Previous (/articles/word-pattern/) Next (/articles/add-and-search-word/)

797. All Paths From Source To Target (/problems/all-paths-from-source-to-target/)

July 2, 2020 | 5.5K views

Average Rating: 5 (10 votes)

Given a directed, acyclic graph of `N` nodes. Find all possible paths from node `0` to node `N-1`, and return them in any order.

The graph is given as follows: the nodes are 0, 1, ..., graph.length - 1. graph[i] is a list of all nodes j for which the edge (i, j) exists.

```
Example:
Input: [[1,2], [3], [3], []]
Output: [[0,1,3],[0,2,3]]
Explanation: The graph looks like this:
0--->1
|    |
v    v
2--->3
There are two paths: 0 -> 1 -> 3 and 0 -> 2 -> 3.
```

Note:

- The number of nodes in the graph will be in the range `[2, 15]`.
- You can print different paths in any order, but you should keep the order of nodes inside one path.

Solution

Approach 1: Backtracking

Overview

If a hint is ever given on the problem description, that would be ***backtracking***.

Indeed, since the problem concerns about the *path exploration* in a *graph* data structure, it is a perfect scenario to apply the backtracking algorithm.

> As a reminder, backtracking (https://en.wikipedia.org/wiki/Backtracking) is a general algorithm that incrementally builds candidates to the solutions, and abandons a candidate ("*backtrack*") as soon as it determines that the candidate cannot possibly lead to a valid solution.

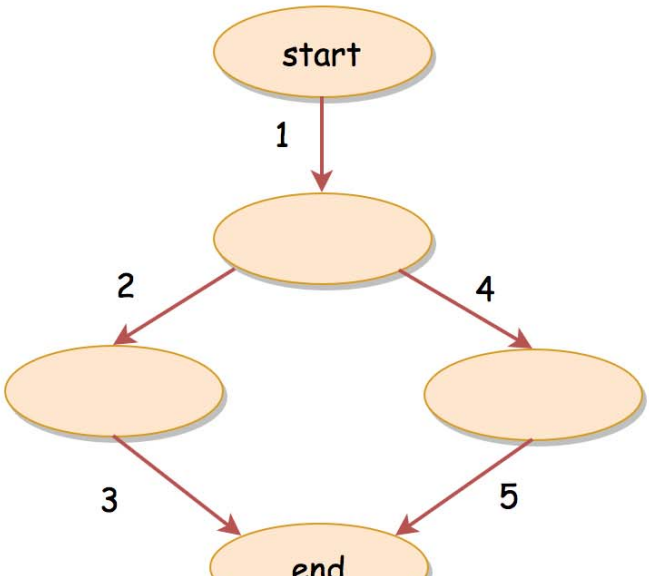
For more details about how to implement a backtracking algorithm, one can refer to our Explore card (https://leetcode.com/explore/learn/card/recursion-ii/472/backtracking/).

Intuition

> Specifically, for this problem, we could assume ourselves as an agent in a game, we can explore the graph one step at a time.

At any given node, we try out each neighbor node *recursively* until we reach the target or there is no more node to hop on. By trying out, we mark the choice before moving on, and later on we reverse the choice (*i.e.* backtrack) and start another exploration.

To better demonstrate the above idea, we illustrate how an agent would explore the graph with the *backtracking* strategy, in the following image where we mark the order that each edge is visited.



Algorithm

The above idea might remind one of the Depth-First Search (**DFS**) traversal algorithm. Indeed, often the backtracking algorithm assumes the form of DFS, but with the additional step of *backtracking*.

