2497. Maximum Star Sum of a Graph

Medium Graph Sorting Array Sorting Heap (Priority Queue)

Leetcode Link

Problem Description

The problem describes an undirected graph with n nodes numbered from 0 to $n-1$. Each node has an associated value defined in the array `vals`, where `vals[i]` represents the value of the ith node. Additionally, we are given a list of undirected edges defined in the array `edges`. An edge connects two nodes, implying that the nodes are directly reachable from each other.

A *star graph* is a specific type of subgraph in which one central node is connected to some or all other nodes through edges, but those other nodes are not connected to each other. The star sum is the combined value of the nodes in this subgraph, including the central node and its neighbors.

The task is to find the maximum star sum possible by including at most k edges in the star graph. In other words, what is the highest sum of node values we can achieve when we select a subset of the graph that forms a star graph, with the central node having up to k neighbors.

Intuition

To find the maximum star sum, we should aim to include the nodes with the highest values adjacent to our chosen central node. Since the edges are undirected, we need to consider each node as a potential center of the star graph and calculate the maximum sum obtainable with it as the center.

The given solution follows these steps:

1. Create an adjacency list, g, using `defaultdict(list)` from Python's collections module, to store the nodes (a) and their corresponding connected nodes' values (`vals[b]`) if their value is greater than zero.
2. For all edges `(a, b)`, do the following:
   - If `vals[b]` is positive, append it to $g[a]$ (connected nodes' values for a).
   - Conversely, if `vals[a]` is positive, append it to $g[b]$.
3. Sort the lists of connected nodes' values in descending order for each node in g. It ensures that the highest valued neighbor is considered first.
4. Calculate the `star sum` for each node as the sum of its value, v, and the sum of at most k highest valued neighbors from the respective list in g. This is accomplished by iterating through all nodes enumerated along with their values (`enumerate(vals)`) and computing the sum of a node's value and the total of its k highest connected nodes' values.
5. Finally, the `maxStarSum` function returns the maximum `star sum` obtained from all possible center nodes.

The key to the solution is ensuring that the highest valued neighbors are included in the star graph to maximize the star sum, constrained by the limit of at most k edges.

Solution Approach

The solution implements a simple but clever approach to maximize the star sum by leveraging a few common data structures and algorithms, as follows:

Data Structures Used:

1. **Adjacency List**: A defaultdict of lists is used to create an adjacency list g. It stores the values of connected nodes for each node (excluding values that are zero, as they do not contribute to the sum).
2. **List**: Python lists are used to store connected nodes' values and to sort these values in descending order.

Algorithm Steps:

1. **Creating Adjacency List (Undirected)**: The solution loops through each edge `(a, b)` in `edges`:
   - For node a, if `vals[b] > 0`, it appends this value to the list $g[a]$.
   - For node b, similarly, if `vals[a] > 0`, it appends this value to the list $g[b]$.

   This is because the graph is undirected, meaning either a or b can be the center of a star graph with the other being its neighbor.
2. **Sorting Neighbors' Values**: For each node's list of connected nodes in g, the values are sorted in reverse (descending) order. It ensures that when we choose up to k neighbors, we're choosing the ones that maximize our sum.
3. **Calculating Star Sum**: With the sorted lists for each potential center node, the star sum is calculated by taking the value of the current node v and adding the sum of the k highest neighboring node values from its list in g. This is done using a list comprehension:

```
s = sum(g[i][:k], v)
```

   - `g[i][:k]` fetches at most the k largest values from node i's list of neighbors' values.
   - `sum(g[i][:k], v)` calculates the sum of these up to k values.
   - `v + sum(g[i][:k])` adds the node i's value to this sum to get the star sum for this particular node i as the center.
   - `max(...)` then finds the maximum star sum achievable from all potential center nodes.

Complexity Analysis:

- **Time Complexity**: $O(V + E \log E)$ where V is the number of nodes, and E is the number of edges. This is because E edges are iterated to build the adjacency list, and each neighbors list could potentially be sorted up to E times (in the worst case, where all edges are connected to a single node). Therefore, the bottleneck is the sorting operation.
- **Space Complexity**: $O(E)$, as additional space is required to store neighbors' values for each node, and in the worst case, each node could hold values related to all other nodes.

Patterns Used:

- **Greedy Algorithm**: By sorting the neighbors' values in descending order and picking the top k elements, the solution uses a greedy approach where it always picks the local optimal choice (highest value) at each step to maximize the star sum in the end.

The solution effectively combines these data structures and algorithms to efficiently solve the maximum star sum problem while respecting the constraint of at most k edges.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have a graph with 5 nodes ($n=5$), numbered from 0 to 4. The values associated with each node are given by the array `vals = [1, 2, 3, 4, 5]`, and we are allowed to include at most $k=2$ edges in our star graph to maximize the star sum. The edges are given by the array `edges = [[0, 1], [0, 2], [0, 3], [1, 4], [2, 4]]`.

