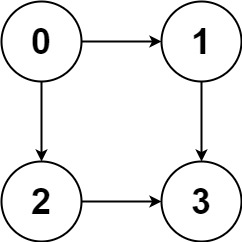
<https://leetcode.com/problems/all-paths-from-source-to-target/description/>

Given a directed acyclic graph (DAG) of n nodes labeled from 0 to n - 1, find all possible paths from node 0 to node n - 1 and return them in any order.

The graph is given as follows: graph[i] is a list of all nodes you can visit from node i (i.e., there is a directed edge from node i to node graph[i][j]).

**Example 1:**

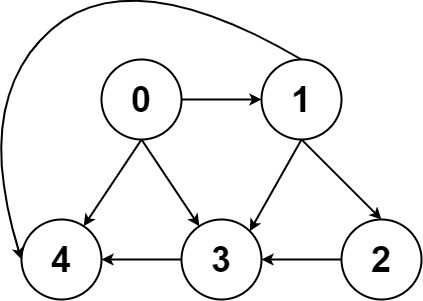


**Input:** graph = [[1,2],[3],[3],[]]

**Output:** [[0,1,3],[0,2,3]]

**Explanation:** There are two paths: 0 -> 1 -> 3 and 0 -> 2 -> 3.

**Example 2:**



**Input:** graph = [[4,3,1],[3,2,4],[3],[4],[]]

**Output:** [[0,4],[0,3,4],[0,1,3,4],[0,1,2,3,4],[0,1,4]]

**Constraints:**

n == graph.length

2 <= n <= 15

0 <= graph[i][j] < n

graph[i][j] != i (i.e., there will be no self-loops).

All the elements of graph[i] are **unique**.

The input graph is **guaranteed** to be a **DAG**.

**Attempt 1: 2025-02-08**

**Solution 1: DFS + Backtracking (30 min)**

class Solution {

    public List<List<Integer>> allPathsSourceTarget(int[][] graph) {

        List<List<Integer>> result = new ArrayList<>();

        List<Integer> path = new ArrayList<>();

        // Start from node 0

        path.add(0);

        helper(graph, result, path, 0);

        return result;

    }

    private void helper(int[][] graph, List<List<Integer>> result, List<Integer> path, int node) {

        // If the current node is the target (n-1), add the current path to the result

        if(node == graph.length - 1) {

            // Add a copy of the current path

            result.add(new ArrayList<>(path));

            return;

        }

        // Explore all neighbors of the current node

        for(int neighbor : graph[node]) {

            // Add the neighbor to the current path

            path.add(neighbor);

            // Recursively explore

            helper(graph, result, path, neighbor);

            // Backtrack: remove the neighbor from the path

            path.remove(path.size() - 1);

        }

    }

}

Time Complexity:

In the worst case, every node is connected to every other node, resulting in (O(2^n)) paths.

For each path, we spend (O(n)) time to copy it to the result.

Overall time complexity: (O(2^n \* n)).

Space Complexity:

The recursion stack can go up to (O(n)) in depth.

The space required to store all paths is (O(2^n \* n)).

Overall space complexity: (O(2^n \* n)).

**Refer to Deepseek**

LeetCode problem 797. All Paths From Source to Target is a classic graph problem where you are given a directed acyclic graph (DAG) and asked to find all possible paths from the source node (node 0) to the target node (node n-1). This problem can be solved using Depth-First Search (DFS) or Backtracking.

**Problem Summary:**

You are given a directed acyclic graph (DAG) represented as an adjacency list.

The graph has n nodes labeled from 0 to n - 1.

You need to find all possible paths from node 0 (source) to node n - 1 (target).

Return the paths as a list of lists, where each inner list represents a path.

**Key Observations:**

**Directed Acyclic Graph (DAG):**

**Since the graph is a DAG, there are no cycles, and you don't need to worry about revisiting nodes.**

All Paths:

You need to explore all possible paths from the source to the target, not just the shortest or longest path.

**Backtracking:**

**Backtracking is a natural fit for this problem because you need to explore all possible paths and backtrack when you reach the target or a dead end.**

**Approach: DFS with Backtracking**

**Steps:**

Start from the Source:

Begin DFS from node 0 (the source).

Explore Neighbors:

For each neighbor of the current node, recursively explore all paths.

Base Case:

If the current node is the target (n - 1), add the current path to the result.

**Backtracking:**

**After exploring a path, remove the current node from the path to allow other paths to be explored.**

Result:

Collect all valid paths in a list and return it.

**Solution Code (DFS with Backtracking):**

import java.util.\*;

class Solution {

public List<List<Integer>> allPathsSourceTarget(int[][] graph) {

List<List<Integer>> result = new ArrayList<>();

List<Integer> path = new ArrayList<>();

path.add(0); // Start from node 0

dfs(graph, 0, path, result);

return result;

}

private void dfs(int[][] graph, int node, List<Integer> path, List<List<Integer>> result) {

// If the current node is the target (n-1), add the current path to the result

if (node == graph.length - 1) {

result.add(new ArrayList<>(path)); // Add a copy of the current path

return;

}

// Explore all neighbors of the current node

for (int neighbor : graph[node]) {

path.add(neighbor); // Add the neighbor to the current path

dfs(graph, neighbor, path, result); // Recursively explore

path.remove(path.size() - 1); // Backtrack: remove the neighbor from the path

}

}

}

**Explanation of the Code:**

Initialization:

result stores all valid paths.

path stores the current path being explored.

