2077. Paths in Maze That Lead to Same Room

Medium Graph

Leetcode Link

Problem Description

The problem involves finding the confusion score of a maze, which is determined by counting the number of unique cycles of length 3 among the rooms. A cycle of length 3 would look like a trip from one room to a second, then to a third, and back to the first, without repetition of rooms or revisiting the starting room before the cycle is complete. The array `corridors` provides the connections between rooms, where each connection enables two-way travel between the connected rooms.

For example, if we have a cycle `1 -> 2 -> 3 -> 1`, it counts as one valid cycle. We need to ensure that each trio of rooms forms exactly one cycle for counting purposes. Cycles with more than three rooms or cycles that don't return to the starting room after exactly three steps are not considered in the confusion score.

The goal is to calculate the total number of these length 3 cycles in the entire maze based on the corridors provided.

Intuition

To find all possible cycles of length 3, we first convert the `corridors` list into a graph representation where each room has a list of directly connected rooms. A `defaultdict(set)` is used to facilitate this as it automatically handles the creation of keys and prevents duplicate entries for connections between rooms.

Once we have the graph, for each room `i`, we look at all pairs of its connected rooms (using the `combinations` function from the `itertools` module). For every pair of connected rooms `j` and `k`, we check if there's a direct connection from `j` to `k`. If such a connection exists, we've found a cycle of length 3 (`i -> j -> k -> i`), so we increment a counter `ans`. However, each cycle will be counted three times (once for each room in the cycle as the starting room), so we'll divide the total count by 3 to get the correct number of unique cycles.

By iterating through all rooms and their connections, we comprehensively check every possibility for cycles of length 3 and calculate the confusion score to return it.

Solution Approach

The Reference Solution Approach employs a graph data structure, a combinations utility, and some simple arithmetic. Here's a step-by-step breakdown of the implementation:

1. **Graph Representation**: First, we need to represent the maze as a graph where the nodes are rooms, and edges are the corridors between them. We use a `defaultdict(set)` from Python's collections module to make this easy. A `set` is chosen for each room's connections because it prevents duplication of corridors and allows for quick membership testing.
2. **Building the Graph**: With the input `corridors` list, which contains pairs of rooms connected by corridors, we iterate through each pair. For each corridor `(a, b)`, we add room `b` to the set of connections for room `a` and vice versa, effectively constructing an undirected graph.
3. **Searching for Cycles**: To find all unique 3-room cycles, we look at each room `i` and consider all combinations of its connected rooms. We use Python's `itertools.combinations(..., 2)` to generate all unique pairs of connected rooms `(j, k)` without repetition.
4. **Cycle Validation**: For each pair of rooms `(j, k)` connected to room `i`, we check if `j` and `k` are directly connected to each other - this would complete the cycle `i -> j -> k -> i`. If a direct connection exists, we increment a counter `ans`. This process ensures that we only count cycles once and that they are of length 3.
5. **Avoiding Double Counting**: Since each cycle appears three times (once starting at each room), we divide the total count by 3 at the end to get the number of unique cycles. The floor division operator `//` ensures that the result is an integer.

Here is a snippet of how this is being implemented in code:

```
# g = defaultdict(set)
# for a, b in corridors:
#     g[a].add(b)
#     g[b].add(a)

# ans = 0
# for i in range(1, n + 1):
#     for j, k in combinations(g[i], 2):
#         if k in g[j]:
#             ans += 1
#
# return ans // 3
```

In summary, the solution involves constructing an undirected graph, identifying all possible unique cycles of length 3 by examining each node's connections, validating those cycles, and then ensuring that we count each cycle only once by dividing the count by three. This approach leads to an efficient calculation of the confusion score for any given maze represented by its corridors.

Example Walkthrough

Let's consider a small maze with 4 rooms and a set of corridors that connect them:

Corridors: `[(1, 2), (1, 3), (2, 3), (2, 4), (3, 4)]`

Graph Representation

Following the solution approach, we first convert the list of corridors into a graph representation.

```
1. Room 1: connected to Room 2 and Room 3
2. Room 2: connected to Room 1, Room 3, and Room 4
3. Room 3: connected to Room 1, Room 2, and Room 4
4. Room 4: connected to Room 2 and Room 3
```

Building the Graph

The graph constructed from the corridors looks like this:

```
g = defaultdict(set)
g[1] = {2, 3, 4}
g[2] = {1, 3, 4}
g[3] = {1, 2, 4}
g[4] = {2, 3}
```

Searching for Cycles

Next, we consider each room and look for all combinations of connected rooms to find cycles of length 3.

For room 1: The combinations of connected rooms are (2, 3).