And for the DFS traversal, we often adopt the **recursion** as its main form of implementation. With recursion, we could implement a backtracking algorithm in a rather intuitive and concise way. We break it down into the following steps:

- Essentially, we want to implement a recursive function called `backtrack(currNode, path)` which continues the exploration, given the current node and the path traversed so far.
    - Within the recursive function, we first define its base case, *i.e.* the moment we should terminate the recursion. Obviously, we should stop the exploration when we encounter our target node. So the condition of the base case is `currNode == target`.
    - As the body of our recursive function, we should enumerate through all the neighbor nodes of the current node.
    - For each iteration, we first mark the choice by appending the neighbor node to the path. Then we *recursively* invoke our `backtrack()` function to explore *deeper*. At the end of the iteration, we should reverse the choice by popping out the neighbor node from the path, so that we could start all over for the next neighbor node.
- Once we define our `backtrack()` function, it suffices to add the initial node (*i.e.* node with index `0`) to the path, to *kick off* our backtracking exploration.

Java Python3 Articles > 797. All Paths From Source To Tar Copy

```python
class Solution:
    def allPathsSourceTarget(self, graph: List[List[int]]) -> List[List[int]]:

        target = len(graph) - 1
        results = []

        def backtrack(currNode, path):
            # if we reach the target, no need to explore further.
            if currNode == target:
                results.append(list(path))
                return
            # explore the neighbor nodes one after another.
            for nextNode in graph[currNode]:
                path.append(nextNode)
                backtrack(nextNode, path)
                path.pop()
        # kick of the backtracking, starting from the source node (0).
        path = deque([0])
        backtrack(0, path)

        return results
```

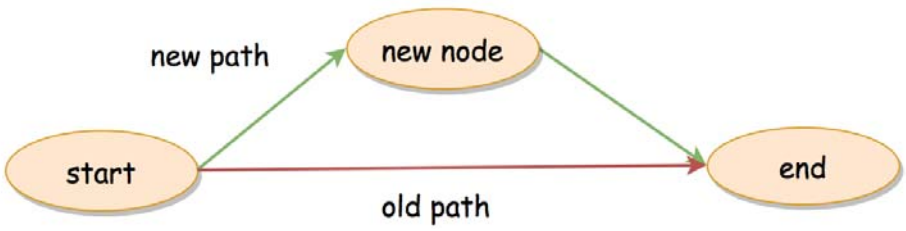
Complexity Analysis

Let N be the number of nodes in the graph.

First of all, let us estimate how many paths there are at maximum to travel from the Node `0` to the Node `N-1` for a graph with N nodes.

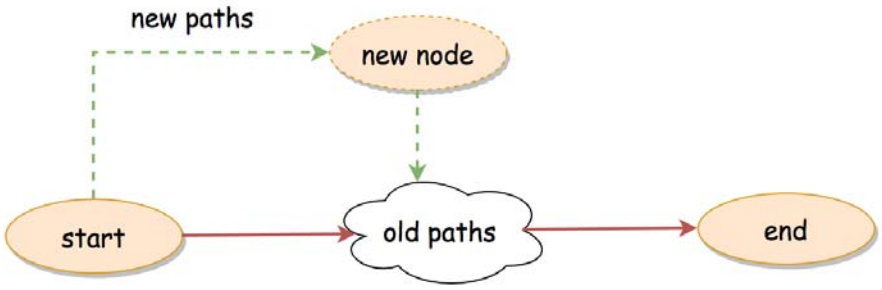
Let us start from a graph with only two nodes. As one can imagine, there is only one single path to connect the only two nodes in the graph.

Now, let us add a new node into the previous two-nodes graph, we now have two paths, one from the previous path, the other one is bridged by the newly-added node.



> If we continue to add nodes to the graph, one insight is that every time we add a new node into the graph, the number of paths would **double**.

With the newly-added node, new paths could be created by preceding all previous paths with the newly-added node, as illustrated in the following graph.



As a result, for a graph with N nodes, at maximum, there could be $\sum_{i=0}^{N-2} 2^i = 2^{N-1} - 1$ number of paths between the starting and the ending nodes.