DFS Function:

The dfs function explores all paths starting from the current node.

If the current node is the target (graph.length - 1), the current path is added to the result.

**Backtracking:**

**After exploring a neighbor, the last node is removed from the path to allow other paths to be explored.**

Result:

The result list contains all valid paths from the source to the target.

**Complexity Analysis:**

Time Complexity:

In the worst case, every node is connected to every other node, resulting in (O(2^n)) paths.

For each path, we spend (O(n)) time to copy it to the result.

Overall time complexity: (O(2^n \* n)).

Space Complexity:

The recursion stack can go up to (O(n)) in depth.

The space required to store all paths is (O(2^n \* n)).

Overall space complexity: (O(2^n \* n)).

**Example Walkthrough:**

**Input:**

graph = [[1, 2], [3], [3], []]

**Graph Representation:**

0 -> 1

0 -> 2

1 -> 3

2 -> 3

3 -> []

**Execution:**

Start at node 0. Path: [0].

Explore neighbor 1:

Path: [0, 1].

Explore neighbor 3:

Path: [0, 1, 3] (target reached). Add to result.

Backtrack to node 0. Path: [0].

Explore neighbor 2:

Path: [0, 2].

Explore neighbor 3:

Path: [0, 2, 3] (target reached). Add to result.

**Output:**

[[0, 1, 3], [0, 2, 3]]

**Alternative Approach: BFS (Iterative)**

If you prefer an iterative approach, you can use BFS with a queue to explore all paths. However, BFS is less intuitive for this problem because it requires storing paths explicitly, which can be memory-intensive.

**Key Takeaways:**

DFS with Backtracking is the most natural and efficient approach for this problem.

The problem is a good example of how to use backtracking to explore all possible paths in a graph.

The solution is concise and leverages the properties of a DAG (no cycles).

**You mention about backtracking, what's the relationship between DFS and backtracking ? or let's say what's the difference between using DFS with backtracking and using DFS without backtracking ? when must we use DFS with backtracking ?**

Great question! The relationship between **DFS (Depth-First Search)** and **Backtracking** is often misunderstood, but they are closely related concepts. Let’s break it down:

**DFS vs. Backtracking**

**1. DFS (Depth-First Search):**

DFS is a graph traversal algorithm that explores as far as possible along each branch before backtracking.

It is used to traverse or search through a graph or tree structure.

**In DFS, you visit a node, then recursively visit its unvisited neighbors until you reach a dead end. Then, you backtrack to explore other branches.**

Key Characteristics:

Explores one path at a time.

Uses a stack (implicitly via recursion or explicitly with a data structure).

**Does not necessarily involve "undoing" actions (backtracking).**

**2. Backtracking:**

**Backtracking is a general algorithmic technique for solving problems by exploring all possible solutions incrementally and abandoning a solution ("backtracking") as soon as it determines that the solution cannot be completed.**

**It is often implemented using DFS because DFS naturally explores one path at a time and can easily "undo" decisions when it backtracks.**

Key Characteristics:

Explores all possible solutions.

Involves making a choice, exploring further, and then undoing the choice if it doesn’t lead to a solution.

Often used in problems like permutations, combinations, and pathfinding.

**Relationship Between DFS and Backtracking:**

**Backtracking is a refinement of DFS:**

**Backtracking uses DFS to explore all possible solutions, but it adds the ability to "undo" decisions when a path doesn’t lead to a valid solution.**

**In other words, backtracking is DFS with pruning (eliminating invalid paths early).**

DFS without backtracking:

In standard DFS, you don’t explicitly "undo" decisions. You simply explore all reachable nodes from the current node.

For example, in a tree traversal, you don’t need to backtrack because you don’t revisit nodes.

DFS with backtracking:

In backtracking, you explicitly "undo" decisions (e.g., remove the last element from a path) after exploring a branch.

This is necessary when you need to explore multiple paths that share common nodes (e.g., finding all paths in a graph).

**When to Use DFS with Backtracking:**

You must use DFS with backtracking in the following scenarios:

Finding All Possible Solutions:

When the problem requires exploring all possible solutions (e.g., all paths, all permutations, all combinations).

Example: LeetCode 797 (All Paths From Source to Target), LeetCode 46 (Permutations).

Pruning Invalid Paths Early:

When you can determine that a partial solution cannot lead to a valid solution, so you abandon it early.

Example: LeetCode 39 (Combination Sum), LeetCode 51 (N-Queens).

Exploring Multiple Paths That Share Nodes:

When multiple paths can share the same nodes, and you need to explore all of them.

Example: Finding all paths in a graph.

Problems with Constraints:

When the problem has constraints that require you to explore all valid configurations.

Example: Sudoku solver, subset problems.

