Lecture 23: Greedy Algorithms

Date: 11/06/2023

An algorithm is greedy when the path picked is regarded as the best option based on a specific criterion without considering future consequences. But it typically evaluates feasibility before making a final decision.
⇒ The correctness of the solution depends on the problem and criteria used.

Examples,

- Minimum Spanning Tree using Prim’s Algorithm
- Shortest Path using Dijkstra’s Algorithm

Advantages

⇒ Greedy algorithms are quite straightforward to implement and easy to understand.
⇒ They most of the time have lower time complexities
⇒ They're useful in solving optimization problems, returning a maximum or minimum value.

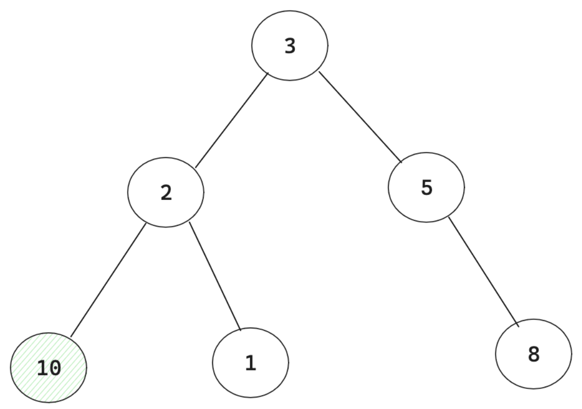
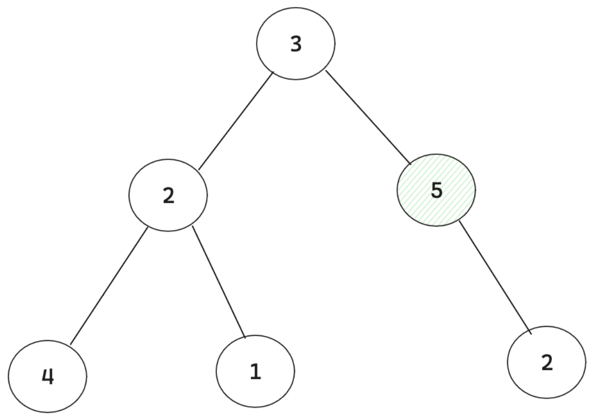
Disadvantages

⇒ Sometimes, they don’t offer the best solution.
⇒ They don’t check the correctness of the result produced.

Problems

Find Max Value in Tree

Given a root of a tree, find the node with maximum value.

Example 1:	Example 2:
	

Brute Force Approach

⇒ we can try all options and then decide the minimum of those. Which guarantees us that the answer will be correct.

```
function findMax(Node root){
    vector<Node> nodes = dfs(root)

    int answer = INT_MIN
    for node in nodes{
        answer = max(answer, node->val)
    }

    return answer
}
```

Time Complexity

⇒ We first need to get all the nodes, using the dfs or bfs algorithm. Which gives us O(V+E) time complexity. After that, we need to traverse all the nodes in a for loop. Which will be θ(V)
⇒ Total Time Complexity: O(V+E)

Greedy Approach

⇒ The input tree can be anything, so we can't make the brute force algorithm efficient by reducing the search space.
⇒ So, let's try to convert the problem so that we can make some assumptions and remove some of the options.

⇒ If we convert a regular tree to a binary search tree (BST), they have a property that all the nodes in the left subtree will always be less than the nodes in the right subtree.
⇒ We can use this information to always pick the right subtree path. Now, this algorithm also gives us the maximum node of the tree.

```
function findMax(Node root){
    root = convertToBST(root)

    node = root;
    while node != null {
        answer = node->right->val;
        node = node->right;
    }

    return answer;
}
```

Time Complexity
⇒ Converting a tree to BST is not an easy task and it takes more than O(V+E) time.
⇒ So, even after converting the tree, finding the max node is very efficient.
Preprocessing part is our bottleneck.

⇒ For this problem, brute force solution will work just fine.

Schedule Meetings

Given list of meetings, with start time and end time. Schedule them in such a way that the person can attend maximum meetings.

Input:	Output:
[	[
[1, 10],	[1, 3],
[5, 6],	[5, 6],
[2, 4],	[8, 10]
[1, 3],	]
[8, 10]	
]	

Brute Force Approach

⇒ To try all the options for this problem, we need to write a backtracking algorithm. Which can find all possible combinations of meetings. And for each one we can check which ones are valid (not clashing) and then select the maximum size one.

```
function scheduleMeetings(listOfMeetings){
    all_combinations = get_combinations(listOfMeetings)

    answer = []
    for combination in all_combination{
        if meetings_clash(combination)
        continue

        if len(combination) > len(answer)
        answer = combination
}
```

Time Complexity

⇒ We are finding all possible combinations and then for each combination we need to check if any meeting is clashing or not.

⇒ The number of combinations possible for an array of size n is 2^n.

⇒ Time complexity: $\Omega(2^n)$

Greedy Approach

⇒ Again, as input can be anything and it does not follow any property, so we can't make any assumptions and reduce the search space.

⇒ Sort the list of meetings based on the **ending time**. Now, because of that, we know that once we schedule the meeting i, the best next meeting to schedule should be i+1. But, we may also have clashes. So we need to check before scheduling the meeting i+1.
⇒ Greediness in this algorithm is that once we know we can schedule the meeting i without any clash, we schedule it, without worrying about future meetings.

```
function scheduleMeetings(listOfMeetings){
    sort listOfMeetings based on second element

    scheduled = []
    for meeting in listOfMeetings{
        if does_clash(scheduled[-1], meeting)
        continue

        scheduled.append(meeting)
    }
}
```

⇒ Note that we only check clashes with the last meeting scheduled. Because, as the input is sorted now, if last meeting does not clash then none of the previous meeting will clas