Introduction to Backtracking – Data Structure and Algorithm Tutorials

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 in time (by time, here, is referred to the time elapsed till reaching any level of the search tree). Backtracking can also be said as an improvement to the brute force approach. So basically, the idea behind the backtracking technique is that it searches for a solution to a problem among all the available options. Initially, we start the backtracking from one possible option and if the problem is solved with that selected option then we return the solution else we backtrack and select another option from the remaining available options. There also might be a case where none of the options will give you the solution and hence we understand that backtracking won't give any solution to that particular problem. We can also say that backtracking is a form of recursion. This is because the process of finding the solution from the various option available is repeated recursively until we don't find the solution or we reach the final state. So we can conclude that backtracking at every step eliminates those choices that cannot give us the solution and proceeds to those choices that have the potential of taking us to the solution.

According to the wiki definition,

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

There are three types of problems in backtracking –

1. Decision Problem – In this, we search for a feasible solution.
2. Optimization Problem – In this, we search for the best solution.
3. Enumeration Problem – In this, we find all feasible solutions.

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 Dcorresponding to the instance, all the constraints that need to be satisfied in order to solve the problem are

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

1. Add to Backtracking | Set 1Backtracking | Set 1the first move that is still left (All possible moves are added to Sone 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.

Basic terminologies:

- **Solution vector:**

The desired solution X to a problem instance P of input size n is as a vector of candidate solutions that are selected from some finite set of possible solutions S.

Thus, a solution can be represented as an n-tuple (X1, X2, ..., Xn) and its partial solution is given as (X1, X2,..., Xi) where i<n.

E.g. for a 4-queens problem X = {2,4,1,3) is a solution vector.

- **Constraints:**

Constraints are the rules to confine the solution vector (X1, X2..... Xa).

They determine the values of candidate solutions and their relationship with each other. There are two types of constraints:

(1) implicit constraints

and

(2) explicit constraints.

1. ***Implicit constraints:***

These are the rules that identify the tuples in the solution space S that satisfy the specified criterion function of a problem instance P. They give the directives of relating all

E.g., in the case of the N-queens problem, all xi's must be distinct satisfying the criterion function of non- attacking queens, in the case of the 0/1 knapsack problem all x's with value 'I' must represent the item giving overall maximum profit and having total weight S knapsack capacity.

2.Explicit constraints:

These are the rules by which all candidate solutions x_{i} ^ prime s are restricted to take on values only from a specified set in a problem instance P. Explicit constraints vary with the instances of the problem.

Eg., in case of the N-queens problem, if N = 4 then x i in \{1, 2, 3, 4\} and if then x i in[1, 2, 3, 4, 5, 6 N = 8 7,8); in case of 0/1 knapsack problem xe \{0, 1\} where x_{i} = 0 represents the exclusion of an item i and x_{i} = 1 represents the inclusion of an item i.

- **Solution space:**

All candidate solutions xi's satisfying the explicit constraints form the solution space S of a problem instance P. In a state space tree, all paths from the root node to a leaf node describe the solution space.

Eg of in the case of N-queens problem, all n! orderings (x_{1}, x_{2} ,...,x n) form the solution space of that problem instance.

- **State space tree:**

A representation of the solution space S of a problem instance P in the form of a tree is defined as the state space tree.

It facilitates systematic search in the solution space to determine the desired solution to a problem.

A solution space of a given problem can be represented by different state space trees.

- **State space:**

The state space of a problem is described by all paths from a root node to other nodes in a state space tree.

- **Problem state:**

Each node in a state space tree describes a problem state or a partial solution formed by making choices from the root of the tree to that node.

These are the problem states producing a tuple in the solution space S. At every internal node, the solution states are partitioned into disjoint sub-solution spaces. In a state space tree for a variable tuple size, all nodes are solution states.

In a state space tree for a fixed tuple size, only the leaf nodes are solution states.

- **Answer states:**

These are the solution states that satisfy the implicit constraints.

These states thus describe the desired solution-tuple (or answer-tuple).

- **Promising node:**

A node is promising if it eventually leads to the desired solution.

A promising node corresponds to a partial solution that is still feasible.

Any time the partial node becomes infeasible, that branch will no longer be pursued.

- **Non-promising node:**

A node is non-promising if it eventually leads to a state that cannot produce the desired solution. A non-promising node corresponds to a partial solution that shows infeasibility to get a complete solution.

Such nodes are killed by a bounding function without further exploration.

- **Live node:**

It is a node that has been generated and all of whose children are not yet been generated.

- **E-node:**

It is a live node whose children are currently being generated.

- **Dead node:**

A node that is either not to be expanded further or for which all its children have been generated is known as a dead node.

- **Depth first node generation:**

Here, the latest live node becomes the next E-node.The moment a new child of the current E-node is generated, that child will be the new E-node. In backtracking, a state space tree is constructed by using the depth first node generation approach

It is also known as a “validity function”, or “criterion function”, or “promising function”.

It is an optimization function B(x1, x2. Xa) which is to be either maximized or minimized for a given problem instance P.

It optimizes the search of a solution vector (X1, X2,... Xn) in the solution space S of a problem instance P.