**When to Use DFS Without Backtracking:**

You can use DFS without backtracking in the following scenarios:

Traversing a Tree or Graph:

When you only need to visit each node once (e.g., tree traversal, finding connected components).

Example: LeetCode 104 (Maximum Depth of Binary Tree), LeetCode 200 (Number of Islands).

Finding a Single Solution:

When you only need to find one valid solution (e.g., a path, a configuration).

Example: Finding a path from source to target (if only one path is needed).

No Need to Undo Decisions:

When the problem does not require revisiting nodes or undoing decisions.

Example: Topological sorting, cycle detection.

**Example: DFS with Backtracking vs. DFS Without Backtracking**

**Problem: Find All Paths from Source to Target (LeetCode 797)**

DFS with Backtracking:

You explore all paths from the source to the target.

After exploring a path, you backtrack to explore other paths.

Example:

private void dfs(int[][] graph, int node, List<Integer> path, List<List<Integer>> result) {

if (node == graph.length - 1) {

result.add(new ArrayList<>(path)); // Add a copy of the current path

return;

}

for (int neighbor : graph[node]) {

path.add(neighbor); // Make a choice

dfs(graph, neighbor, path, result); // Explore further

path.remove(path.size() - 1); // Backtrack (undo the choice)

}

}

DFS Without Backtracking:

**You would only find one path from the source to the target.**

**You don’t undo decisions, so you can’t explore multiple paths.**

Example:

private boolean dfs(int[][] graph, int node, List<Integer> path, List<List<Integer>> result) {

if (node == graph.length - 1) {

result.add(new ArrayList<>(path)); // Add a copy of the current path

return true; // Stop after finding one path

}

for (int neighbor : graph[node]) {

path.add(neighbor); // Make a choice

if (dfs(graph, neighbor, path, result)) {

return true; // Stop after finding one path

}

// No backtracking here

}

return false;

}

**Key Differences:**

|  |  |  |
| --- | --- | --- |
| Feature | DFS Without Backtracking | DFS With Backtracking |
| **Purpose** | Traverse or find a single solution | Explore all possible solutions |
| **Undo Decisions** | No | Yes |
| **Use Case** | Tree/graph traversal, single path | All paths, permutations, pruning |
| **Example Problems** | Tree depth, connected components | All paths, N-Queens, Sudoku |

**Summary:**

DFS without backtracking is used for traversal or finding a single solution.

DFS with backtracking is used when you need to explore all possible solutions, undo decisions, or prune invalid paths early.

Use backtracking when the problem requires exploring multiple paths or configurations, especially in combinatorial problems.

**Solution 2: Iterative DFS on DAG (30 min, no Backtracking required)**

class Solution {

    public List<List<Integer>> allPathsSourceTarget(int[][] graph) {

        // To store all valid paths

        List<List<Integer>> result = new ArrayList<>();

        Stack<List<Integer>> stack = new Stack<>();

        // Initialize stack with the starting path [0]

        List<Integer> initialPath = new ArrayList<>();

        initialPath.add(0);

        stack.add(initialPath);

        int target = graph.length - 1;

        // Iterative DFS

        while(!stack.isEmpty()) {

            // Get the current path

            List<Integer> curPath = stack.pop();

            // Get the last node in the path

            int curNode = curPath.get(curPath.size() - 1);

            // If the current node is the target, add the path to the result

            if(curNode == target) {

                // Add a copy of the path

                result.add(curPath);

                continue;

            }

            // Explore all neighbors of the current node

            for(int neighbor : graph[curNode]) {

                // Create a new path copy

                List<Integer> newPath = new ArrayList<>(curPath);

                // Add the neighbor to the new path

                newPath.add(neighbor);

                // Push the new path to the stack

                stack.push(newPath);

            }

        }

        return result;

    }

}

Time Complexity: O(2^n⋅n), where nn is the number of nodes.

This is because there can be up to 2^n paths, and each path can have up to nn nodes.

Space Complexity: O(2^n⋅n), due to the storage of all paths.

**Refer to**

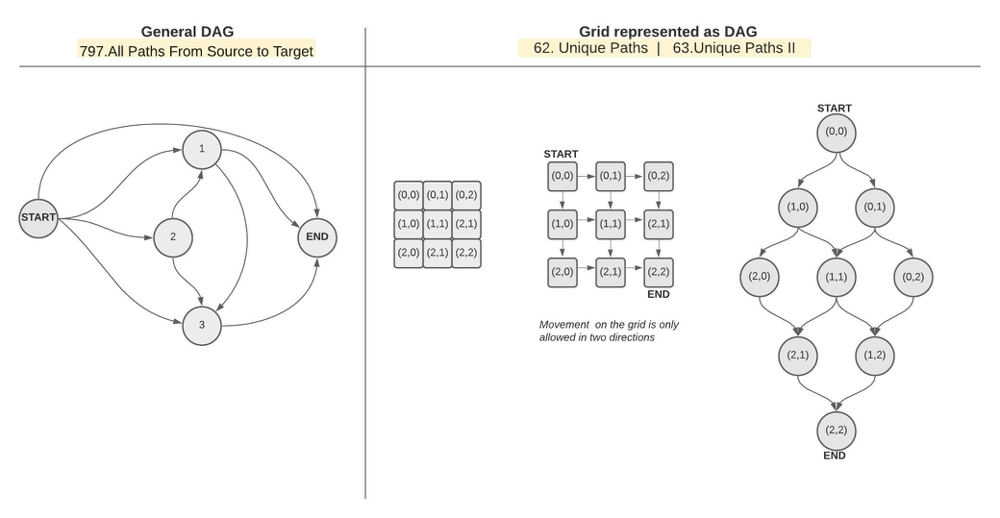
**[Python] Iterative DFS with detailed time complexity & visuals**

