2242. Maximum Score of a Node Sequence

Hard Graph Array Enumeration Sorting Leetcode Link

Problem Description

In this LeetCode problem, we are given an undirected graph with `n` nodes, where each node has an associated score. The task is to find the maximum possible score of any valid node sequence with exactly 4 nodes. A valid node sequence must meet two criteria: first, each pair of consecutive nodes in the sequence must be connected by an edge, and second, no node can be repeated within the sequence. The score of the sequence is simply the sum of its nodes' scores. If there is no such sequence, we need to return `-1`.

An undirected graph means that if there is an edge connecting nodes `u` and `v`, we can travel from `u` to `v` and from `v` to `u` similarly. The node ids range from `0` to `n - 1`. The array `scores` represents the score of each node, while the array `edges` contains pairs of nodes that are connected by an edge. The challenge is to navigate the graph and find the optimal sequence of four nodes that maximizes the score.

Intuition

The intuition behind the solution is to consider node sequences as potential paths of length four. Since we want to maximize the score of such a sequence, we need to think strategically about which nodes we select. Considering every possible path of length four in the graph would be computationally expensive, especially if the graph is large. To make the problem tractable, the solution takes advantage of the following observations:

1. For any two connected nodes `a` and `b`, they might be the middle part of the sequence (second and third nodes), and we want to find the best possible candidates to place as the first and fourth nodes. These are the nodes connected to `a` and `b`, respectively, with the highest scores.

2. We only need to consider the three highest-scoring neighbors for each node when trying to form a sequence that includes that node because adding a lower-scoring neighbor would never produce the maximum sum.

With these insights, the solution first constructs a graph representation where each node is mapped to its three highest-scoring neighbors. Then, it iterates over each edge in the original graph, trying to construct sequences by checking all combinations of `a`'s top scoring neighbors with `b`'s top scoring neighbors that don't already include `a` or `b`. This approach vastly reduces the number of potential combinations to consider.

The maximum score found during this process is then returned. If no valid sequence of length four is found, we return `-1`.

Solution Approach

The implementation of the solution follows the intuition previously discussed and employs different data structures and algorithms to achieve the optimal path search. Here's a step-wise breakdown of the solution approach:

1. **Graph Representation**:
   - The algorithm starts by creating a default dictionary `g` that will hold the list of neighboring nodes for each node. In Python, `defaultdict(list)` from the `collections` module ensures that each new key starts with an empty list as its value.
   - It then iterates through the provided `edges` list to build the undirected graph by adding each node of an edge to the neighbor list of the other. The graph representation is adjacency lists, essentially a mapping from each node to its neighbors.
2. **Filtering Top Neighbors**:
   - To reduce the complexity for later steps, the algorithm then filters the neighbors of each node. For each node key in `g`, we keep only the top three neighbors with the highest scores. This is done using the `nlargest` function from the `heapq` module, which selects the largest `n` elements according to the provided key function. In this case, `lambda x: scores[x]` is the key function, which ranks neighbors based on their respective scores in the `scores` array.
3. **Finding the Optimal Sequence**:
   - Now, the algorithm goes through each edge in the original `edges` list, treating the pair of nodes on the edge `(a, b)` as the potential middle of the sequence.
   - For every such pair, we look at all combinations of the neighbors of `a` (denoted as `c`) and the neighbors of `b` (denoted as `d`). Here, we apply a crucial constraint: we only consider `c` and `d` that are distinct from both `a` and `b` and each other (`b != c != d != a`). This ensures we are only creating valid four-node sequences without repeats.
   - For every valid combination, we calculate the total score `t` of the sequence by summing the scores of `a`, `b`, `c`, and `d`. We keep track of the maximum score found so far in the variable `ans`.
   - If no sequence is found, `ans` retains its initial value of `-1`, otherwise, `ans` contains the maximum score of all valid sequences.

