Graphs - DFS

Example 1: 547 - Number of Provinces

There are `n` cities. A province is a group of directly or indirectly connected cities and no other cities outside of the group. You are given an `n x n` matrix `isConnected` where `isConnected[i][j] = isConnected[j][i] = 1` if the ith city and the jth city are directly connected, and `isConnected[i][j] = 0` otherwise. Return the total number of provinces.

This is an undirected graph where the graph is given as an adjacency matrix, and the problem is asking for the number of connected components. We can think of each city as a node and each connected component as a province.

Because the graph is undirected, a DFS from any given node will visit every node in the connected component it belongs to. To avoid cycles with undirected graphs, we need to use a set `seen` to track which nodes we have already visited. After performing a DFS on a connected component, all nodes in that component will be inside `seen` (because the DFS will visit every node in the component). Therefor, we can iterate from `0` until `n`, and each time we find a node that hasn't been visited yet, we know we also have a component that hasn't been visited yet, so we perform a DFS to "mark" the component as visited and increment our answer. The use of the set will prevent us from counting the same component more than once.

For convenience, we can convert the adjacency matrix to a hash map that maps nodes to an array of their neighbors before starting.

Depending on the language you're using, if your keys are well defined (like the integers between `0` to `n - 1`), it might be better to use a boolean array instead of a set to implement `seen`.

A traversal on a given `node` will visit every node in the connected component `node` belongs to. Remember that in a binary tree, if you did a traversal starting from the root, you would visit every node in the tree.

We can use a data structure `seen` to tell us which cities we have already visited. This is another standard idea in all graph problems to avoid visiting a node multiple times. When we are performing the traversal and are at a given `node`, we iterate over the neighbors. For each `neighbor`, we first check if `neighbor` has been visited. If it has, we ignore it. If it hasn't we mark it as visited in `seen` and then recursively call `dfs(neighbor)` (just like we did with binary trees).

We iterate over the cities, and if we find a city `i` is not visited yet, we can perform a DFS starting from `i`. As we know, this traversal will visit every node in the connected component that `i` belongs to. After the traversal, `seen` will be updated with the entire province. We can increment the answer, and we don't need to worry about the province we just visited anymore because `seen` will prevent us from revisiting it.

```
from collections import defaultdict

class Solution:
    def findCircleNum(self, isConnected: List[List[int]]) -> int:
        def dfs(node):
            for neighbor in graph[node]:
                # the next 2 lines are needed to prevent cycles
                if neighbor not in seen:
                    seen.add(neighbor)
                    dfs(neighbor)

        # build the graph
        n = len(isConnected)
        graph = defaultdict(list)
        for i in range(n):
            for j in range(i + 1, n):
                if isConnected[i][j]:
                    graph[i].append(j)
                    graph[j].append(i)

        seen = set()
        ans = 0

        for i in range(n):
            if i not in seen:
                # add all nodes of a connected component to the set
                ans += 1
                seen.add(i)
                dfs(i)

        return ans
```

With trees, we are given objects representing the nodes. Here, the nodes aren't exactly given to us. We are simply told that there exists some nodes numbered from `0` to `n - 1`, we are given

information regarding the edges.

Thus we treat the integers from `[0, n - 1]` as the nodes. This is why our `dfs(node)` function is taking an integer as an argument. With trees, we passed the node object as an argument. Here we pass the integer label of the node. The graph only "exists" as an idea. It is up to you to implement a method of representing the nodes and edges and traversing over them.

If you want to implement the DFS iteratively, you can just modify the `dfs` function while keeping everything else the same. Just like with binary trees, we use a stack to perform the traversal. In each while loop iteration, we pop a `node` from the stack, and then the iteration is equivalent to a function call with `node` as the argument.

```
def dfs(start):
    stack = [start]
    while stack:
        node = stack.pop()
        for neighbor in graph[node]:
            if neighbor not in seen:
                seen.add(neighbor)
                stack.append(neighbor)
```

The time complexity of DFS on a graph is slightly different than when it is on a binary tree. With binary tree questions, we argued that each node is visited at most once, and each visit cost $O(1)$. With graphs, we also only visit each node at most once, but the work is not necessarily $O(1)$, because there is a for loop that iterates over the node's neighbors.

