2360. Longest Cycle in a Graph

Hard | Depth-First Search | Graph | Topological Sort — Leetcode Link

Problem Description

You are given a directed graph that consists of `n` nodes. These nodes are numbered from `0` to `n - 1`. Each node in the graph has a maximum of one outgoing edge to another node. The representation of this graph is an array `edges` of size `n`, where each element `edges[i]` signifies a directed edge from node `i` to node `edges[i]`. If a node `i` does not have an outgoing edge, then it is indicated by `edges[i] == -1`.

The task is to determine the length of the longest cycle within the graph. A cycle is a path that begins and ends at the same node. If no cycle exists in the graph, the function should return `-1`.

Intuition

To find the longest cycle in a directed graph where each node has at most one outgoing edge, we can think of this problem as exploring paths starting from each node until we either revisit a node, indicating a cycle, or reach a node without an outgoing edge.

We proceed as follows:

1. Iterate over all nodes, using each as a potential cycle starting point.
2. For each starting node, move to the next node using the outgoing edge and mark each visited node along the path.
3. Keep tracking visited nodes to recognize when we encounter a cycle.
4. When we find ourselves at a node that we have visited previously, it indicates the start of a cycle. We measure the length of the path from this node back to itself to calculate cycle length.
5. If we reach a node without an outgoing edge (`edges[i] == -1`), we conclude there's no cycle from this path.
6. We keep track of the maximum length found as we explore cycles from different starting points.

The solution provided uses these steps with an array `vis[]` to keep track of visited nodes, ensuring each node is processed only once, which helps in maintaining a linear time complexity relative to the number of nodes.

Solution Approach

The implementation of the solution follows the steps outlined in the intuition:

1. Initialize a list `vis` with `n` elements, all set to `False`, to record whether a node has been visited (`True`) or not (`False`).

2. Set a variable `ans` to `-1`. This will hold the length of the longest cycle found so far.

3. Iterate over all nodes `i` (from `0` to `n-1`). For each node:

   a. If `vis[i]` is `True`, the node has already been visited in some previous path exploration, so we skip to the next node.

   b. If `vis[i]` is `False`, initialize an empty list `cycle` to store the path taken starting from the current node (it helps us identify cycles later).

   c. Inside a while loop, visit nodes following the outgoing edges (`edges[j]`) until you reach a node with no outgoing edge (`-1`) or revisit a node (indicating a cycle).

   ```
   - For each visited node `j`, set `vis[j]` to `True` and append `j` to the `cycle` list.
   - Update `j` to be the next node to visit, i.e., `j` becomes `edges[j]`.
   ```

4. Once the while loop ends, check if it was due to reaching a node without an outgoing edge. If `j == -1`, continue to the next node `i`.

5. If the loop ended due to revisiting a node `j`:

   a. Determine the index `k` in `cycle` where the node `j` occurs, indicating the start of the cycle. This is done using a generator expression that searches for the first occurrence of `j` in `cycle`. If `j` is not found (which won't be the case here since `j` caused the termination of the loop), `inf` (infinity) is used as the default value.

   b. Calculate the length of the cycle as the difference between the total number of nodes in the `cycle` list and the index `k`. This gives us the length from the start of the cycle to the end of `cycle`.

6. Update `ans` by taking the maximum of the current `ans` and the length of the cycle found in step 5b.

7. After the loop over all nodes completes, the variable `ans` holds the length of the longest cycle, or `-1` if no cycle was found. Return `ans`.

The solution makes efficient use of the `vis` array for marking visited nodes to avoid reprocessing and the `cycle` list for tracking the current path to identify cycles quickly. The algorithm's complexity is O(n) since each node and edge is visited at most once.

Example Walkthrough

Let's walk through an example to illustrate the solution approach using the steps outlined above.

Suppose we are given the following directed graph as an `edges` array:

```
edges = [1, 2, -1, 4, 5, 3]
```

This graph has 6 nodes (n = 6). The edges array implies:

- Node 0 has an outgoing edge to node 1.
- Node 1 has an outgoing edge to node 2.
- Node 2 has no outgoing edge (indicated by -1).
- Node 3 has an outgoing edge to node 4.
- Node 4 has an outgoing edge to node 5.
- Node 5 has an outgoing edge to node 3.

We are going to determine the length of the longest cycle.