<https://leetcode.com/problems/all-paths-from-source-to-target/solutions/986429/python-iterative-dfs-with-detailed-time-complexity-visuals/>

**Iterative DFS Approach - No backtracking**

This approach uses brute-force DFS to generate all possible paths from cell (0,0) to cell (n-1, m-1). The approach in the solution tab talks about backtracking where in fact **backtracking is NOT required at all in this problem** as we need to generate all possible paths. This means all paths are part of the answer. There are no "wrong" paths that we might need to prune or backtrack from.

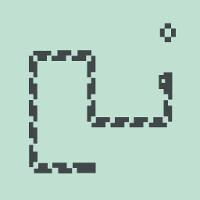
The DFS solution to this problem is very similar to the DFS I used to solve : 63. Unique Paths II, 62. Unique Paths with one exception. The DAG in this problem is a general DAG (Directed Acyclic Graph) whereas the DAG we traverse in problems (62 and 63) is a grid-DAG. That's just my way of saying it's a DAG that represents a grid. See picture below for a visual illustration of the difference between these two.



As you can see, while the grid DAG has a well-defined sturcture, the general-case DAG is unpredictable. This difference though will have no impact on the DFS algorithm and will only materlize in the time complexity of the algorithm. **More on that later.**

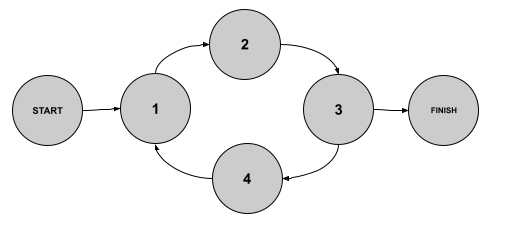
**When do we actually need backtracking ?**

**Backtracking would be required if the input graph was NOT a DAG.** A graph that is acyclic requires backtracking when traversed because our DFS path might circle back and hit itself. Visually, this scenario is very similar to the retro snake game where the snake needs to avoid hiting its growing tail. A snake could hit its own tail only when its movement can form a cycle. (aka. cyclic graph). **Therefore, if the graph is guaranteed not to have cycles, we do not need to bother with backtracking.**

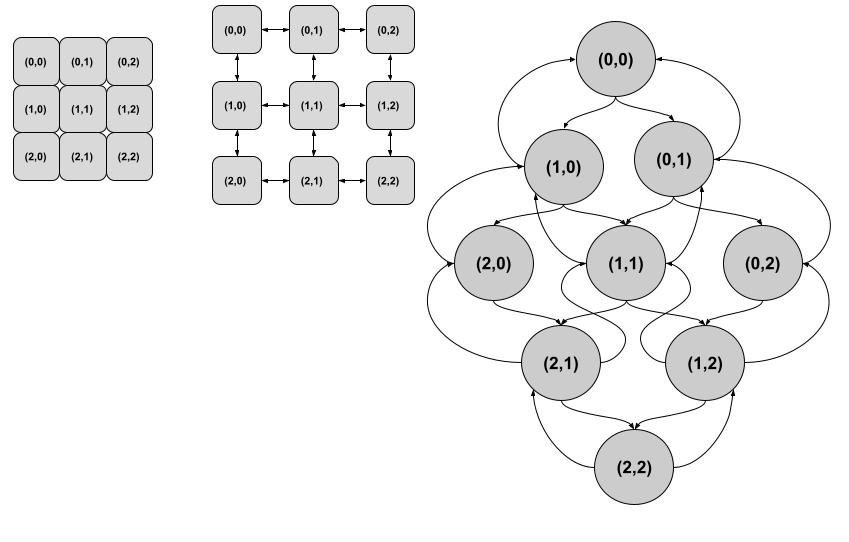


**A cyclic graph can take many forms.**

A general cyclic graph



A grid cyclic graph: i.e. a grid in which movement is allowed in all four directions (left, down, right, up) - 79. Word Search