It helps to reject the candidate solutions not leading to the desired solution to the problem. Thus, it kills the live nodes without exploring their children if constraints are not satisfied.

Eg, in the case of the knapsack problem, the criterion function is the maximization of the profit by filling a knapsack.

- **Static trees:**

These are the state space trees whose tree formulation is independent of the problem instance being solved.

- **Dynamic trees:**

These are the state space trees whose tree formulation varies with the problem instance being solved.

Difference between Recursion and Backtracking:
In recursion, the function calls itself until it reaches a base case. In backtracking, we use recursion to explore all the possibilities until we get the best result for the problem.

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);
```

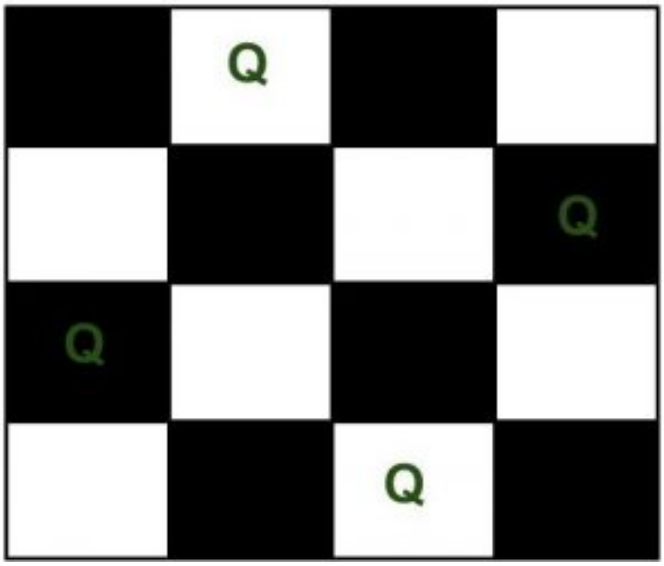
```
            findSolutions(n+1, other params);
            removeValue(val, n);
```

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}
```

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.

Algorithms of Backtracking:

- **Generate k-ary Strings**
- **Graph Coloring Problem**
- **Hamiltonian Cycles**
- **N-Queens Problem**
- **Knapsack Problem**

```
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.
4) 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

Last Updated : 15 Mar, 2023

134

1. Selection Sort Algorithm – Data Structure and Algorithm Tutorials
2. Introduction to Sorting Techniques – Data Structure and Algorithm Tutorials
3. Introduction to Min-Heap – Data Structure and Algorithm Tutorials
4. Binary Search - Data Structure and Algorithm Tutorials
5. Insertion Sort - Data Structure and Algorithm Tutorials
6. Introduction to Disjoint Set Data Structure or Union-Find Algorithm
7. Top 20 Backtracking Algorithm Interview Questions
8. Static Data Structure vs Dynamic Data Structure
9. What is the difference between Backtracking and Recursion?
10. Difference between Backtracking and Branch-N-Bound technique

Related Tutorials

1. Learn Data Structures with Javascript | DSA Tutorial
2. Introduction to Max-Heap – Data Structure and Algorithm Tutorials
3. Introduction to Set – Data Structure and Algorithm Tutorials
4. Introduction to Map – Data Structure and Algorithm Tutorials
5. What is Dijkstra’s Algorithm? | Introduction to Dijkstra's Shortest Path Algorithm

Next

Difference between Backtracking and Branch-N-Bound technique

GeeksforGeeks

Vote for difficulty

Current difficulty : Medium

Easy | Normal | Medium | Hard | Expert

Improved By : atulim, tripathipriyanshu1998, sureshpradhana19, shreyasnaphad, twel12, krisania804, laxmishinde5t82

Article Tags : Algorithms-Backtracking, Tutorials, Backtracking, Branch and Bound, DSA, Recursion

Practice Tags : Backtracking, Recursion

Improve Article | Report Issue

GeeksforGeeks
A-143, 9th Floor, Sovereign Corporate Tower, Sector-136, Noida, Uttar Pradesh - 201305

feedback@geeksforgeeks.org

C++
GoLang
SQL
R Language
Android Tutorial

Linked List
Stack
Queue
Tree
Graph

Algorithms

Sorting
Searching
Greedy
Dynamic Programming
Pattern Searching
Recursion
Backtracking

Web Development

HTML
CSS
JavaScript
Bootstrap
ReactJS
AngularJS
NodeJS

Data Science & ML

Data Science With Python
Data Science For Beginner
Machine Learning Tutorial
Maths For Machine Learning
Pandas Tutorial
NumPy Tutorial
NLP Tutorial

Interview Corner

Company Preparation
Preparation for SDE
Company Interview Corner
Experienced Interview
Internship Interview
Competitive Programming
Aptitude

Python

Python Tutorial
Python Programming Examples
Django Tutorial
Python Projects
Python Tkinter
OpenCV Python Tutorial

GfG School

CBSE Notes for Class 8
CBSE Notes for Class 9
CBSE Notes for Class 10
CBSE Notes for Class 11
CBSE Notes for Class 12
English Grammar

UPSC/SSC/BANKING

SSC CGL Syllabus
SBI PO Syllabus
IBPS PO Syllabus

Write & Earn

Write an Article
Improve an Article
Pick Topics to Write