The key algorithms and data structures used in this solution are:

- **Graph representation using adjacency lists**: to efficiently store and explore neighbor nodes.
- **Heap-based selection**: using `nlargest` to efficiently sort and filter the neighbors based on their scores.
- **Greedy approach**: by trying to extend each edge with the highest scoring neighbors, we apply a greedy strategy aimed at maximizing the node sequence score.
- **Exhaustive search**: within the constraints of the greedy filtering, we still need to exhaustively search through the limited number of potential neighbors for the optimal sequence.

This combination of strategies is designed to find the maximum scoring node sequence of length four with a balance between performance and ensuring that no potential solution is missed.

Example Walkthrough

Let's walk through the solution approach with a simple example.

Assume the input graph is represented by:

- `n = 5` (so we have nodes 0 to 4)
- `scores = [20, 10, 30, 40, 25]`
- `edges = [(0, 1), (1, 2), (2, 3), (3, 4)]`

This graph looks like a straight line: 0 -- 1 -- 2 -- 3 -- 4

Step 1: Graph Representation

Create a default dictionary `g` to hold the adjacency list of the graph. It will end up looking like this:

- `g[0] = [1]`
- `g[1] = [0, 2]`
- `g[2] = [1, 3]`
- `g[3] = [2, 4]`
- `g[4] = [3]`

Step 2: Filtering Top Neighbors

For each node, we will determine the top 3 scoring neighbors (although in our simple graph, no node has more than two neighbors):

- `g[0] = [1]` (Only has one neighbor)
- `g[1] = [2]` (Highest-scoring neighbor of node 1 is node 2)
- `g[2] = [3]` (Highest-scoring neighbor of node 2 is node 3)
- `g[3] = [4]` (Highest-scoring neighbor of node 3 is node 4)
- `g[4] = [3]` (Only has one neighbor)

Step 3: Finding the Optimal Sequence

Now, we'll iterate through the `edges` to find sequences of 4 nodes:

- For edge `(0, 1)`, our potential sequence is Node 1's high scoring neighbor (Node 2) and one of Node 0's neighbors (Only Node 1). But for a valid sequence, we need 4 unique nodes, and we can't create one because the graph is a straight line and node 1 is already in use.
- For edge `(1, 2)`, we can choose Node 1's highest-scoring neighbor, which is Node 0 and Node 2's highest-scoring neighbor, which is Node 3. This results in a potential sequence of `[0, 1, 2, 3]`.
- For edge `(2, 3)`, we can take Node 2's highest-scoring neighbor, Node 1, and Node 3's highest-scoring neighbor, Node 4. This leads to a potential sequence of `[1, 2, 3, 4]`.

The total scores for the potential sequences are:

- Total score for `[0, 1, 2, 3]` = `20 + 10 + 30 + 40 = 100`
- Total score for `[1, 2, 3, 4]` = `10 + 30 + 40 + 25 = 105`

Since we are looking for the maximum score, the best sequence is `[1, 2, 3, 4]` with a total score of `105`.

Hence, the algorithm will return `105` as the maximum score of any valid node sequence with exactly 4 nodes for the given graph. If we could not find any valid sequence, we would return `-1`, but in this case, we successfully found one.

Python Solution

```python
from collections import defaultdict
from heapq import nlargest

class Solution:
    def maximumScore(self, scores: List[int], edges: List[List[int]]) -> int:
        # Create a graph represented as an adjacency list
        graph = defaultdict(list)

        # Populate the graph with edges
        for node_a, node_b in edges:
            graph[node_a].append(node_b)
            graph[node_b].append(node_a)

        # Keep only the top 3 adjacent nodes with the highest scores for each node
        for node in graph:
            graph[node] = nlargest(3, graph[node], key=lambda x: scores[x])

        # Initialize the maximum score to -1 (as a default for an impossible score)
        max_score = -1

        # Iterate through all edges to calculate possible scores
        for node_a, node_b in edges:
            # Check possible combinations with the top scoring adjacent nodes
            for adjacent_from_a in graph[node_a]:
                for adjacent_from_b in graph[node_b]:
                    # Ensure all nodes in the quadruple are distinct
                    if node_b != adjacent_from_a != adjacent_from_b != node_a:
                        # Calculate the total score of this quadruple
                        total_score = scores[node_a] + scores[node_b] + scores[adjacent_from_a] + scores[adjacent_from_b]
                        # Update the max score if the current total is greater
                        max_score = max(max_score, total_score)

        # Return the highest score found (or -1 if no such quadruple is possible)
        return max_score

