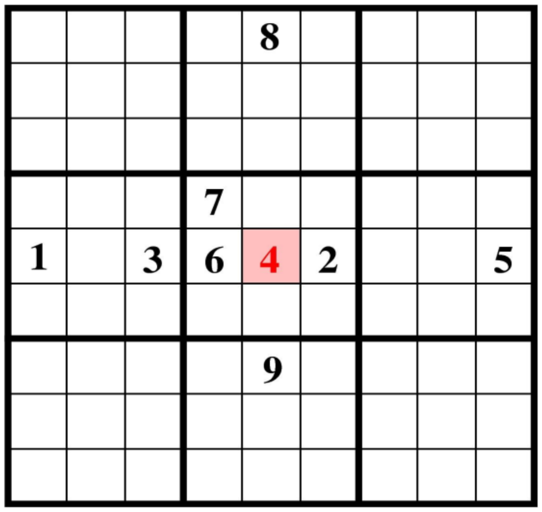
This is the basic rule. By this rule, if any number is already assigned at a position, then this number cannot be valid for any other positions in the same row, column or sub-block. For example, consider the sudoku grid



In the above example, as number ‘d’ is assigned to 2nd row, 1st column , now number ‘d’ cannot be assigned in the shaded region.

II. Absence Rule 1

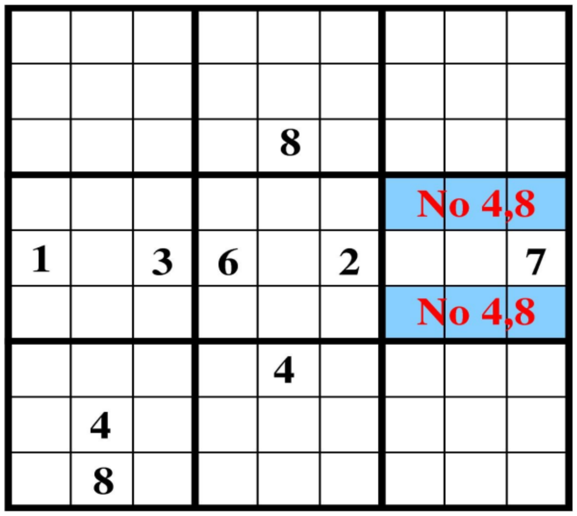
If due to row constraints, column constraints and sub-block constraints, a position has only one legal value, we assign it. For example consider the sudoku grid



In the above example, the position at 5th row, 5th column has only legal value ‘4’ due to row , column and sub-block constraints, so we assign it ‘4’.

III. Absence Rule 2

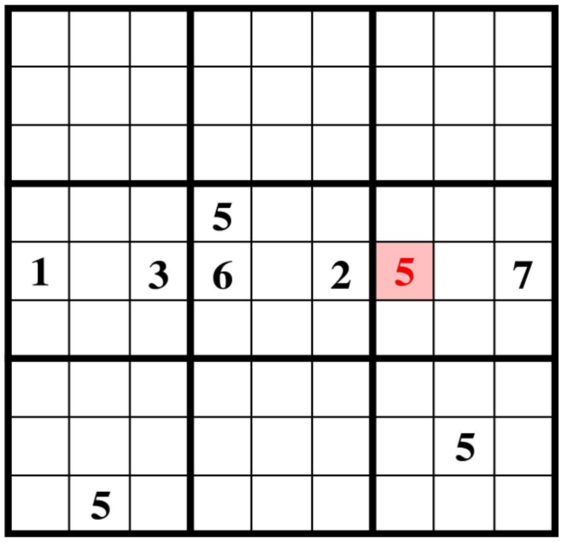
If due to row constraints, two numbers are allowed in only two squares in the same row, these two numbers cannot be allowed in any other position in the corresponding sub-block. This also applies for columns as well. For example consider the sudoku grid



In the above example, in the 5th row number ‘4’ and number ‘8’ are allowed only in the 7th and 8th columns, so, the 4th row and 6th row of the corresponding sub-block must not contain numbers ‘4’ and ‘8’.