1. Start with initialization:
   - `vis` = [False, False, False, False, False, False]
   - `ans` = -1
2. Begin iterating over all nodes:
3. For `i = 0`:
   - `vis[0]` is `False`, so we proceed.
   - Start an empty list `cycle = []`.
   - Begin a while loop: Visit node 1, set `vis[0] = True`, append to `cycle = [0]`. Visit node 2, set `vis[1] = True`, append to `cycle = [0, 1]`. Since node 2 has no outgoing edge, the loop ends.
   - No cycle is detected.
4. For `i = 1`:
   - `vis[1]` is already `True`, so we skip this iteration.
5. For `i = 2`:
   - `vis[2]` is `False`. However, node 2 has no outgoing edge (`edges[2] == -1`), so we can't start a cycle from here.
   - Set `vis[2] = True`.
6. For `i = 3`:
   - `vis[3]` is `False`, so we proceed.
   - Start an empty list `cycle = []`.
   - Begin a while loop: Visit node 4, set `vis[3] = True`, append to `cycle = [3]`. Visit node 5, set `vis[4] = True`, append to `cycle = [3, 4]`. Visit node 3 (again), set `vis[5] = True`, and append to `cycle = [3, 4, 5]`.
   - We revisited node 3, which indicates a cycle.
   - Determine the index `k` in `cycle` where node 3 occurs, `k = 0`.
   - Calculate the length of the cycle: `len(cycle) - k = 3 - 0 = 3`.
   - Update `ans` = max(-1, 3) = 3.
7. For `i = 4` and `i = 5`:
   - Both `vis[4]` and `vis[5]` are `True`, so we skip these iterations.

After completing the loop, we find that `ans = 3`, which indicates that the longest cycle in the graph has a length of 3. Hence, the function would return `3`.

Python Solution

```python
from typing import List

class Solution:
    def longestCycle(self, edges: List[int]) -> int:
        # Initialize the number of nodes in the edge list
        num_nodes = len(edges)
        # Create a visitation status list to keep track of visited nodes
        visited = [False] * num_nodes
        # Initialize the answer to -1, which stands for no cycle found
        longest_cycle_length = -1

        # Iterate over each node
        for node in range(num_nodes):
            # Skip processing if the current node has been visited
            if visited[node]:
                continue

            # Initialize the node for cycle checking
            current_node = node
            # Initialize list to store nodes in the current cycle
            node_cycle = []

            # Continue traversing the graph unless we hit a node
            # that points to -1 or it has been visited
            while current_node != -1 and not visited[current_node]:
                # Mark the node as visited
                visited[current_node] = True
                # Append current node to the cycle
                node_cycle.append(current_node)
                # Move to the next node in the graph
                current_node = edges[current_node]

            # If the end of an edge chain points to -1, a cycle isn't possible
            if current_node == -1:
                continue

            # Calculate the length of the cycle. To do this, we find the index of
            # the node that we revisited which caused the cycle detection
            cycle_length = len(node_cycle)
            # Find the starting index of the cycle within node_cycle list
            cycle_start_index = next((k for k in range(cycle_length) if node_cycle[k] == current_node), float('inf'))
            # Update the longest_cycle_length with the maximum
            longest_cycle_length = max(longest_cycle_length, cycle_length - cycle_start_index)

        # Return the length of the longest cycle, or -1 if no cycle is found
        return longest_cycle_length
```