For room 2: The combinations of connected rooms are (1, 3), (1, 4), and (3, 4).

For room 3: The combinations of connected rooms are (1, 2), (1, 4), and (2, 4).

For room 4: The combinations of connected rooms are (2, 3).

Cycle Validation

We then check whether the combinations actually form cycles of length 3.

- For room 1, the pair (2, 3) is connected, forming a cycle: `1 -> 2 -> 3 -> 1`.
- For room 2, the pair (3, 4) is connected, forming a cycle: `2 -> 3 -> 4 -> 2`.
- For room 3, the pair (2, 4) is connected, forming a cycle: `3 -> 2 -> 4 -> 3`.
- For room 4, the pair (2, 3) is connected, but this cycle was already counted when considering room 2, so it's not unique.

Avoiding Double Counting

Each of the valid cycles identified will appear three times, once for each room in the cycle. We've found the cycles:

```
1 2 3 -> 1 -> 2 -> 3
2 3 4 -> 2 -> 3 -> 4
3 2 4 -> 3 -> 2 -> 4
```

These are counted for rooms 1, 2, and 3 respectively. Since each of these cycles is counted thrice (one from each room's perspective), we get a total count of 3 cycles. However, there is only one unique cycle for each set.

Final Answer

By dividing the total count by 3, we obtain the final answer:

```
ans = 3 // 3 = 1
```

So, the confusion score for the maze with these corridors is 1. There is only one unique cycle of length 3 among the rooms in this maze.

Python Solution

```
from collections import defaultdict
from itertools import combinations

class Solution:
    def numberOfPathsInSelf, numRooms: int, corridors: List[List[int]]) -> int:
        # Create a graph as a dictionary with default value type 'set', for adjacency list representation
        graph = defaultdict(set)

        # Build the graph, where each node has a set of its adjacent nodes
        for first, second in corridors:
            graph[first].add(second)
            graph[second].add(first)

        # Initialize counter to keep track of the number of triangular paths
        triangularPathsCount = 0

        # Iterate through each node
        for node, neighbors in graph.items():
            # For each node, get all possible pairs of adjacents nodes
            for neighbor1, neighbor2 in combinations(neighbors, 2):
                # Check if this pair of nodes forms a triangle with the current node
                if neighbor1 in graph[neighbor2]:
                    # If so, increment the counter
                    triangularPathsCount += 1

        # Since each triangle is counted three times, divide the result by 3
        return triangularPathsCount // 3
```

Java Solution

```
class Solution {
    public int numberOfPaths(int n, int[][] corridors) {
        // Graph represented as an array of HashSets where each hashset is a list of connected nodes
        HashSet<Integer>[] graph = new HashSet[n + 1];

        // Initialize each node's adjacency list
        for (int i = 0; i <= n; i++) {
            graph[i] = new HashSet<>();
        }

        // Build the graph from corridor connections
        for (int[] corridor : corridors) {
            int node1 = corridor[0];
            int node2 = corridor[1];
            graph[node1].add(node2);
            graph[node2].add(node1);
        }

        // Count the number of valid paths
        int pathCount = 0;

        // Iterate over every node to find potential triangles
        for (int currentNode = 1; currentNode <= n; currentNode++) {
            // Get neighbors of the current node
            List<Integer> neighbors = new ArrayList(graph[currentNode]);
            int neighborCount = neighbors.size();
            // Check every pair of neighbors to find a triangle
            for (int i = 0; i < neighborCount; ++i) {
                int neighbor1 = neighbors.get(i);
                for (int neighbor2 = neighbors.get(i);

                    // If the neighbors are also connected, we found a triangle, increment the count
                    if (graph[neighbor1].contains(neighbor2)) {
                        pathCount++;
                    }
                }
            }
        }

        // Since each triangle is counted 3 times (once for each vertex), divide by 3 to get the correct count
        return pathCount / 3;
    }
}
```

C++ Solution

```
class Solution {
public:
    int numberOfPaths(int n, vector<vector<int>>& corridors) {
        // Initialize an adjacency list for the graph where each node
        // has a set of connected nodes (to avoid duplicates and unordered graph)
        vector<unordered_set<int>> graph(n + 1);

        // Populate the graph with corridors data
        for (const auto& corridor : corridors) {
            int node1 = corridor[0], node2 = corridor[1];
            // Since this is an undirected graph, add each node to the other's adjacency list
            graph[node1].insert(node2);
            graph[node2].insert(node1);
        }

        // Initialize a variable to store the number of triangular paths found
        int answer = 0;

        // Iterate over each node in the graph to check for triangular paths
        for (int currentNode = 1; currentNode <= n; ++currentNode) {
            // Get a list of neighbors for the current node for easier iteration
            vector<int> neighbors;
            neighbors.assign(graph[currentNode].begin(), graph[currentNode].end());

            // Iterate over pairs of neighbors to check if they are also connected to each other
            for (int i = 0; i < neighbors.size(); ++i) {
                for (int j = i + 1; j < neighbors.size(); ++j) {
                    int neighbor1 = neighbors[i], neighbor2 = neighbors[j];

                    // If neighbor1 is connected to neighbor2, it forms a triangular path
                    answer += graph[neighbor1].count(neighbor2);
                }
            }
        }

        // Since each triangular path is counted three times (once for each vertex),
        // we divide by 3 to get the correct number of unique paths.
        return answer / 3;
    }
};
```