- Time Complexity: $\mathcal{O}(2^N \cdot N)$
  - As we calculate shortly before, there could be at most $2^{N-1} - 1$ possible paths in the graph.
  - For each path, there could be at most $N - 2$ intermediate nodes, *i.e.* it takes $\mathcal{O}(N)$ time to build a path.
  - To sum up, a **loose** upper-bound on the time complexity of the algorithm would be $(2^{N-1} - 1) \cdot \mathcal{O}(N) = \mathcal{O}(2^N \cdot N)$, where we consider it takes $\mathcal{O}(N)$ efforts to build each path.
  - It is a loose uppper bound, since we could have overlapping among the paths, therefore the efforts to build certain paths could benefit others.
- Space Complexity: $\mathcal{O}(2^N \cdot N)$
  - Similarly, since at most we could have $2^{N-1} - 1$ paths as the results and each path can contain up to N nodes, the space we need to store the results would be $\mathcal{O}(2^N \cdot N)$.
  - Since we also applied *recursion* in the algorithm, the recursion could incur additional memory consumption in the function call stack. The stack can grow up to N consecutive calls. Meanwhile, along with the recursive call, we also keep the state of the current path, which could take another $\mathcal{O}(N)$ space. Therefore, in total, the recursion would require additional $\mathcal{O}(N)$ space.

- To sum up, the space complexity of the algorithm is $\mathcal{O}(2^N \cdot N) + \mathcal{O}(N) = \mathcal{O}(2^N \cdot N)$.

Approach 2: Top-Down Dynamic Programming

Intuition

The backtracking approach applies the paradigm of divide-and-conquer, which breaks the problem down to smaller steps. As one knows, there is another algorithm called ***Dynamic Programming*** (DP), which also embodies the idea of divide-and-conquer.

As it turns out, we could also apply the DP algorithm to this problem, although it is less optimal than the backtracking approach as one will see later.

> More specifically, we adopt the **Top-Down** DP approach, where we take a *laissez-faire* strategy assuming that the target function would work out on its own.

Given a node `currNode`, our target function is `allPathsToTarget(currNode)`, which returns all the paths from the current node to the target node.

The target function could be calculated by iterating through the neighbor nodes of the current node, which we summarize with the following *recursive* formula:

$$\forall \text{nextNode} \in \text{neighbors}(\text{currNode}),$$
$$\text{allPathsToTarget}(\text{currNode}) = \{\text{currNode} + \text{allPathsToTarget}(\text{nextNode})\}$$

The above formula can be read intuitively as: "*the paths from the current node to the target node consist of all the paths starting from each neighbor of the current node.*"

Algorithm

Based on the above formula, we could implement a DP algorithm.

- First of all, we define our target function `allPathsToTarget(node)`.
    - Naturally our target function is a recursive function, whose base case is when the given `node` is the target node.
    - Otherwise, we iterate through its neighbor nodes, and we invoke our target function with each neighbor node, *i.e.* `allPathsToTarget(neighbor)`
    - With the returned results from the target function, we then prepend the current node to

the downstream paths, in order to build the final paths.

- With the above defined target function, we simply invoke it with the desired starting node, *i.e.* node `0`.

Note that, there is an important detail that we left out in the above step. In order for the algorithm to be fully-qualified as a DP algorithm, we should **reuse** the intermediate results, rather than re-calculating them at each occasion.

Specially, we should **cache** the results returned from the target function `allPathsToTarget(node)`, since we would encounter a node multiple times if there is an overlapping between paths. Therefore, once we know the paths from a given node to the target node, we should keep it in the cache for reuse. This technique is also known as ***memoization***.

Java | Python3 Copy

```
class Solution:
    def allPathsSourceTarget(self, graph: List[List[int]]) -> List[List[int]]:

        target = len(graph) - 1

        # apply the memoization
        @lru_cache(maxsize=None)
        def allPathsToTarget(currNode):
            if currNode == target:
                return [[target]]

            results = []
            for nextNode in graph[currNode]:
                for path in allPathsToTarget(nextNode):
                    results.append([currNode] + path)

            return results

        return allPathsToTarget(0)
```

Complexity Analysis

Let N be the number of nodes in the graph. As we estimated before, there could be at most $2^{N-1} - 1$ number of paths.