Java Solution

```java
class Solution {
    public int longestCycle(int[] edges) {
        int numberOfNodes = edges.length;
        boolean[] visited = new boolean[numberOfNodes]; // This array holds whether a node has been visited
        int maxCycleLength = -1; // Store the length of the longest cycle found, -1 if none

        // Iterate through all nodes
        for (int startNode = 0; startNode < numberOfNodes; ++startNode) {
            // Skip if the current node has been visited
            if (visited[startNode]) {
                continue;
            }
            int currentNode = startNode;
            List<Integer> cycle = new ArrayList<>(); // Current potential cycle path

            // Traverse the graph until a loop is found or there are no more nodes to visit
            for (; currentNode != -1 && !visited[currentNode]; currentNode = edges[currentNode]) {
                visited[currentNode] = true; // Mark this node as visited
                cycle.add(currentNode); // Add the current node to the cycle
            }

            // If a loop is detected, calculate its length
            if (currentNode != -1) {
                // Find the index of the node where the loop starts
                for (int cycleIndex = 0; cycleIndex < cycle.size(); ++cycleIndex) {
                    if (cycle.get(cycleIndex) == currentNode) {
                        // Cycle length is the size of the cycle minus the index of the start node of the loop
                        maxCycleLength = Math.max(maxCycleLength, cycle.size() - cycleIndex);
                        break;
                    }
                }
            }
        }
        return maxCycleLength; // Return the length of the longest cycle found
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int longestCycle(vector<int>& edges) {
        int numNodes = edges.size();
        vector<bool> visited(numNodes, false);
        int longestCycleLength = -1;  // Initialize with -1 to represent no cycle found.

        // Loop through each node to find the longest cycle, if any.
        for (int start = 0; start < numNodes; ++start) {
            // Skip if the node has been visited.
            if (visited[start]) {
                continue;
            }

            // Use two pointers to traverse the graph and record the cycle.
            int current = start;
            vector<int> cycleNodes;

            // Explore the graph to find a cycle.
            for (; current != -1 && !visited[current]; current = edges[current]) {
                visited[current] = true; // Mark the node as visited.
                cycleNodes.push_back(current); // Add to cycle list.
            }

            // If we did not encounter a cycle, continue with the next node.
            if (current == -1) {
                continue;
            }

            // Check the recorded nodes to determine the cycle's length.
            for (int idx = 0; idx < cycleNodes.size(); ++idx) {
                // Find the start index of the cycle within the cycle list.
                if (cycleNodes[idx] == current) {
                    // Calculate and store the maximum cycle length found so far.
                    longestCycleLength = max(longestCycleLength, static_cast<int>(cycleNodes.size() - idx));
                    break; // Break as we found the cycle start point.
                }
            }
        }
        return longestCycleLength; // Return the longest cycle length found.
    }
};
```

Typescript Solution

```typescript
// Function to find the length of the longest cycle in a graph represented as an array of edges
function longestCycle(edges: number[]): number {
    const nodeCount = edges.length;  // Total number of nodes in the graph
    const visited = new Array(nodeCount).fill(false);  // An array to keep track of visited nodes
    let longestCycleLength = -1;  // Initialize the length of the longest cycle as -1 (indicating no cycles)

    // Iterate through each node to check for cycles
    for (let i = 0; i < nodeCount; ++i) {
        // Skip already visited nodes
        if (visited[i]) {
            continue;
        }

        let currentNode = i;  // Start from the current node
        const currentCycle: number[] = [];  // List to keep track of the current cycle

        // Traverse the graph using the edges until a visited node or the end (-1) is reached
        while (currentNode != -1 && !visited[currentNode]) {
            // Mark the current node as visited
            visited[currentNode] = true;

            // Add the current node to the cycle list
            currentCycle.push(currentNode);

            // Move to the next node by following the edge
            currentNode = edges[currentNode];
        }

        // Check if the traversal ended on an already visited node, indicating a cycle
        if (currentNode == -1) {
            // If no cycle is found, move on to the next node
            continue;
        }

        // If a cycle is found, calculate the length of the cycle
        for (let nodeIndex = 0; nodeIndex < currentCycle.length; ++nodeIndex) {
            // Check if the cycle loops back to the starting node
            if (currentCycle[nodeIndex] == currentNode) {
                // Update the length of the longest cycle if the current cycle is longer
                longestCycleLength = Math.max(longestCycleLength, currentCycle.length - nodeIndex);
                break;  // Exit the loop as the cycle length has been found
            }
        }
    }

    // Return the length of the longest cycle found in the graph
    return longestCycleLength;
}

// Note: The edges array represents a directed graph where each position i in the array
// is a node, and its value edges[i] is the node that i is directed to (-1 if it has no outgoing edges).
```

Time and Space Complexity

Time Complexity

The given Python code aims to find the length of the longest cycle in a directed graph where the list `edges` represents adjacency information: an index `i` points to `edges[i]`.

For each node, the algorithm checks whether it has been visited. If a node is unvisited, it starts a DFS-like traversal by following edges until it either hits a previously visited node (indicating the beginning of a cycle) or an end (-1).

- The worst-case scenario for time complexity occurs when each node is part of a complex cycle. The outer loop runs `n` times where `n` is the number of nodes. For each node, the inner while loop could potentially run up to `n` times in the case of one long cycle. Therefore, the worst-case time complexity of the algorithm is `O(n^2)`.

Space Complexity

The space complexity of the algorithm is determined by the additional space used:

- The `vis` list is used to keep track of visited nodes, and its size is `n`.
- The `cycle` list, in the worst case, might contain all nodes if there is a single cycle involving all nodes, which would also be `n`.

Hence, the overall space complexity is `O(n)` due to the storage required for the `vis` and `cycle` lists, which grow linearly with the number of nodes `n`.