Step-by-Step Process:

1. **Creating Adjacency List (Undirected)**: Our adjacency list g would be built as follows by iterating over the edges:

```
1. g[0] -> [2, 3, 4]    # Node 0 is connected to nodes 1, 2, 3 with positive values
2. g[1] -> [2, 5]       # Node 1 is connected to nodes 0 and 4
3. g[2] -> [3, 5]       # Node 2 is connected to nodes 0 and 4
4. g[3] -> [1]          # Node 3 is connected to node 0
5. g[4] -> [2, 3]       # Node 4 is connected to nodes 1 and 2
```

2. **Sorting Neighbors' Values**: The next step is to sort each list of connected nodes' values in descending order:

```
1. g[0] -> [4, 3, 2]    # Sort [2, 3, 4 in descending order
2. g[1] -> [5, 2]       # Sort [ 5, 2 in descending order
3. g[2] -> [5, 3]       # Sort [ 5, 3 in descending order
4. g[3] -> [1]          # Already sorted since there's just one element
5. g[4] -> [3, 2]       # Sort [ 3, 2 in descending order
```

3. **Calculating Star Sum**: Now, for each node, we will calculate the star sum by taking its value and adding the sum of its k highest valued neighbors:

```
1. For node 0: Star sum = 1 + (4+3) = 8 (Take the 2 highest values 4 and 3)
2. For node 1: Star sum = 2 + (5) = 7 (Only 1 neighbor's value can be taken since k=2 and node 1's value has to be included)
3. For node 2: Star sum = 3 + (5) = 8
4. For node 3: Star sum = 4 + (1) = 5
5. For node 4: Star sum = 5 + (3) = 8 (Same reasoning as node 1)
```

Note that we only include up to k neighbor's values, which is why even though node 0 has three neighbors, only the highest two (4 and 3) are included in the sum.

4. **Finding the Maximum Star Sum**: The final step is to find the maximum star sum possible:

```
Maximum Star Sum = max(8, 7, 8, 5, 8) = 8
```

And so, the solution would return 8 as the maximum star sum possible by selecting a subset of the graph that forms a star graph with at most k edges. This example clearly shows how the solution approach effectively uses a greedy method to maximize the star sum.

Python Solution

```python
from collections import defaultdict
from typing import List

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

        # Iterate over each edge and build an adjacency list for nodes with positive values
        for node_a, node_b in edges:
            if values[node_b] > 0:
                graph[node_a].append(values[node_b])
            if values[node_a] > 0:
                graph[node_b].append(values[node_a])

        # Sort the adjacency lists in descending order to prioritize larger values
        for neighbors in graph.values():
            neighbors.sort(reverse=True)

        # Calculate the maximum star sum by adding the node's value to the
        # sum of up to k highest values among its adjacent nodes
        max_star_sum = max(value + sum(graph.get(node_index)[:k]) for node_index, value in enumerate(values))

        # Return the maximum star sum found
        return max_star_sum
```

Java Solution

```java
class Solution {
    // Function to calculate the maximum star sum
    public int maxStarSum(int[] values, int[][] edges, int k) {
        int numNodes = values.length;

        // Create a list of lists to represent the graph
        List<Integer>[] adjacencyLists = new
        Arrays.setAll(graph, x -> new ArrayList<>());

        // Building an adjacency list for each node containing values
        for (int[] edge : edges) {
            int firstNode = edge[0], secondNode = edge[1];

            // Only add the positive values to the adjacency list
            if (values[secondNode] > 0) {
                graph[firstNode].add(values[secondNode]);
            }
            if (values[firstNode] > 0) {
                graph[secondNode].add(values[firstNode]);
            }
        }

        // Sort the adjacency lists in descending order
        for (List<Integer> adjacentValues : graph) {
            Collections.sort(adjacentValues, (a, b) -> b - a);
        }

        // Initialize the answer with the lowest possible value
        int maxStarSum = Integer.MIN_VALUE;

        // Iterate through each node to calculate the sum of the stars
        for (int i = 0; i < numNodes; ++i) {
            int maxValSum = values[i];

            // Add up the k highest values connected to the node
            for (int j = 0; j < Math.min(graph[i].size(), k); ++j) {
                maxValSum += graph[i].get(j);
            }

            // Update the answer with the maximum sum found so far
            maxStarSum = Math.max(maxStarSum, maxValSum);
        }

        // Return the maximum star sum
        return maxStarSum;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Compute the maximum sum of star values with the center at each vertex considering 'k' arms
    int maxStarSum(vector<int>& values, vector<vector<int>>& edges, int k) {
        int numNodes = values.size(); // Number of vertices in the graph

        // Adjacency list to hold the positive values of connected vertices
        vector<vector<int>> adjacencyLists(numNodes);

        // Build the adjacency list with only positive values from connected vertices
        for (auto& edge : edges) {
            int vertex0 = edge[0], vertex1 = edge[1];
            if (values[vertex0] > 0) adjacencyLists[vertex1].emplace_back(values[vertex0]);
            if (values[vertex1] > 0) adjacencyLists[vertex0].emplace_back(values[vertex1]);
        }

        // Sort the adjacency list in descending order to prioritize larger values
        for (auto& neighbors : adjacencyLists) {
            sort(neighbors.begin(), neighbors.end());
        }

        // Variable to store the maximum star value found
        int maxStarValue = INT_MIN;

        // Iterate over each vertex to calculate the star value
        for (int i = 0; i < numNodes; ++i) {
            int starValue = values[i]; // Start with the value of the current vertex

            // Add up to 'k' highest positive values from connected vertices
            for (int j = 0; j < min(adjacencyLists[i].size(), k); ++j) {
                starValue += adjacencyLists[i][j];
            }

            // Update the maximum star value if the current one is higher
            maxStarValue = max(maxStarValue, starValue);
        }

        return maxStarValue; // Return the found maximum star value
    }
};
```