IV. Absence Rule 3

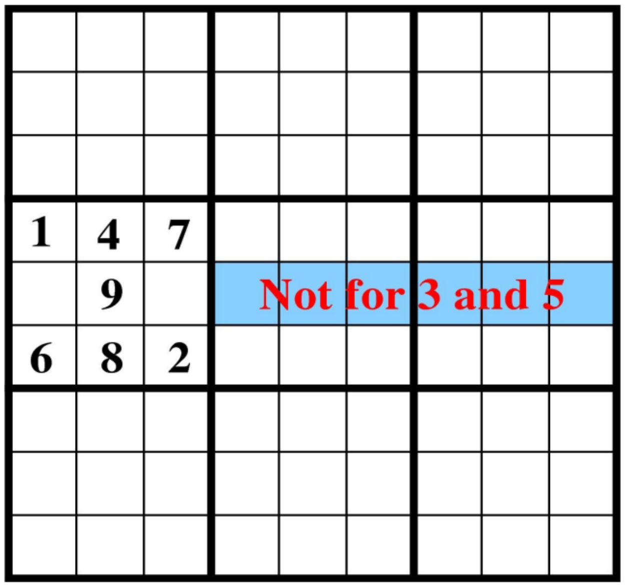
If a number is allowed in only one position in a row, put it there. This also applies for columns and sub-blocks as well. For example consider the sudoku grid



In the above example, in the 5th row number ‘5’ can be at only one possible position which is 5th row and 7th column, so we put it there.

V. Pattern Rule 1

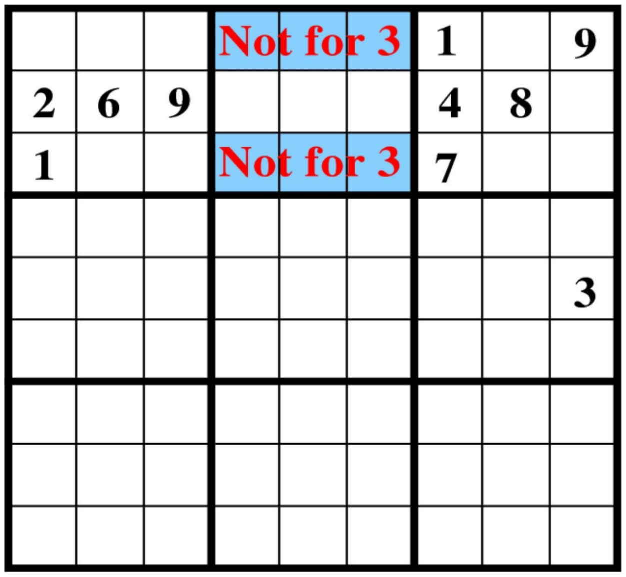
If due to sub-block constraints, two numbers can only be allowed in the same row in same sub-block, these two numbers cannot be in any other squares in that row. This also applies to columns and sub-blocks as well. For example consider the sudoku grid



In the above example, in the 5th row, due to sub-block constraints number ‘3’ and number ‘5’ can only be allowed in 1st and 3rd columns of the same row. So, numbers ‘3’ and ‘5’ cannot be allowed in any other position in the same row.

VI. Pattern Rule 2

If due to row constraints, a number can be in only those squares which are in the same row and same sub-block, it cannot be in any other rows of the same sub-block. This also applies to columns as well. For example consider the sudoku grid



In the above example, for the 2nd row number ‘3’ has 3 possible positions which are all in the same sub-block, so the sub-block cannot have number ‘3’ in any other rows of the sub-block.

Implementation

To find a solution to sudoku problem, we select an unassigned position using *Minimum Remaining Values heuristic* and guess a value from the legal values for that position. We have to maintain a list of legal values for each of the 81 positions. Now we perform *forward checking* for this assignment. Assigning the value will lead to deletion of this value from the legal values lists of all the other position sharing same row, column or sub-block with the position which is being assigned.

We also perform *constraint propagation* by which if at any point, the conditions of any heuristic is satisfied, we can prune more search space. If any inconsistency is located, we have to track back and select a different value from the list of legal values for the position and continue. If a heuristic condition is satisfied, it will prune greater numberof search space. This search is a depth first search as we check for each legal value for a position.