- Time Complexity: $\mathcal{O}(2^N \cdot N)$.
    - To estimate the overall time complexity, let us start from the last step when we prepend

the starting node to each of the paths returned from the target function. Since we have to copy each path in order to create new paths, it would take up to N steps for each final path. Therefore, for this last step, it could take us $\mathcal{O}(2^{N-1} \cdot N)$ time.

- Right before the last step, when the maximal length of the path is $N-1$, we should have 2^{N-2} number of paths at this moment.
- Deducing from the above two steps, again a **loose** upper-bound of the time complexity would be $\mathcal{O}(\sum_{i=1}^{N} 2^{i-1} \cdot i) = \mathcal{O}(2^N \cdot N)$
- The two approach might have the same asymptotic time complexity. However, in practice the DP approach is slower than the backtracking approach, since we copy the intermediate paths over and over.
- Note that, the performance would be degraded further, if we did not adopt the memoization technique here.

- Space Complexity: $\mathcal{O}(2^N \cdot N)$
  - Similarly, since at most we could have $2^{N-1} - 1$ paths as the results and each path can contain up to N nodes, the space we need to store the results would be $\mathcal{O}(2^N \cdot N)$.
  - Since we also applied *recursion* in the algorithm, it could incur additional memory consumption in the function call stack. The stack can grow up to N consecutive calls. Therefore, the recursion would require additional $\mathcal{O}(N)$ space.
  - To sum up, the space complexity of the algorithm is $\mathcal{O}(2^N \cdot N) + \mathcal{O}(N) = \mathcal{O}(2^N \cdot N)$.

Rate this article:

Previous (/articles/word-pattern/) Next (/articles/add-and-search-word/)

Comments: 13 Sort By

Type comment here... (Markdown is supported)

Preview Post


(/gdkou90)

gdkou90 (/gdkou90) ★ 95 July 13, 2020 10:54 AM

LeetCode should provide more solutions like this so that the subscribed users would feel it worthy.

21 Share Reply

SHOW 1 REPLY


(/neosdeus)

NeosDeus (/neosdeus) ★ 300 July 20, 2020 11:54 PM

These explanations saved me hours from digging through the discussion section and trying to find the most well-explained solution. Honestly, if I had these last year I wouldn't be jobless right now.

4 Share Reply


(/jpf1983)

jpf1983 (/jpf1983) ★ 9 July 15, 2020 2:06 AM

thanks for the time complexity analysis

4 Share Reply


(/cranky_coder)

cranky_coder (/cranky_coder) ★ 1 5 hours ago

In solution 1: While calculating the total number of possible paths between origin and destination, why are we doing summation?
If we have 2 nodes, then total paths are 2^0 (i=0)
If we have 3 nodes, then total paths are 2^1 (i=1) {not 2^0 + 2^1}
If we have 4 nodes, then total paths are 2^2 (i=2) {not 2^0 + 2^1 + 2^2}

Read More

2 Share Reply


(/v82_ranjan)

v82_ranjan (/v82_ranjan) ★ 2 July 15, 2020 12:03 AM

Nice Explanation!!!!

2 Share Reply


(/vasuji)

Vasuji (/vasuji) ★ 2 July 4, 2020 9:42 PM

Outstanding and clear!

2 Share Reply


(/move78)

Move78 (/move78) ★ 2 July 3, 2020 1:24 PM

good explanation!

2 Share Reply


(/aneeshak)

aneeshak (/aneeshak) ★ 4 10 hours ago

There is a typo in the first solution
"bactracking" instead of "backtracking" in the line "As a reminder, ... "

1 Share Reply

SHOW 1 REPLY

(/zhdv)

zhdv (/zhdv) ★ 4 12 hours ago

Hi, @liaison (https://leetcode.com/liaison) ,

Here is a minor misprint in space complexity, approach 2. In summary it is incorrectly stated as O(N).

A good article, though!

1 Share Reply

SHOW 1 REPLY

(/rajatag03)

rajatag03 (/rajatag03) ★ 39 12 hours ago

Is it a typo mistake? the space complexity should be written (2N.N) why N in 2nd Approach ??? Can anybody confirm?

1 Share Reply

SHOW 2 REPLIES

‹ 1 2 ›