```

Java Solution

```java
class Solution {

    // Calculates the maximum score from a given set of scores and edges.
    public int maximumScore(int[] scores, int[][] edges) {
        // Number of vertices in the graph
        int numVertices = scores.length;

        // Create a graph using adjacency lists, where each list will only contain the top 3 highest scoring vertices
        List<Integer>[] graph = new List[numVertices];
        Arrays.setAll(graph, v -> new ArrayList<>());

        // Build the adjacency list for each vertex
        for (int[] edge : edges) {
            int vertex1 = edge[0], vertex2 = edge[1];
            graph[vertex1].add(vertex2);
            graph[vertex2].add(vertex1);
        }

        // Sort and truncate each adjacency list to top 3 vertices based on scores
        for (int i = 0; i < numVertices; ++i) {
            graph[i].sort((vertexA, vertexB) -> scores[vertexB] - scores[vertexA]);
            graph[i] = graph[i].subList(0, Math.min(3, graph[i].size()));
        }

        // Initialize the answer to -1, which means it's not possible to find a valid quadruple
        int maxScore = -1;

        // Loop through each edge to find the highest score by considering quadruples
        for (int[] edge : edges) {
            int vertexA = edge[0], vertexB = edge[1];
            // Consider all pairs (vertexC, vertexD) where vertexC is connected to vertexA
            // and vertexD is connected to vertexB, ensuring they are all different vertices.
            for (int vertexC : graph[vertexA]) {
                for (int vertexD : graph[vertexB]) {
                    if (vertexC != vertexB && vertexC != vertexD && vertexA != vertexD) {
                        int currentScore = scores[vertexA] + scores[vertexB] + scores[vertexC] + scores[vertexD];
                        maxScore = Math.max(maxScore, currentScore);
                    }
                }
            }
        }

        // Return the maximum score found, or -1 if no such quadruple exists
        return maxScore;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>
#include <numeric>

class Solution {
public:
    // Calculates the maximum score from a given set of scores and edges.
    int maximumScore(std::vector<int>& scores, std::vector<std::vector<int>>& edges) {
        // Number of vertices in the graph
        int numVertices = scores.size();

        // Graph representation using adjacency lists, keeping only the top 3 vertices sorted by their scores
        std::vector<std::vector<int>> graph(numVertices);

        // Build the adjacency list for each vertex from the edges
        for (const auto& edge : edges) {
            int vertex1 = edge[0], vertex2 = edge[1];
            graph[vertex1].push_back(vertex2);
            graph[vertex2].push_back(vertex1);
        }

        // Sort and keep only the top 3 vertices with highest scores in each adjacency list
        for (int i = 0; i < numVertices; ++i) {
            std::sort(graph[i].begin(), graph[i].end(), [&](int vertexA, int vertexB) {
                return scores[vertexB] - scores[vertexA];
            });

            if (graph[i].size() > 3) {
                graph[i].resize(3);
            }
        }

        // Initialize the maximum score to -1 which signifies that it is not possible to find a valid quadruple
        int maxScore = -1;

        // Check all quadruples formed with the edges to calculate the maximum score
        for (const auto& edge : edges) {
            int vertexA = edge[0], vertexB = edge[1];
            // Consider all pairs (vertexC, vertexD) where vertexC is connected to vertexA
            // and vertexD is connected to vertexB, making sure they are all distinct.
            for (int vertexC : graph[vertexA]) {
                if (vertexC == vertexB) continue;
                for (int vertexD : graph[vertexB]) {
                    if (vertexD == vertexA || vertexD == vertexC) continue;
                    int currentScore = scores[vertexA] + scores[vertexB] + scores[vertexC] + scores[vertexD];
                    maxScore = std::max(maxScore, currentScore);
                }
            }
        }

        // Return the maximum score found, or -1 if no valid quadruple exists.
        return maxScore;
    }
};