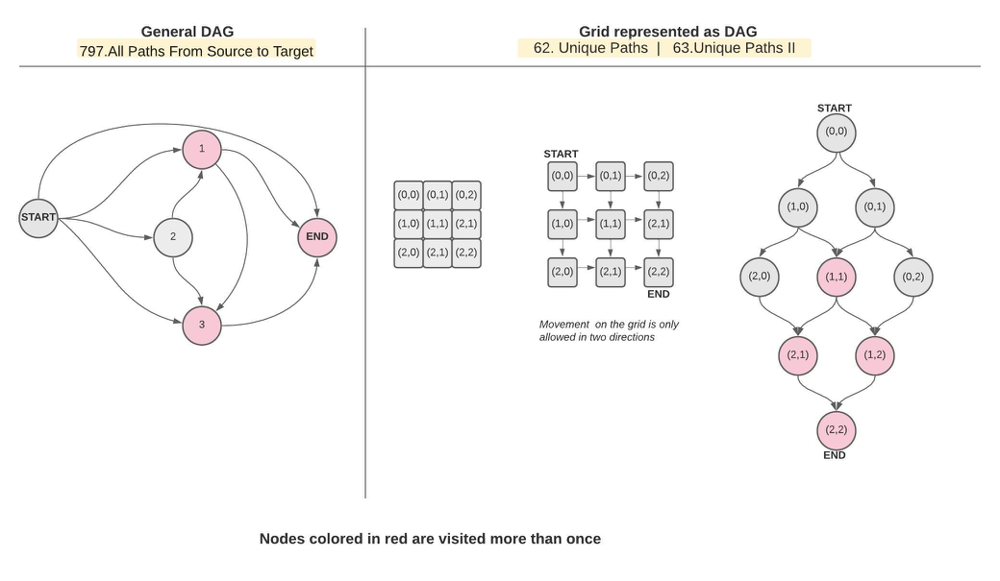


A couple of notes about the algorithm:

**Since the graph is a DAG, we are not worried that we might get stuck in a loop as we traverse it (no cycles/loops), thus the visited set that normally accompanies graph traversal algortihms is not needed.**

**Moreover, a visited set would actualy ruin the algorithm for us and we need to get rid of it, because in this particular problem we need to generate all paths, and for that we might visit some nodes more than once. Those are the nodes that are common among multiple paths. See nodes colored in red below.**

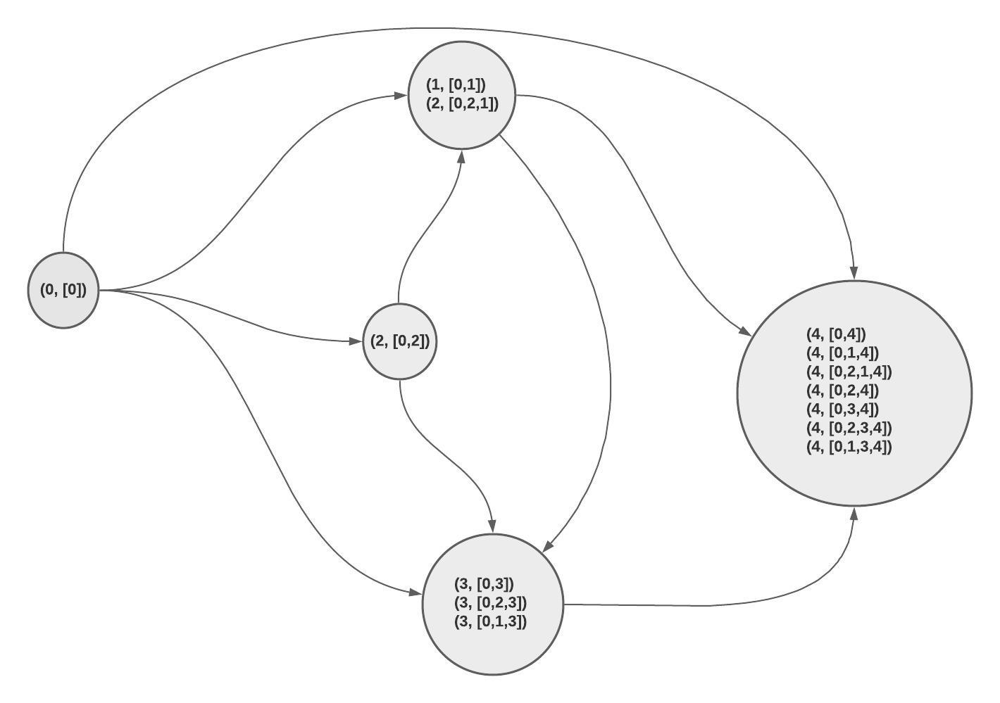
 **So not only that we don't need a visited set, we actually must NOT use one.**



**Algorithm:**

Use an explicit stack

As we traverse the DAG, store the node along with the path leading to it. See sketch below.



**Code:**

def allPathsSourceTarget(graph):

#edges cases:

if not graph:

return []

# build di-graph

d = {}

for i in range(len(graph)):

d[i] = graph[i] # one-way link

# apply DFS on DAG

n = len(graph)

stack = [(0, [0])] # - store noth the (node, and the path leading to it)

res = []

while stack:

node, path = stack.pop()

# check leaf

if node == n - 1:

res.append(path)

# traverse rest

for nei in d[node]:

stack.append((nei, path+[nei]))

return res

**Time Complexity Analysis**

Before we delve into analyzing the time complexity for this solution, let's revisit a few simple algorithms with well-defined time complexities:

When iterating over an array in linear time, we all know our time complexity is O(N) becaue we visit each node once and there is N nodes in total.

