Backtracking Algorithm and Application in Solving Sudoku Problem Recursively

Viet Dung Tran

Faculty of Science, Engineering and Technology

Swinburne University of Technology

Hawthorn, Victoria 3122

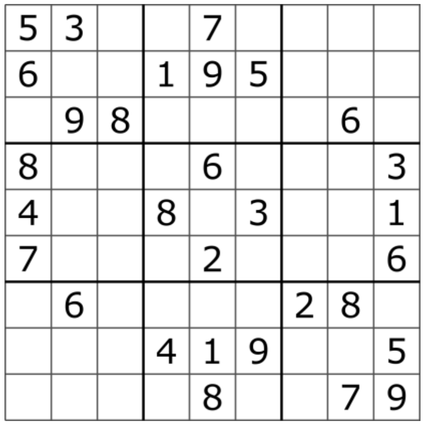
Email: dung.an0412@gmail.com

GitHub: github.com/VietDungTran0412

Abstract

Sudoku puzzle had been becoming one of the most prevalent logical games since it was first introduced in Japan in 1984. The puzzle rules are simple, but the difficulty of this game is universal. The mathematical concept and logical patterns behind the game have always challenged developers to come up with an algorithm that can solve every puzzle regardless of its difficulty. This report discussed the implementation of the backtracking algorithm in Sudoku solver.

1 Introduction



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: Sudoku puzzle

1.1 Backtracking's Goals

Backtracking has been an efficient algorithm in solving recursive problems and combinatorics: *N-Queens problem, Sum of Subsets, Knight's Tour problem* or *Rat in A Maze.*

The goal of backtracking algorithm is almost the same as *Brute-force* approach that optimized the solution by exploring all the possibilities which successfully to satisfy the constraint.

1.2 Scope and Purpose

The paper is aiming at delivering backtracking algorithms through a famous game - Sudoku to those who started studying computer science and those who are concerned about recursive problems especially backtracking algorithms. The explanation and analysis will not be complex. The paper only tried to deliver the fundamentals of backtracking algorithm.

There are two main parts to this paper. In the first part, we are going to discuss the backtracking algorithm in detail. We will analyze the concept behind backtracking. The second part would be implementing backtracking in the Sudoku Solver and analyze how efficient it goes.

2 Theoretical Background

2.1 Sudoku Problem

The Sudoku puzzle is designed of a board of $n^2 \times n^2$ square, divided into $n \times n$ box. To crack the puzzle, players need to fill all the empty squares with values from 1 to 9 which ensure that there is no duplication in terms of columns, rows, and a box of 3×3.

2.2 Recursion Fundamentals

As we mentioned most of the backtracking solution is a form of recursion. The definition of recursion refers to a process in which a recursive process will call itself until it reaches a base case.

Recursion consumes space in stack memory allocation because it is the process of a function calling itself. It is not as good compared to heap but in many problems, recursion is still an effective way to solve many problems.

2.3 Backtracking algorithm

Backtracking algorithm is an improvement of *Brute-forc*e approach. It explores all the possibilities among all available options, but we can always back up as far as needed to reach the previous decision point if it fails to satisfy the constraint. The constraint is created based on the requirements of the problem. Backtracking could be regarded as a selective tree traversal where the root note is the solution and the accurate result would be completed when it reaches the deepest node of the tree (Karumanchi, 2011). This method is also following *Depth-first Search* algorithm, but the tree will not be explored further if it is against the constraint.

The use of backtracking in mathematics and computer science in the real world is various. Although the time complexity of the problem could arise because of optimization, the efficiency of the algorithm is on top level in finding a feasible solution to a decision problem

3 Solving Sudoku

The concept is we are trying to build every possible solution at first by trying digits from 1 to 9 in empty squares then remove the solution if there is any duplication.

3.1 Setting up Constraint

The first step before solving optimization problems by backtracking is setting up constraint. The constraint of backtracking in this case would be each time an empty square is assigned by a solution, the constraint will check if there are any duplicated values in its column, row, and box. If the solution fails to pass the constraint, remove the assigned number.

The constraint now is solid, but it is not enough to solve this problem, but it fails to check all the empty square. The problem said that we must fill all the empty square to solve the problem. Therefore, checking and finding an empty square could be regarded as a step in creating the constraint

3.2 Solving Optimization problem

Backtracking algorithm is designed to solve many optimization problems. In Sudoku problem, the algorithm will build up as many possibilities by recursive calls at each time assigning an empty square by a digit. As backtracking algorithms concept, the further solution will not be achieved if it failed the constraint we had been created.