In this search we have to copy the lists of legal values at each point of guessing a value at a position because when an inconsistency is located, reverting back changes could be much more complicated as an assignment can lead to many changes via constraint propagation. This search is a depth first search.

Here is the **pseudo code** for the implementation :

```
Assign_and_Propagate( row, column, value)
    Remove numbers other than value fromAllowed[row][column]
    if(Check_Heuristic_Conditions() == true)
        Propagate()
DFS_search()
    Find MRV_position
    Guess a value
    If(Assign_and_Propagate(MRV_position, guessed value) == true)
        DFS_search()
    Else
        Try different value
SolveSudoku()
    Initialize 81 lists with Allowed values = 1 to 9
    Assign_and_Propagate(initial values)
    DFS_search()
```

Integer Linear Programming

Here we discuss another method for solving Sudoku Puzzles. The key idea is to transform the puzzle from a square 9 X 9 grid to a cubic 9 X 9 X 9 array of binary values (0 or 1). The value of Array[i][j][k] indicates whether cell referred by the ith row and the jth column is occupied by the value k or not. If the value is 0, then the above place is occupied by the value k and if the value is 1, then that place is not occupied by the value k.

The objective function is not needed here, and might as well be 0. The problem is really just to find a feasible solution, meaning one that satisfies all the constraints. Now we need to write all the constraints and give it to an Integer Linear Programming solver. For the above representation of Sudoku constraints, we can write the constraints as :

1. Each cell must have only one value, i.e.

$$\sum_{k=1}^{9} x(i, j, k) = 1 \qquad \text{for each } i \,\&\, j$$

2. Each row must contain all unique digits, i.e.

$$\sum_{j=1}^{9} x(i, j, k) = 1 \qquad \text{for each } i \,\&\, k$$

3. Each column must contain all unique digits, i.e.

$$\sum_{i=1}^{9} x(i, j, k) = 1 \qquad \text{for each } j \,\&\, k$$

4. Also, Each 3 X 3 box must contain all unique digits,i.e.

$$\sum_{i=1}^{3}\sum_{j=1}^{3} x(i, j, k) = 1 \qquad \text{foreach k \& 3 X 3 box}$$

The ILP solver checks for the feasibility of all the constraints and finds a solution.

Simplex Algorithm

This algorithm is a basic algorithm for solving Linear Programming problems. It uses the fact that for a linear program in standard form, if the objective function has a maximum value on the feasible region then it has this value on one of the extreme points. This in itself reduces the problem to a finite computation since there is a finite number of extreme points, but the number of extreme points can be extremely large for most of the linear programming problems.

It is also known that if an extreme point is not a maximum point of the objective function then there is an edge containing the point so that the objective function is strictly increasing on the edge moving away from the point. If the edge is finite then the edge connects to another extreme point where the objective function has a greater value, otherwise the objective function is unbounded above on the edge and the linear program has no solution. The simplex algorithm applies this insight by traversing along edges to extreme points with greater and greater objective values. This continues until the maximum value is reached or an unbounded edge is visited, concluding that the problem has no solution.

The geometrical operation of moving from a basic feasible solution to an adjacent basic feasible solution is implemented as a *pivot operation*. The simplex algorithm proceeds by performing successive pivot operations which each give an improved basic feasible solution. The choice of pivot element at each step is largely determined by the requirement that this pivot improve the solution.

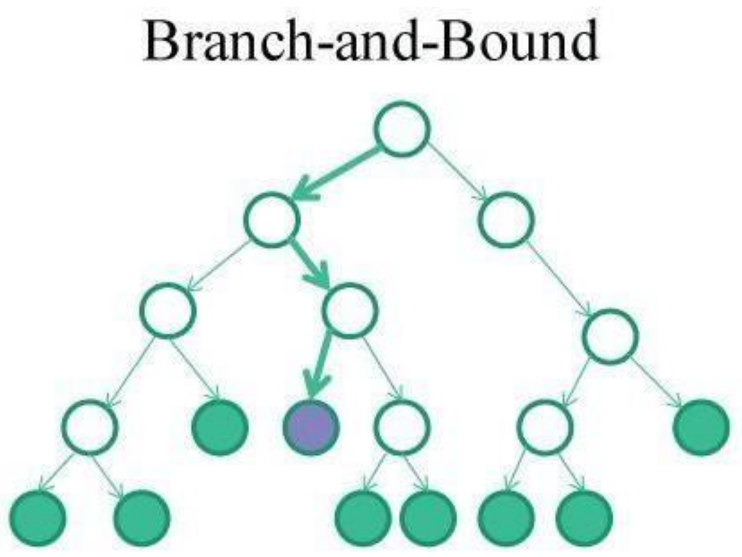
We will use this algorithm to find the solution for the LP relaxation problem of the corresponding Integer Linear programming problem in the below section.