# O(N) Time Complexity

for i in range(len(array)):

# do something

Simiarly when we run a DFS algorithm on a graph while maintaining a visited set we are guaranteed every node is visited only once and thus our time complexity is also linear to the number of vertices/nodes O(E+V)

# O(E+V) Time Complexity

stack = []

while stack: # - O(V), V = number of verticies/nodes

node = stack.pop()

visited.add(node)

for nei in d[node]: # - O(E), E = max number of edges a node has

if nei not in visited:

stack.append(nei)

On the other hand, when iterating twice over an array in a nested-loop fashion, we know the cost is O(N^2) because each node is visited N times. And since we have a total of N nodes, which makes the overall cost N^2

# O(N^2) Time Complexity

for i in range(len(array)):

for j in range(len(array)):

# do something

Now back to our algorithm. The time complexity of our algorithm as you may have noticed, is not as straight forward. The time complexity of this solution can be tricky due to the fact that some nodes will be visited a lot more frequently than others **depending on the DAG's shape**. This inconsistency in the cost of visiting each node requires more detailed analysis. From the few examples above we notice that time complexity **really boils down to how many times we are visiting each node/item** in our data-strcuture. Perhaps an initial good guess is that our time complexity is somewhere between O(N) and O(N^2) since some nodes are visited once while others more than once. Another observation we could make is that the leaf/end node is the one with the highest frequency of visits. And if we look closer, we see that the number of visits made to the leaf/end node **must in fact be equal to the number of paths from (start -> end) since all paths eventually lead to it.** Hence we can safely say the leaf/end node is visited K times where K is the number of all paths from start node to end node.

**DAG - Grid example**

Now, what we covered above is true for both the grid-DAG and the general-case DAG, but let's first double-down on the grid-DAG:

Notice that depending on the grid's dimensions:

K can be less than N (as in the 3x3 grid) N=9 > k=6

but can also be more than N (4x4 gird) N=16 < k=20

k = the number of paths (number of visits to the leaf/end node)

Turns out k or the number of paths, can be calculated exactly in Grid-DAG example **using combinatorics:**

In order to reach the (n,m) node, we need to make a total of (m-1) + (n-1) movements:

(m-1) to the right

(n-1) to to the cell below

n = choosing from = a total of (m-1) + (n-1) = (m+n-2)

k = choosing/flipping

nPk = n!/(n-k)! \* k!

nPk = (m+n-2)!/(m-1)! \* (n-1)!

nPk = (2n-2)!/(n-1)! \* (n-1)! for a square grid

Hence overall time complexity (worst case) = O(E + K\*V) = O(2 + K\*V) , E = 2 (each node has only two edges)

Average case = O(2 + K\*V) where K is the average cost to visit a node

(k = cost to visit all nodes / number of nodes)

**DAG General-case**

The advantage we had in the grid DAG is that we knew what our DAG looked like. We knew the size of the graph and we used that to deduce how many paths are going to be there. In the general-case DAG however, we know nothing about the shape, size of our input DAG. Due to the ambigious nature of the input, time-complexity analysis gets even more trickier. Turns out the **best we can do is come up with an upper limit**on the number of paths in the DAG.

**Maximum number of paths for acyclic di-graph with start and end node**

We can arrive at an upper limit in one of two ways:

**1- Finding the Number of Subsets of a Set**

Let's start with stating the obvious: Every path from start to end must contain the start and end nodes and can include anywhere from zero to N-2 intermediate nodes. In other words, its a given that the start and end nodes are included in the path, so we play around with the remaining (N-2) nodes. In set theory lingo that translates to the following:

For a DAG that has N-2 intermediate nodes.

C(N-2, 0) + C(N-2, 1) + C(N-2, 2) + C(N-2, 3) + ....... + C(N-2, N-2)