The number of cases would be extremely high because each empty square must be assigned a value from 1 to 9. If applying *Brute-force* approach for this problem, the number of possibilities from problems in *Figure 1* could be calculated by the following formula:

$$p = \prod_{k=1}^{n} 9_k$$

While p is the number of possibilities, n is the number of empty squares in the problem and k denotes the index of each empty cell. In *Figure 1,* there are 50 empty squares, number of possibilities will be:

$$p = \prod_{k=1}^{50} 9_k = 9^{50}$$

We will not go further in calculating the result because it would be so large. But basically, applying backtracking could reduce such a huge number of possibilities by cutting down the solution that fails to satisfy the constraint.

3.3 Backtracking Implementation in Computation

To generate the problem in programming, first we must create a grid of 9×9. A Sudoku table could be regarded as a matrix of 9×9, and that is why we will initialize the puzzle as a *two-dimensional* array. Unfilled squares will be assigned by 0 and we will add the default values in the appropriate squares.

The algorithm could be computationally implemented by the following steps:

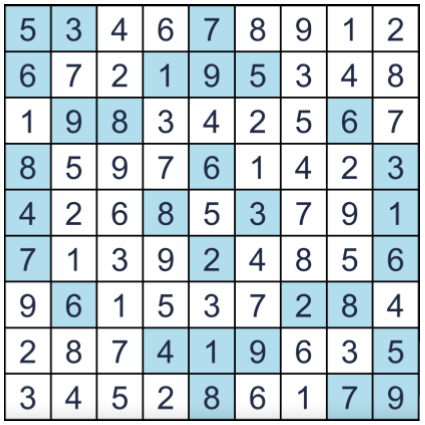
Step 1. Traverse and find all the empty squares on the puzzle.

Step 2. Assign the empty squares by the digits from 1 to 9.

Step 3. Checking if the assigned values successfully satisfy the constraint. If it is not return it values to 0.

Step 4. Recursively call the function to traverse all possibilities until we reach the solution.

Step 5. Return true if the problem has been solved and print out the solution. Otherwise, return false if solution does not exist.



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

Figure 2: Solution for problem in *Figure 1*

The computational implementation of backtracking can solve all the Sudoku problems regardless of their complexity.

3.4 Pseudocode code Implementation

Backtracking implementation would be in *pseudocode* following those above steps. In definition, *pseudocode* is an actual programming language, but it does not follow strict rules for a programming language. There are several differences such as the array would be *1-indexed* instead of *0-indexed*.

- Backtracking :

```
function sudoku_solution(grid)
  cell = {0,0}
  if is_empty_square(grid,cell) is true
    return true
  row = cell[1]
  column = cell[2]
  for i = 1 to 9
    if is_safe(grid,row,col,i) is true
        grid[row][col] = i
        if sudoku_solution(grid) is true
          return true
        else
        grid[row][col] = 0
    return false
```

- Finding empty square:

```
function is_empty_square(grid,cell)
  for row = 1 to grid.length
    for col = 1 to grid[i].length
      if grid[row][col] equals to 0
        cell[1] = row
        cell[2] = col
          return false
  return true
```

- Constraint :

```
function is_safe(grid,row,col,num)
  checked_row=is_row_safe(grid,row,num)
  checked_col=is_column_safe(grid,col,num)
  check_box=is_box_safe(grid,row-
row%3,col-col%3,num)
  if checked_row and checked_col and
checked_box all true
    return true
  else
    return false
function is_row_safe(grid,row,num)
  for col = 1 to grid[col].length
    if grid[row][col] equals to num
      return false
  return true
function is_column_safe(grid,col,num)
  for row = 1 to grid.length
    if grid[row][col] equals to num
      return false
  return true
function is_box_safe(grid,row,col,num)
  size = 3
  for i = 1 to size
    for j = 1 to size
      if grid[row+i][col+j] equals to num
```

```
        return false
  return true
```

3.5 Time Complexity Analysis

As we mentioned, the algorithm could be extremely slow, these two charts below which were made in Python will prove that the time consumption is directly proportional to the complexity of the Sudoku problem