```

Typescript Solution

```typescript
// Our function that calculates the maximum score
function maximumScore(scores: number[], edges: number[][]): number {
    // Number of vertices in the graph
    const numVertices: number = scores.length;

    // Create a graph using adjacency lists, where each list will only contain
    // the top 3 highest scoring vertices connected to it
    const graph: number[][] = new Array(numVertices).fill(0).map(() => []);

    // Build the adjacency list for each vertex
    for (const edge of edges) {
        const [vertex1, vertex2] = edge;
        graph[vertex1].push(vertex2);
        graph[vertex2].push(vertex1);
    }

    // Sort and truncate each adjacency list to top 3 vertices based on scores
    graph.forEach((adjList, index) => {
        adjList.sort((vertexA, vertexB) => scores[vertexB] - scores[vertexA]);
        graph[index] = graph[index].slice(0, Math.min(3, graph[index].length));
    });

    // Initialize the answer to -1, which means it's not possible to find a
    // valid quadruple with the current set of edges
    let maxScore: number = -1;

    // Loop through each edge to find the highest score by considering quadruples
    for (const edge of edges) {
        const [vertexA, vertexB] = edge;
        // Consider all pairs (vertexC, vertexD) where vertexC is connected to vertexA
        // and vertexD is connected to vertexB, ensuring they are all unique vertices.
        for (const vertexC of graph[vertexA]) {
            for (const vertexD of graph[vertexB]) {
                if (vertexC !== vertexB && vertexC !== vertexD && vertexA !== vertexD) {
                    const currentScore: number = scores[vertexA] + scores[vertexB] + scores[vertexC] + scores[vertexD];
                    maxScore = Math.max(maxScore, currentScore);
                }
            }
        }
    }

    // Return the maximum score found, or -1 if no such quadruple exists
    return maxScore;
}

```

Time and Space Complexity

Time Complexity

The provided code has several components to consider in terms of time complexity:

1. Building the graph `g`: The first for loop goes through each edge in the `edges` list, which runs O(E) times, where E is the number of edges given.

2. Trimming the adjacency lists to the top 3 scores: For each node, the `nlargest` function is called, which takes O(m log n) time where m is the total number of elements to sort (in this case, the number of adjacent nodes) and n is the number of elements to find (3 in this case). Since we perform this operation for all nodes `k`, and in the worst case, this could be for all nodes 'V', this step can be bounded by O(V * m log n), where V is the number of vertices. Since m can also be bounded by E in the worst case (if all edges are connected to one vertex), this step simplifies to O(E log 3), and since finding the largest 3 elements can be done in constant time, it simplifies to O(E).

3. The nested loops to calculate the maximum score: This is the most computationally expensive part of the code. There is a nested four-loop structure here. However, due to the previous trimming of the adjacency lists, the third and fourth loops run at most 3 times each since they loop over the top 3 scores for vertices a and b, respectively. Thus, these loops contribute a constant factor, and the time complexity comes down to the double loop over the edges. Hence, the time complexity for this part is O(E).

Putting all of this together, the total time complexity is O(E + E log 3 + E) which can be simplified to O(E) because E is the dominant term and log 3 is a constant, and also because O(E log 3) simplifies to O(E) when log 3 is constant.

Space Complexity

The space complexity of the provided code is determined by:

1. The graph `g` which holds an adjacency list for each vertex. As we store only the 3 largest scores for each vertex, the space complexity is not directly proportional to the number of edges. Instead, it can be limited to O(V) where V is the number of vertices, as each vertex holds a list of at most 3 other vertices.

2. The temp variables `t` and `ans` occupy O(1) space.

Hence, the total space complexity is O(V).