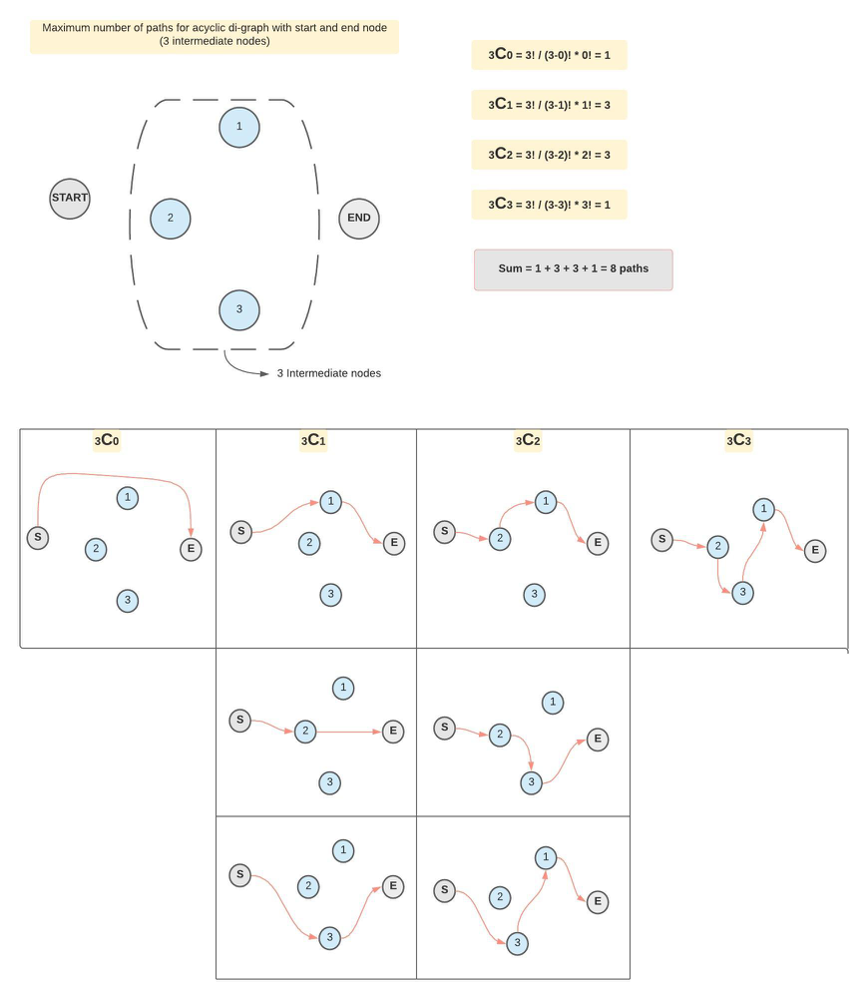
For simplicity, let n = N-2

C(n, 0) = How many ways we can choose zero intermediate nodes between start and end = n!/(n-k)! \* k! = n!/(n! \* 0!) = 1

C(n, 1) = How many ways we can choose 1 intermediate nodes between start and end ...

etc ...

Below is a visual illustration of the possible paths for a sample DAG:



Why did we use combinations nCk and not Permutations nPk?

We should count a group of nodes (only once) and not count every permutation of the same group/combination.

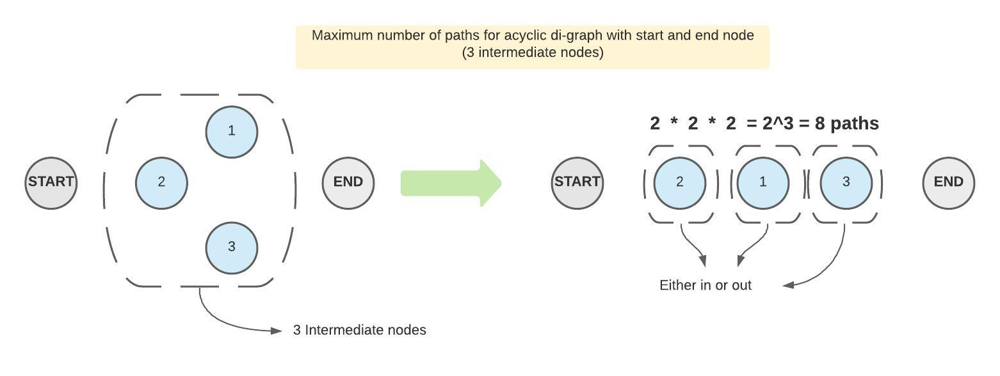
In other words, I should not count : Start -> A -> B -> End and Start -> B -> A -> End as two different paths, because in reality, the actual DAG is one of these two solutions (not both).

**2- Children and chairs**

Another way to find the upper limit mathematically is as follows:

For each one of the intermediate nodes (excluding start and end node) there is one of two options: Either be part of the path, or no. This can be expressed as follows: If we have 3 intermediate nodes, we can treat them as chairs.

[2] \* [2] \* [2] = 2^3 = 8 paths



K = number of paths = 2^(N-2)

Time comlexity = E + k\*V = E + 2^(N-2)\*V

**Refer to Deepseek R1**

**But in above solution, the author saying Backtracking not required for this problem ?**

Okay, let me try to figure out why the author of that solution says backtracking isn't required for the problem. So the problem is LeetCode 797, All Paths From Source to Target, where we need to find all paths from node 0 to node n-1 in a DAG.

**First, I remember that backtracking is usually used when we need to explore all possible paths and then undo the last step to try other options. Like in permutations or combinations, where you add an element, explore, then remove it to try another. But here, maybe there's a different approach.**

**The user mentioned an iterative DFS approach. Let me think about how that works. In recursive DFS with backtracking, we add a node to the path, explore its neighbors, and then remove the node (backtrack) after returning from the recursion. But in an iterative approach using a stack, maybe each path is maintained separately.**

**So in iterative DFS, you can push different paths onto the stack. Each time you process a node, you take the current path, add the neighbor, and push that new path into the stack. Since each path is a separate copy, you don't need to backtrack because each path in the stack represents a different branch. There's no shared state that needs to be undone.**

**For example, when you visit node 0, you create a path [0]. Then, for each neighbor of 0, say 1 and 2, you create new paths [0,1] and [0,2] and push them into the stack. When processing [0,1], you check if 1 is the target. If not, you take its neighbors, add them to the path, and create new entries. Since each path is a separate list, there's no need to backtrack because each stack entry is independent.**

This way, you avoid modifying a single path list that needs to be adjusted after recursion. So backtracking, which involves undoing changes to a shared structure, isn't necessary here because each path is handled as its own copy.

