Backtracking | Introduction

Prerequisites :

- Recursion
- Complexity Analysis

Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to the time elapsed till reaching any level of the search tree).

According to the wiki definition,

> ***Backtracking*** *can be defined as a general algorithmic technique that considers searching every possible combination in order to solve an optimization problem.*

How to determine if a problem can be solved using Backtracking?

Generally, every constraint satisfaction problem which has clear and well-defined constraints on any objective solution, that incrementally builds candidate to the solution and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution, can be solved by Backtracking. However, most of the problems that are discussed, can be solved using other known algorithms like *Dynamic Programming* or *Greedy Algorithms* in logarithmic, linear, linear-logarithmic time complexity in order of input size, and therefore, outshine the backtracking algorithm in every respect (since backtracking algorithms are generally exponential in both time and space). However, a few problems still remain, that only have backtracking algorithms to solve them until now.

Consider a situation that you have three boxes in front of you and only one of them has a gold coin in it but you do not know which one. So, in order to get the coin, you will have to open all of the boxes one by one. You will first check the first box, if it does not contain the coin, you will have to close it and check the second box and so on until you find the coin. This is what backtracking is, that is solving all sub-problems one by one in order to reach the best possible solution.

Consider the below example to understand the Backtracking approach more formally,

Given an instance of any computational problem P and data D corresponding to the instance, all the constraints that need to be satisfied in order to solve the problem are represented by C. A backtracking algorithm will then work as follows:

The Algorithm begins to build up a solution, starting with an empty solution set S. **S = {}**

1. Add to S the first move that is still left (All possible moves are added to S one by one). This now creates a new sub-tree s in the search tree of the algorithm.
2. Check if $S + s$ satisfies each of the constraints in C.
   - If Yes, then the sub-tree s is "eligible" to add more "children".
   - Else, the entire sub-tree s is useless, so recurs back to step 1 using argument S.
3. In the event of "eligibility" of the newly formed sub-tree s, recurs back to step 1, using argument $S + s$.
4. If the check for $S + s$ returns that it is a solution for the entire data D. Output and terminate the program.
   If not, then return that no solution is possible with the current s and hence discard it.

Pseudo Code for Backtracking :

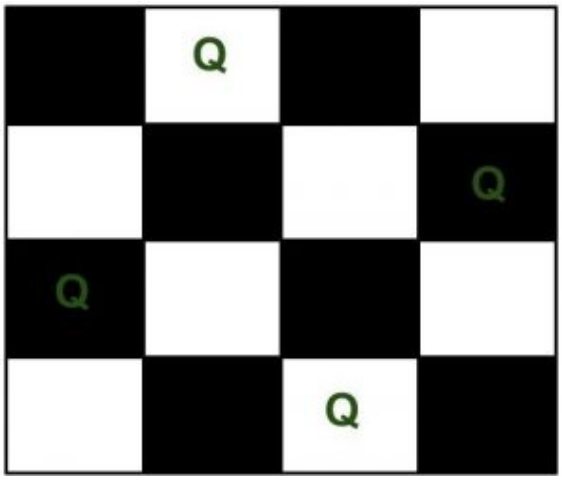
1. Recursive backtracking solution.

```
void findSolutions(n, other params) :
    if (found a solution) :
        solutionsFound = solutionsFound + 1;
        displaySolution();
        if (solutionsFound >= solutionTarget) :
            System.exit(0);
        return

    for (val = first to last) :
        if (isValid(val, n)) :
            applyValue(val, n);
            findSolutions(n+1, other params);
            removeValue(val, n);
```

2. Finding whether a solution exists or not

```
boolean findSolutions(n, other params) :
    if (found a solution) :
        displaySolution();
        return true;

    for (val = first to last) :
        if (isValid(val, n)) :
            applyValue(val, n);
            if (findSolutions(n+1, other params))
                return true;
            removeValue(val, n);
        return false;
```

Let us try to solve a standard Backtracking problem, **N-Queen Problem**.
The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen problem.



The expected output is a binary matrix which has 1s for the blocks where queens are placed. For example, following is the output matrix for the above 4 queen solution.

```
{ 0,  1,  0,  0}
{ 0,  0,  0,  1}
{ 1,  0,  0,  0}
{ 0,  0,  1,  0}
```

Backtracking Algorithm: The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes then we backtrack and return false.

```
1) Start in the leftmost column
2) If all queens are placed
    return true
3) Try all rows in the current column.  Do following for every tried row.
    a) If the queen can be placed safely in this row then mark this [row,
        column] as part of the solution and recursively check if placing
        queen here leads to a solution.
    b) If placing the queen in [row, column] leads to a solution then return
        true.
    c) If placing queen doesn't lead to a solution then unmark this [row,
        column] (Backtrack) and go to step (a) to try other rows.
3) If all rows have been tried and nothing worked, return false to trigger
    backtracking.
```

You may refer to the article on Backtracking | Set 3 (N Queen Problem) for complete implementation of the above approach.

More Backtracking Problems:

- Backtracking | Set 1 (The Knight's tour problem)
- Backtracking | Set 2 (Rat in a Maze)
- Backtracking | Set 4 (Subset Sum)
- Backtracking | Set 5 (m Coloring Problem)

- –> Click Here for More

Recommended Posts:

- N Queen in O(n) space
- Backtracking to find all subsets
- Combinational Sum
- Top 20 Backtracking Algorithm Interview Questions
- Warnsdorff's algorithm for Knight's tour problem
- Recursion
- Traveling Salesman Problem using Branch And Bound
- 0/1 Knapsack using Branch and Bound
- Sudoku | Backtracking-7
- m Coloring Problem | Backtracking-5
- Subset Sum | Backtracking-4
- N Queen Problem | Backtracking-3
- Rat in a Maze | Backtracking-2
- The Knight's tour problem | Backtracking-1
- Write a program to print all permutations of a given string

AayushChaturvedi
I am a problem solving enthusiast and I love competitive programming

If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please Improve this article if you find anything incorrect by clicking on the "Improve Article" button below.

Article Tags : Backtracking | Branch and Bound | Recursion | Algorithms-Backtracking

Practice Tags : Recursion | Backtracking

1

3

To-do Done

Based on **2** vote(s)

Feedback Add Notes Improve Article

Please write to us at contribute@geeksforgeeks.org to report any issue with the above content.

Writing code in comment? Please use ide.geeksforgeeks.org, generate link and share the link here.

Load Comments Share this post!