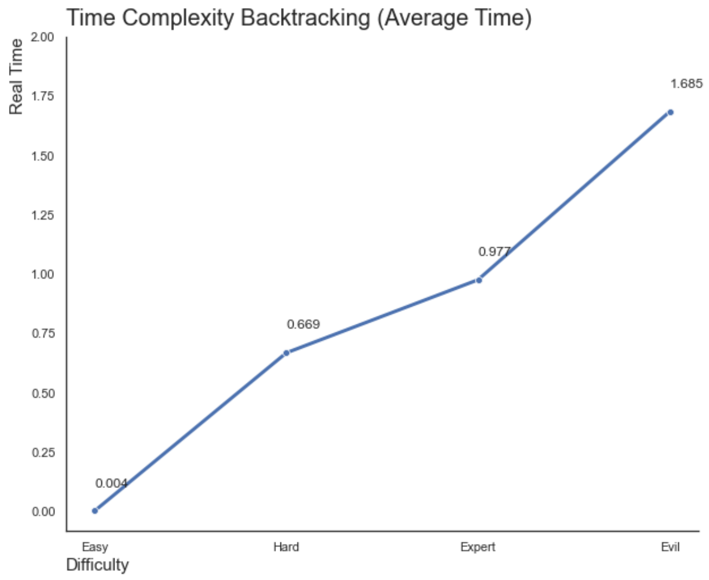


Figure 3: Average Time of Sudoku Solver

The chart in *Figure 3* has shown the average time of Sudoku Solver in the program based on its complexity. The complexity of each program has been measured by *Sudoku.com*. The time complexity of backtracking will increase with the difficulty of the problem. It peaked at 1.685 seconds with "Evil" problems.

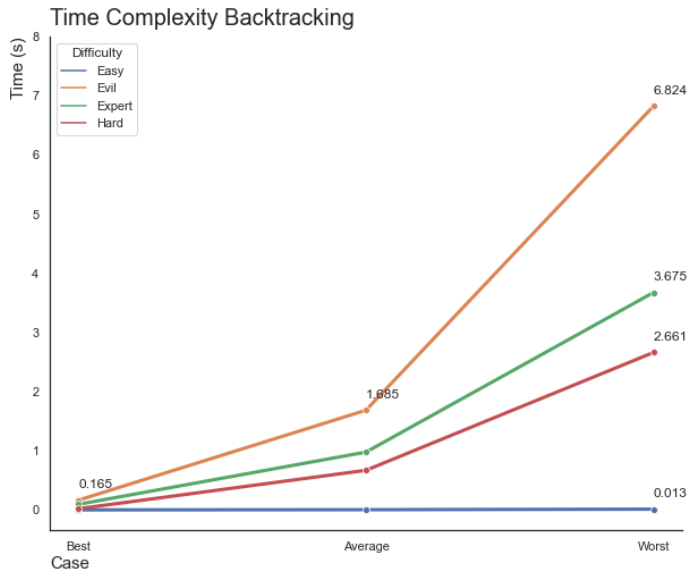


Figure 4: Time Complexity of Sudoku Solver

The chart from *Figure 3*, it has shown the time using backtracking to solve different kinds of problems from the best case to the worst case. Obviously, there has been a dramatic increase from the best case to the worst case of "Evil" problems while the time spent on "Easy" problems remained relatively constant. The "Expert" and "Hard" problems had all shown the same upward trend in the time complexity.

The best-case time complexity of the algorithm could be $O(n^2)$ when it took several steps to find and solve the problem in the grid of $n \times n$. While in the worst case the time could be increased exponentially, the time complexity, in this case, it would be $O(9^{n \times n})$ with nine recursive calls in the solution.

4 Conclusion

4.1 Algorithm discussion

The application of backtracking is various even though the time complexity of its might be high as analysis above. The algorithm is efficient in solving decision problem compared to the use of other recursive algorithm like *Dynamic Programming* and *Greedy*.

In solving certain Sudoku problem, this paper has clearly shown the implementation of backtracking in pseudocode as well as time complexity analysis. The fundamental theorem of backtracking has also been discovered as the way we set up the algorithm. The scope of this paper is limited to discovering the abstract concepts behind because it aims at those who are starting *Data Structures & Algorithms* and those who are beginners in computer science.

4.2 Further Research

There are several approaches for future work. First, the time complexity of the problem could be analyzed further. There are several techniques to measure the time complexity of recursive algorithm such as *Master Theorem for Subtract and Conquer Recurrences*. *Confirming*. These approaches might be abstract within the scope of this paper. Second, we could express the way algorithm worked and the possibilities by using *State Space Tree*. This representation could be efficient while we are trying to analyze the solution of a recursive problem. Research could be spent if it covered those content.

References

Hilary Priestley, Martin Ward. (2003). *A Multipurpose Backtracking Algorithm*. Oxford.

Karumanchi, N. (2011). *Data Structures and Algorithms Made Easy*. CareerMonk Publication