Typescript Solution

```typescript
// Define a VertexValue array type for clarity
type VertexValue[] = number[];

// Define an Edge[] type for clarity
type Edge[] = number[][];

// Function to compute the maximum sum of star values with the center at each vertex considering 'k' arms
function maxStarSum(values: VertexValue[], edges: Edge[][], k: number): number {
    const numNodes: number = values.length; // Number of vertices in the graph

    // Adjacency list to hold the positive values of connected vertices
    let adjacencyLists: VertexValue[][] = new Array(numNodes).fill(0).map(() => []);

    // Build the adjacency list with only positive values from connected vertices
    for (let edge of edges) {
        let [vertex0, vertex1] = edge[0], edge[1];
        if (values[vertex0] > 0) adjacencyLists[vertex1].push(values[vertex0]);
        if (values[vertex1] > 0) adjacencyLists[vertex0].push(values[vertex1]);
    }

    // Sort the adjacency list in descending order to prioritize larger values
    for (let neighbors of adjacencyLists) {
        neighbors.sort((a, b) => b - a);
    }

    // Variable to store the maximum star value found
    let maxStarValue: number = Number.MIN_SAFE_INTEGER;

    // Iterate over each vertex to calculate the star value
    for (let i = 0; i < numNodes; ++i) {
        let starValue: number = values[i]; // Start with the value of the current vertex

        // Add up to 'k' highest positive values from connected vertices
        for (let j = 0; j < Math.min(adjacencyLists[i].length, k); ++j) {
            starValue += adjacencyLists[i][j];
        }

        // Update the maximum star value if the current one is higher
        maxStarValue = Math.max(maxStarValue, starValue);
    }

    return maxStarValue; // Return the found maximum star value
}
```

Time and Space Complexity

Time Complexity

The time complexity of the code is associated with several operations performed:

1. **Construction of the graph g**: The `for` loop that iterates over the `edges` has a time complexity of $O(E)$ where E is the number of edges because each edge is visited once.
2. **Sorting the adjacency lists**: Each adjacency list in graph g is sorted in reverse order. In the worst case, if all nodes are connected to all other nodes, it will take $O(N \log N)$ time per node to sort, where N is the number of nodes. However, typically, the number of edges connected to a node (degree of a node) is much lower than N, so it will more often be $O(D \log D)$ where D is the average degree of a node.
3. **Calculating the max star sum:** The maximum star sum is computed in a single for loop that iterates over the nodes once. The complexity for this part is primarily from the summation operation which takes $O(k)$ time for each node, as it sums up to k elements from the sorted adjacency list. Since it iterates over all N nodes, this results in a $O(Nk)$ time complexity.

The overall worst-case time complexity can be approximated by adding these components together: $O(E + N \log N + Nk)$.

However, the sorting step's time complexity will often be dominated by the number of edges (sorting smaller adjacency lists): $O(E + D \log D + Nk)$.

Space Complexity

The space complexity involves:

1. **Storing the graph g**: The graph is stored using adjacency lists, which will require $O(N + E)$ space: $O(N)$ for storing N keys and $O(E)$ for storing the list of neighbor values.
2. **Auxiliary space for sorting**: Sorting the adjacency lists will require additional space which depends on the sorting algorithm used by Python's `.sort()` method (Timsort), but since in-place sorting is used, the impact on space complexity should be minimal.
3. **Storing sums:** The space for storing the sums is not significant as it only stores the sums for each node which is $O(1)$ per node resulting in $O(N)$ in total.

The overall space complexity is $O(N + E)$.