The reason visits were $O(1)$ in a binary tree is because a node could have at most 2 children/neighbors, so we didn't need a loop. We just referenced `node.left` and `node.right`. With a graph, a node could have any amount of neighbors, so we need a non-constant loop.

As such, the time complexity for DFS on graphs is usually $O(n + e)$, where n is the number of nodes and `e` is the number of **edges**. In the worst-case scenario, where every node is connected with every other node, $e = n^2$.

- Each node is visited only once
- We iterate over a node's edges only when we are visiting that node
- Because we can only visit a node once, a node's edges are iterated over once
- Therefore, all edges are iterated over only once, which costs $O(e)$

This is similar to the argument we made in the sliding window article that justified an `$O(n)$` time complexity despite the nested `while` loop. The nested while loop could only iterate n

times across the entire algorithm. Here, the for loop inside the function iterates e times total across the entire algorithm.

Technically in this problem, the time complexity is $O(n^2)$ because the input is given as an adjacency matrix, so we **always** need $O(n^2)$ to build the hash map. The e is dominated by n^2 (because $O(e < n^2)$, so it can be ignored.

What about the space complexity? When we build `graph`, we are storing all the edges in arrays. We also need some space for the recursion call stack ($O(n)$ in the worst case) as well as for `seen`. Therefore the space complexity is $O(n + e)$.

The space complexity isn't $O(n^2)$ because `e` is not necessarily dominated. In the worst case scenario, $e = n^2$, but e is still independent of n. In the time complexity, we **always** iterated over the entire matrix to build the graph, but in terms of space complexity, the hash map only grows if the edges actually exist. So the more general $O(n + e)$ is the space complexity.

Example 2: 200 - Number of Islands

Given an `m x n` 2D binary `grid` which represents a map of `1` (land) and `0` water, return the number of islands. An island is surrounded by water and is formed by connecting adjacent land cells horizontally or vertically.

As mentioned earlier, a graph can be given in the form of a matrix where squares are nodes and their neighbors are the adjacent squares. In this problem, it says that land is connected horizontally (left/right) or vertically (up/down). We can think of each land square as a node, and the up/down/left/right relationship forming edges. The problem is asking us for the number of islands, aka the number of connected components.

This is exactly the same problem as Example 1 (find the number of connected components in an undirected graph); it's just the format of the graph that is different.

```
class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
        def valid(row, col):
            return 0 <= row < m and 0 <= col < n and grid[row][col] == "1"

        def dfs(row, col):
            for dx, dy in directions:
                next_row, next_col = row + dy, col + dx
                if valid(next_row, next_col) and (next_row, next_col) not in
seen:
```

```
                    seen.add((next_row, next_col))
                    dfs(next_row, next_col)

        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
        seen = set()
        ans = 0
        m = len(grid)
        n = len(grid[0])
        for row in range(m):
            for col in range(n):
                if grid[row][col] == "1" and (row, col) not in seen:
                    ans += 1
                    seen.add((row, col))
                    dfs(row, col)

        return ans
```

Here's the DFS function implemented iteratively:

```
def dfs(start_row, start_col):
    stack = [(start_row, start_col)]
    while stack:
        row, col = stack.pop()
        for dx, dy in directions:
            next_row, next_col = row + dy, col + dx
            if valid(next_row, next_col) and (next_row, next_col) not in
seen:
                seen.add((next_row, next_col))
                stack.append((next_row, next_col))
```