Therefore, the author's point is that using an iterative approach with separate path copies in the stack eliminates the need for explicit backtracking steps. Instead of modifying and then undoing a single path, you just create new paths for each neighbor, which is more memory-intensive but avoids backtracking.

**So the key difference is in how the paths are managed: in recursive backtracking, you reuse a single list and backtrack by removing elements, while in iterative DFS, each path is a separate copy, so no undoing is needed. Hence, backtracking isn't required in the iterative approach.**

The author's statement that "backtracking is not required" in their solution refers to the iterative DFS approach they used, which avoids explicit backtracking by managing paths differently. Here's why backtracking isn't needed in their approach and how it differs from traditional backtracking:

**Key Insight**

In recursive DFS with backtracking, you modify a single shared path (e.g., a List), add/remove nodes during exploration, and backtrack by undoing changes.

In iterative DFS, you avoid backtracking by creating \*\*independent copies of paths\*\* stored on the stack. Each path is treated as a separate entity, eliminating the need to "undo" steps.

**Example: Iterative DFS (No Backtracking)**

def allPathsSourceTarget(graph):

stack = [ [0] ] # Start with path [0]

target = len(graph) - 1

result = []

while stack:

path = stack.pop()

current\_node = path[-1]

if current\_node == target:

result.append(path)

continue

for neighbor in graph[current\_node]:

new\_path = path + [neighbor] # Create a new path copy

stack.append(new\_path) # Push new path to stack

return result

**Java Version**

class Solution {

public List<List<Integer>> allPathsSourceTarget(int[][] graph) {

// Initialize stack with the starting path [0]

Stack<List<Integer>> stack = new Stack<>();

List<Integer> initialPath = new ArrayList<>();

initialPath.add(0);

stack.push(initialPath);

int target = graph.length - 1; // Target node

List<List<Integer>> result = new ArrayList<>(); // To store all valid paths

// Iterative DFS

while (!stack.isEmpty()) {

List<Integer> path = stack.pop(); // Get the current path

int current\_node = path.get(path.size() - 1); // Get the last node in the path

// If the current node is the target, add the path to the result

if (current\_node == target) {

result.add(new ArrayList<>(path)); // Add a copy of the path

continue;

}

// Explore all neighbors of the current node

for (int neighbor : graph[current\_node]) {

List<Integer> new\_path = new ArrayList<>(path); // Create a new path copy

new\_path.add(neighbor); // Add the neighbor to the new path

stack.push(new\_path); // Push the new path to the stack

}

}

return result;

}

}

Time Complexity: O(2^n⋅n), where n*n* is the number of nodes. This is because there can be up to 2^n paths, and each path can have up to n*n* nodes.

Space Complexity: O(2^n⋅n), due to the storage of all paths.

**Why No Backtracking?**

No Shared State:

Each path (e.g., [0, 1, 3], [0, 2, 3]) is a separate list. When a new neighbor is added, a new path is created and pushed to the stack.

There’s no shared list to modify/undo.

Implicit Exploration:

The stack processes paths independently. Once a path is processed, it’s discarded (no need to backtrack).

**Comparison: Recursive DFS (Backtracking Required)**

def allPathsSourceTarget(graph):

result = []

def backtrack(node, path):

if node == len(graph) - 1:

result.append(path.copy())

return

for neighbor in graph[node]:

path.append(neighbor) # Modify shared path

backtrack(neighbor, path)

path.pop() # Backtrack: undo the modification

backtrack(0, [0])

return result

**Why Backtracking Here?**

A single path list is reused across recursive calls.

After exploring a neighbor, you must pop the last node to restore the path’s state for other branches.

**When Can You Avoid Backtracking?**

Backtracking is unnecessary when:

No Shared State: Each path is an independent copy (common in iterative approaches).

No Mutations: Paths are immutable (e.g., strings or tuples in Python).

Copy-on-Write: New paths are created by appending to existing ones (as in the iterative DFS example).

**Tradeoffs**

|  |  |  |  |
| --- | --- | --- | --- |
| **Approach** | **Backtracking?** | **Space Complexity** | **Readability** |
| **Recursive DFS** | Required | Lower (shared path) | Intuitive, clean |
| **Iterative DFS** | Not required | Higher (path copies) | Explicit path copies |

**Conclusion**

The author’s iterative DFS avoids backtracking by creating new path copies for each branch. This simplifies code logic (no pop/undo steps) but uses more memory. Traditional backtracking is still valid but requires careful state management. Both approaches are correct; the choice depends on tradeoffs between memory and code style.

**Refer to**

[DFS and Backtracking](note://WEB75d228b94561abef24dcf8b249091745)

[L2328.Number of Increasing Paths in a Grid (Ref.L329,L797)](note://WEB1ab466bd849e83e490085cec0d601f1d)