Branch and Bound

Integer Linear Programming(ILP) problems are generally solved using a linear-programming based branch-and-bound algorithm. Basic LP-based branch-and-bound can be described as follows.

We begin with the original ILP. Not knowing how to solve this problem directly, we remove all of the integrality restrictions. The resulting LP is called the ***linear-programming relaxation*** of the original ILP. We can then solve this LP. If the result happens to satisfy all of the integrality restrictions, even though these were not explicitly imposed, then we have been quite lucky. This solution is an optimal solution of the original ILP, and we can stop. If not, as is usually the case, then the normal procedure is to pick some variable that is restricted to be integer, but whose value in the LP relaxation is fractional. For example, suppose that this variable is x and its value in the LP relaxation is 5.7. We can then exclude this value by imposing the restrictions $x \leq 5.0$ and $x \geq 6.0$. If the original MIP is denoted P_0, then we might denote these two new MIPs

by P_1, where $x \leq 5.0$ is imposed, and P_2, where $x \geq 6.0$ is imposed. The variable x is then called a ***branching variable***, and we are said to have *branched* on x, producing the two sub-MIPs P_1 and P_2.
It should be clear that if we can compute optimal solutions for each of P_1 and P_2, then we can take the better of these two solutions and it will be optimal to the original problem, P_0. In this way we have replaced P_0 by two simpler ILPs. We now apply the same idea to these two ILPs, solving the corresponding LP relaxations and, if necessary, selecting branching variables. In doing so, we generate what is called a ***search tree***. The MIPs generated by the search procedure are called the *nodes* of the tree, with P_0 designated as the *root node*. The leaves of the tree are all the nodes from which we have not yet branched. In general, if we reach a point at which we can solve or otherwise dispose of all leaf nodes, then we will have solved the original ILP.



Branching alone would amount to brute force enumeration of candidate solutions and testing them all. To improve on the performance of brute-force search, a Branch and Bound algorithm keeps track of ***bounds*** on the minimum that it is trying to find, and uses these bounds to "prune" the search space, eliminating candidate solutions that it can prove will not contain an optimal solution.

A Branch and Branch algorithm performs a top-down recursive search through the tree of nodes formed by the branch operation. Upon visiting an node i, it checks whether bound(i) is greater than the lower bound for some other node that it already visited. If so, node i may be safely discarded from the search and the recursion stops. This pruning step is usually implemented by maintaining a global variable that records the minimum lower bound seen among all nodes examined so far.

Using this algorithm we can find a feasible solution for the constraints of our Sudoku puzzle, which gives a solution to the Sudoku puzzle.

Improvements in ILP

We can improve the ILP method by using all the previously mentioned heuristics. Using these heuristics will prune the search space for the ILP solver before submitting the Sudoku ILP problem to the solver. This will largely reduce the time for solving the Sudoku ILP problem.

Also in our case, the number of constraints is much larger than the number of variables. So, converting the Sudoku ILP problem to its dual problem may improve the algorithm further.

Dataset

For testing the working of the algorithm, sudoku problems from the following links were used :

- https://projecteuler.net/project/resources/p096_sudoku.txt

  50 sudoku puzzles, easy to hard, all having unique solutions
- http://norvig.com/top95.txt

  95 hard sudoku puzzles
- http://norvig.com/hardest.txt

  11 very hard puzzles
- http://staffhome.ecm.uwa.edu.au/~00013890/sudoku17

  500 sudoku puzzles with 17 filled positions

Results and Conclusion

ILP problem for Sudoku was solved using Gurobi Optimisation tool, which is one of the best ILP solvers. It was observed that the euler 50 puzzles, ranging from easy to hard, all with unique solutions took an average time of 0.0049 seconds with a maximum of

43. seconds while the backtracking algorithm took an average time of **0.011 seconds** with a maximum of **0.102 seconds**.