Typescript Solution

```
// Interface representing a pair of integers
interface Pair {
    first: number;
    second: number;
}

// Function to calculate the number of triangular paths
function numberOfPaths(n: number, corridors: Pair[]): number {
    // Initialize an adjacency list for the graph
    const graph: Set<number>[] = new Array(n + 1);

    // Fill the graph array with empty sets for each node
    for (let i = 0; i <= n; ++i) {
        graph[i] = new Set();
    }

    // Populate the graph with corridors data
    for (const corridor of corridors) {
        // Extracting endpoints of the corridors
        const node1 = corridor.first;
        const node2 = corridor.second;

        // Add each node to the other's adjacency list
        graph[node1].add(node2);
        graph[node2].add(node1);
    }

    // Variable to store the number of triangular paths found
    let answer = 0;

    // Check each node for triangular paths
    for (let currentNode = 1; currentNode <= n; currentNode++) {
        // Create an array of neighbors from the current node's adjacency list
        const neighbors: number[] = Array.from(graph[currentNode]);

        // Iterate over pairs of neighbors to check for direct connections
        for (let i = 0; i < neighbors.length; ++i) {
            for (let j = i + 1; j < neighbors.length; ++j) {
                const neighbor1 = neighbors[i];
                const neighbor2 = neighbors[j];

                // If neighbor1 is directly connected to neighbor2, a triangular path is formed
                if (graph[neighbor1].has(neighbor2)) {
                    answer++;
                }
            }
        }
    }

    // Every triangle path has been counted 3 times, divide by 3 to normalize
    return answer / 3;
}

// Example usage
// Define some corridors as per the interface Pair
const corridors: Pair[] = [
    { first: 1, second: 2 },
    { first: 1, second: 3 },
    { first: 2, second: 3 }
];

// Call our function with a node and the corridors array
const triangularPaths = numberOfPaths(3, corridors);
console.log(triangularPaths); // Output should be 1 as there is one triangle path (1 - 2 - 3)
```

Time and Space Complexity

Time Complexity:

The given code consists of building a graph and then finding all the triangles in it. Here's a breakdown of the time complexity:

1. Building the adjacency graph `g` has a time complexity of $O(E)$, where E is the number of edges or corridors because we iterate over each corridor to build the graph.
2. The outer loop runs n times (where n is the number of rooms), so it has a time complexity of $O(N)$.
3. Inside the outer loop, the `combinations` function is called. Each call of `combinations` can generate up to $O(d^2)$ combinations, where d is the degree (number of edges) of node i. The degree can vary for each node, and in the worst case, it could be $n-1$, which would result in $O((n-1)^2)$ combinations for that node.
4. Checking if `j` is in `g[k]` is $O(1)$ with the hash set data structure, and this is done once for each combination generated in the previous step. So this operation can be potentially $O(n^2)$ in the worst case for each node.
5. Finally, the `ans` is divided by 3 outside of the loops, which is a constant-time operation $O(1)$.

Taking all these into account, the total time complexity is $O(N * E > N * (n-1)^2)$, which simplifies to $O(N * (n-1)^2)$ in the worst-case scenario when the graph is dense (n dominates E). In other words, the time complexity can be expressed as $O(N^3)$ when each node is connected to every other node.

Space Complexity:

The space complexity of the code is mainly due to the storage required for the adjacency graph.

1. The adjacency graph `g` will have a space complexity of $O(V + E)$, where V is the number of nodes and E is the number of edges, to store all vertices and their edges.
2. There is some additional overhead due to the storage of combinations in the inner loop, but this does not increase space complexity as it is temporary and does not depend on the size of the input.
3. The space complexity of other variables used (like `ans`, `i`, `j`, `k`) is $O(1)$.

The dominant term in the space complexity is the storage for the graph, which gives us $O(V + E)$ space complexity.

Putting the time and space complexities together, we have:

- Time Complexity: $O(N^3)$
- Space Complexity: $O(V + E)$