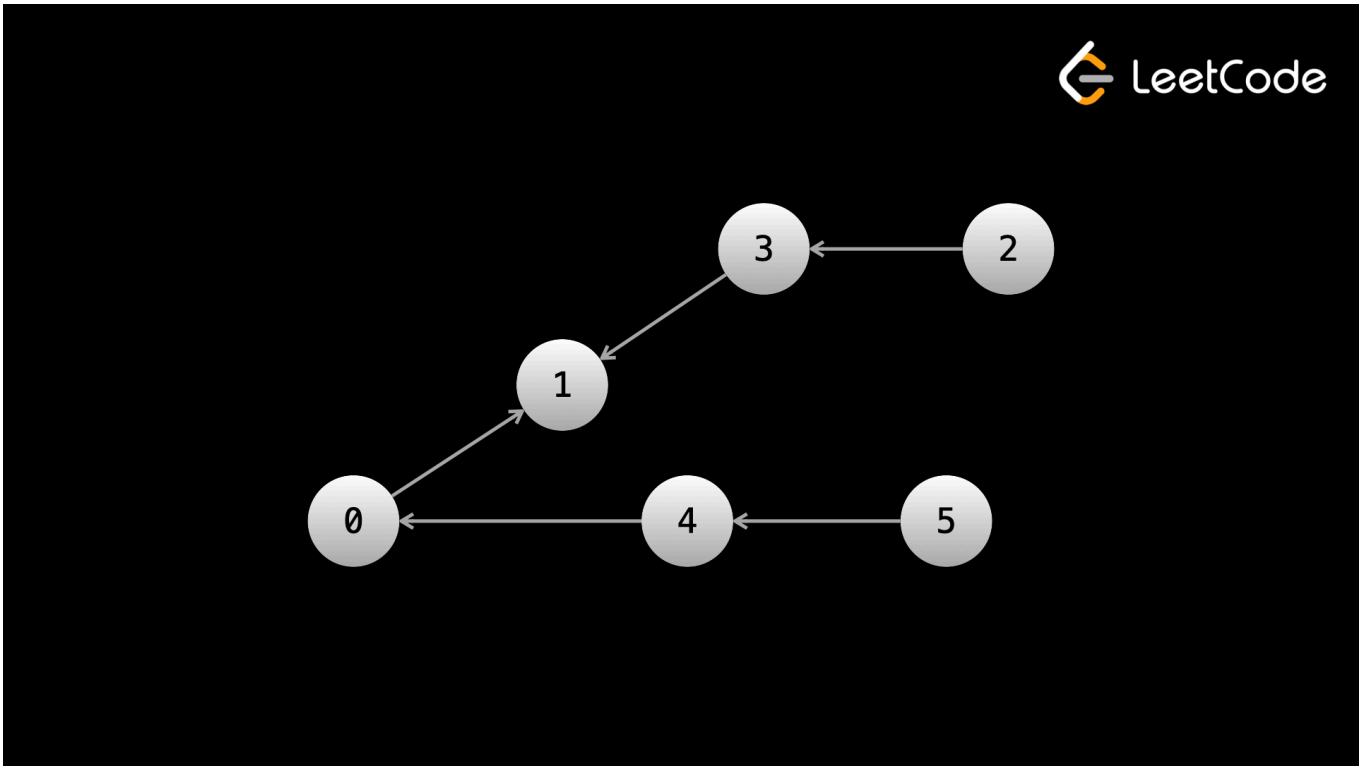
Note we can avoid using the set by modifying the input. The point of the set is to avoid visiting the same square multiple times. We only visit squares with values `"1"`, so instead of putting a square in a set, we could just change that square's value to `"0"`. However, some interviewers may not want you to modify the input (especially if it is something passed by reference like an array).

We said before that DFS on a graph has a time complexity of $O(nodes + edges)$. This was because we didn't know how many edges a given node could have. However, here the problem explicitly defines that a node can have (at most(4 edges. Therefore, the work done at each

node is once again $O(1)$, and since we are only visiting each node once, the time complexity is equal to the number of nodes, which is $O(mn)$.

Example 3: 1466 - Reorder Routes to Make All Paths Lead to the City Zero

There are `n` cities numbered from `0` to `n - 1` and `n - 1` roads such that there is only one way to travel between two different cities. Roads are represented by `connections` where `connections[i] = [x, y]` represents a road from city `x` to city `y`. The edges are directed. You need to swap the direction of some edges so that every city can reach city `0`. Return the minimum number of swaps needed.



The format of this graph actually fits the technical definition of a tree, with `0` being the root. This is typical when a problem states that there are `n` nodes and `n - 1` edges (and you can't have multiple edges between the same two nodes).