ILP solver took an average time of *0.0125 seconds* with a maximum of *0.046 seconds*.

For Norvig's 95 hard puzzles, the average time was 0.05 seconds with a maximum of 0.661 seconds while the backtracking algorithm took an average time of **1.98 seconds** with a maximum of **55.45 seconds**.

ILP solver took an average time of *0.011 seconds* with a maximum of *0.038 second*s.

For norvig's 11 hardest problems the average time was 0.0122 seconds with a maximum of 0.0373 seconds while the backtracking algorithm took an average time of

44. seconds with a maximum of **0.152 seconds**.

ILP solver took an average time of *0.018 seconds* with a maximum of *0.037 seconds*.

For bigger dataset of 500 sudoku puzzles, the following observations were made :

- Average time taken was 0.0758 seconds (**6.8602 seconds** while backtracking)
- The maximum time taken was 1.525 seconds (**117.04 seconds** while backtracking)
- 5 out of 500 sudoku puzzles took more than 1 second while with backtracking **8 out of 500** sudoku puzzles took more than **50 seconds**
- 44 out of 500 sudoku puzzles took more than 0.2 seconds while with backtracking **50 out of 500** sudoku puzzles took more than **20 seconds**

- 95 out of 500 sudoku puzzles took more than 0.1 seconds while with backtracking **320 out of 500** sudoku puzzles took more than **1 second**
- Rest 405 out of 500 sudoku puzzles took less than 0.1 seconds while with backtracking **only 20 out of 500** sudoku puzzles took less than **0.1 seconds**

The results show that using these heuristics improves the algorithm for finding a solution. These heuristics together with *Constraint Propagation, Forward checking* and *Minimum Remaining Values* prune the search space by a bigger margin, which makes this algorithm better and efficient. We can further optimize the algorithm by considering more heuristics but condition checking for those heuristics must be fast to achieve better efficiency. Although the Sudoku problem is a difficult constraint satisfaction problem, it is not completely invulnerable to search methods. Puzzles of size 9 X 9 or less can be solved very quickly and consistently. The backtracking search can consistently solve 9X9 Sudoku puzzles after considering fewer than 200 states. Considering that there are 6,670,903,752,021,072,936,960 valid Sudokupuzzles, searching only 200 of them to find a solution is excellent.

It can be seen that the ILP model with heuristics performs better on an average Sudoku but in case of harder Sudoku puzzles, the Heuristic backtracking model performs better as the pruning is done at each step in the later.

References

- Solving every sudoku puzzle By Peter Norvig http://norvig.com/sudoku.html
- Felgenhauer, Bertram and Frazer Jarvis. “Enumerating possible Sudoku grids.” 20 June 2005 http://www.afjarvis.staff.shef.ac.uk/sudoku/sudoku.pdf
- Taruna Kumari, Preeti Yadav, Lavina , " Study Of Brute Force and Heuristic Approach to solve sodoku" , International Journal of Emerging Trends & Technology in Computer Science (IJETTCS) , Volume 4, Issue 5(2), September - October 2015 , pp. 052-055 , ISSN 2278-6856 http://www.ijettcs.org/pabstract.php?vol=Volume4Issue5(2)&pid=IJETTCS-2015-10-10-17
- Mike Schermerhorn. “A Sudoku Solver” https://www.cs.rochester.edu/u/brown/242/assts/termprojs/Sudoku09.pdf
- Anshul Kanakia and John Klingner. “Methods for Solving Sudoku Puzzles” http://tuvalu.santafe.edu/~aaronc/courses/5454/csci5454_spring2013_CSL5.pdf
- Felgenhauer, Bertram and Frazer Jarvis. “Enumerating possible Sudoku grids.” 20 June 2005

  http://www.afjarvis.staff.shef.ac.uk/sudoku/sudoku.pdf
- Branch and Bound Algorithm https://en.wikipedia.org/wiki/Branch_and_bound
- Simplex Algorithm https://en.wikipedia.org/wiki/Simplex_algorithm