Here, we have a directed graph given as an array of edges. What edges do we need to swap? The problem states that there is only one way to travel between two different cities. We want every city to be able to reach `0`, and because there is only one road between cities, **all** roads must be directed towards `0`. This means we can traverse **away** from `0`, and anytime we see that an edge is pointing away from `0`, we know we need to swap it.

Although the graph is a directed graph, let's convert it into an undirected one so that we can reach all nodes from `0`. Then we can do a DFS starting at `0`. During this DFS, every traversal

we do is away from `0`, so when we see an edge that we are crossing `(node, neighbor)` is in `connections`, we know we need to swap it (increment the answer).

Let's say we start a DFS from `0`. Because we use `seen` to prevent revisiting nodes, we will always be moving away from `0` (since we started there).

If we are at node `A` and we move to node `B`, we know the edge `A -> B` must be pointing away from `0`. We can put all the edges in the input into a set, and then when we perform the traversal, we check if each edge `node -> neighbor` is in the set. If `node -> neighbor` is in the set, it means that it was in the original input and since it is pointing away from `0`, we must increment our answer as it is an edge that needs to be flipped. The reason we put the original edges in a set is so that this check can be $O(1)$.

The only problem is that because the edges are directed, starting a DFS from `0` will likely not visit every node. To remedy this, when we build our `graph` from the input; we can treat the edges as undirected. This is solely to enable us to perform the DFS starting from `0`.

Then we start a DFS from `0` and every `node -> neighbor` edge we encounter that is in the original input must be flipped.

Note: In C++, we are using `set` instead of `unordered_set`. It allows us to use mutable elements, but in return, the time complexity of operations is $O(logn)$ instead of $O(1)$. This is for the sake of simpler code; you could achieve $O(1)$ complexity by using `unordered_set` and converting the pairs to an immutable form like in Java.

```
class Solution:
    def minReorder(self, n: int, connections: List[List[int]]) -> int:
        roads = set()
        graph = defaultdict(list)
        for x, y in connections:
            graph[x].append(y)
            graph[y].append(x)
            roads.add((x, y))

        def dfs(node):
            ans = 0
            for neighbor in graph[node]:
                if neighbor not in seen:
                    if (node, neighbor) in roads:
                        ans += 1
                    seen.add(neighbor)
                    ans += dfs(neighbor)
```

```
            return ans

        seen = {0}
        return dfs(0)
```

Iterative version:

```
class Solution:
    def minReorder(self, n: int, connections: List[List[int]]) -> int:
        roads = set()
        graph = defaultdict(list)
        for x, y in connections:
            graph[x].append(y)
            graph[y].append(x)
            roads.add((x, y))

        ans = 0
        stack = [0]
        seen = {0}
        while stack:
            node = stack.pop()
            for neighbor in graph[node]:
                if neighbor not in seen:
                    if (node, neighbor) in roads:
                        ans += 1
                    seen.add(neighbor)
                    stack.append(neighbor)

        return ans
```

The time and space complexity of this algorithm is $O(n)$ because we only visit each node once, do constant work, and are told the number of edges is $n-1 = O(n)$. `roads`, `graph`, and `seen` all take up at most $O(n)$ space.

Example 4: 841 - Keys and Rooms

There are `n` rooms labeled from `0` to `n - 1` and all the rooms are locked except for room `0`. Your goal is to visit all the rooms. When you visit a room, you may find a set of distinct keys in it.

Each key has a number on it , denoting which room it unlocks, and you can take all of them with you to unlock the other rooms. Given an array `rooms` where `rooms[i]` is the set of keys that you can obtain if you visited room `i`, return true if you can visit all the rooms, or false otherwise.

In the previous 3 examples, we have seen a graph given as an adjacency matrix, a graph in the form of a matrix, and a graph given as an array of edges. Here, `rooms[i]` is an array of other rooms we can visit from the current room, which makes this a graph given as an adjacency list. We start at room `0` and need to visit every room. At every node `i`, the neighbors are `rooms[i]`. If we can start a DFS at `0` and visit every node, then the answer is true. How can we tell how many rooms we visited at the end of the DFS? All the nodes we visited are stored in `seen`.

When we visit a room, we find some keys that enable us to visit other rooms. Thus we can model the problem as a graph. The rooms are nodes and the keys are edges.

The input to the graph is the most convenient one - an adjacency list. We don't need to build `graph` like we did in the previous examples because the input already sesrves that function - if we want to find the neighbors of a given node `i`, we can simply check `rooms[i]`.

```
class Solution:
    def canVisitAllRooms(self, rooms: List[List[int]]) -> bool:
        def dfs(node):
            for neighbor in rooms[node]:
                if neighbor not in seen:
                    seen.add(neighbor)
                    dfs(neighbor)

        seen = {0}
        dfs(0)
        return len(seen) == len(rooms)
```

Iterative:

```
class Solution:
    def canVisitAllRooms(self, rooms: List[List[int]]) -> bool:
        seen = {0}
        stack = [0]

        while stack:
            node = stack.pop()
```

```
            for neighbor in rooms[node]:
                if neighbor not in seen:
                    seen.add(neighbor)
                    stack.append(neighbor)

        return len(seen) == len(rooms)
```

Adjacency lists are the most convenient input format when the nodes are numbered from `0` to `n - 1` because we don't need to convert it to a hash map - it basically is already in that format. So the only extra space we use here is in `seen` and the recursion call stack, which both are $O(n)$. The time complexity is $O(n + e)$ as we visit each node once and the for loops inside each visit will iterate up to `e` times total across the entire algorithm.

Example 5: 1557 - Minimum Number of Vertices to Reach All Nodes

Given a directed acyclic graph, with `n` vertices numbered from `0` to `n - 1`, and an array `edges` where `edges[i] = [x, y]` represents a directed edge from node `x` to node `y`. Find the smallest set of vertices from which all nodes in the graph are reachable.

This can be rephrased as the smallest set of nodes that cannot be reached from other nodes, because if a node can be reached from another node, then we would rather just include the "parent" rather than the "child" in our set.

A node cannot be reached from another node if it has an **indegree** of 0 (no edges are entering the node). Therefore, we can just find the indegree of all nodes and only include the ones with a zero indegree.

Note: If the graph had cycles, we would run into some edge cases. Imagine if the graph was just one cycle (a circle). Which node would we return? Technically, returning any of them would be correct. Our algorithm, however, would return nothing because none of the nodes would have an indegree of 0. Fortunately, the given graph is acyclic, so we don't have to worry about these cases.

```
class Solution:
    def findSmallestSetOfVertices(self, n: int, edges: List[List[int]]) ->
List[int]:
        indegree = [0] * n
        for _, y in edges:
            indegree[y] += 1
```

```
        return [node for node in range(n) if indegree[node] == 0]
```

Example 5: 1557 - Minimum Number of Vertices to Reach All Nodes

Given a directed acyclic graph, with `n` vertices numbered from `0` to `n - 1`, and an array `edges` where `edges[i] = [x, y]` represents a directed edge from node `x` to node `y`. Find the smallest set of vertices from which all nodes in the graph are reachable.

This can be rephrased as the smallest set of nodes that cannot be reached from other nodes, because if a node can be reached from another node, then we would rather just include the "parent" rather than the "child" in our set.

A node cannot be reached from another node if it has an **indegree** of 0 (no edges are entering the node). Therefore, we can just find the indegree of all nodes and only include the ones with a zero indegree.

Note: If the graph had cycles, we would run into some edge cases. Imagine if the graph was just one cycle (a circle). Which node would we return? Technically, returning any of them would be correct. Our algorithm, however, would return nothing because none of the nodes would have an indegree of 0. Fortunately, the given graph is acyclic, so we don't have to worry about these cases.

```
class Solution:
    def findSmallestSetOfVertices(self, n: int, edges: List[List[int]]) ->
List[int]:
        indegree = [0] * n
        for _, y in edges:
            indegree[y] += 1

        return [node for node in range(n) if indegree[node] == 0]
```

SEE PROBLEMS 1971, 323, 695, 2368