2497. Maximum Star Sum of a Graph Medium Greedy Sorting Heap (Priority Queue) Graph Array

## Leetcode Link

# The problem describes an undirected graph with n nodes numbered from 0 to n-1. Each node has an associated value defined in the

Problem Description

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

k neighbors. Intuition

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

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:

To find the maximum star sum, we should aim to include the nodes with the highest values adjacent to our chosen central node.

Conversely, if vals[a] is positive, append it to g[b].

2. For all edges (a, b), do the following: If vals[b] is positive, append it to g[a] (connected nodes' values for a).

3. Sort the lists of connected nodes' values in descending order for each node in g. It ensures that the highest valued neighbor is

1. Create an adjacency list, g, using defaultdict(list) from Python's collections module, to store the nodes (a) and their

considered first.

corresponding connected nodes' values (vals[b]) if their value is greater than zero.

- 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))
- 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

The key to the solution is ensuring that the highest valued neighbors are included in the star graph to maximize the star sum,

Data Structures Used:

1. Creating Adjacency List (Undirected): The solution loops through each edge (a, b) in edges:

o g[i] [:k] fetches at most the k largest values from node i's list of neighbors' values.

• max(...) then finds the maximum star sum achievable from all potential center nodes.

edges are connected to a single node). Therefore, the bottleneck is the sorting operation.

For node a, if vals[b] > 0, it appends this value to the list g[a].

sum(g[i][:k]) calculates the sum of these up to k values.

node could hold values related to all other nodes.

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

1 For node 0: Star sum = 1 + (4+3) = 8 (Take the 2 highest values 4 and 3)

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

def max\_star\_sum(self, values: List[int], edges: List[List[int]], k: int) -> int:

# Sort the adjacency lists in descending order to prioritize larger values

# Calculate the maximum star sum by adding the node's value to the

# Iterate over each edge and build an adjacency list for nodes with positive values

# Create a graph represented as an adjacency list

graph[node\_a].append(values[node\_b])

graph[node\_b].append(values[node\_a])

# sum of up to k highest values among its adjacent nodes

public int maxStarSum(int[] values, int[][] edges, int k) {

// Building an adjacency list for each node containing values

// Only add the positive values to the adjacency list

// Create a list of lists to represent the graph

List<Integer>[] graph = new List[numValues];

int from = edge[0], to = edge[1];

graph[from].add(values[to]);

graph[to].add(values[from]);

Arrays.setAll(graph, x -> new ArrayList<>());

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.

(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.

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

### 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.

comprehension:

Algorithm Steps:

algorithms, as follows:

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

1 return max(v + sum(g[i][:k]) for i, v in enumerate(vals))

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.

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

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

Space Complexity: O(E), as additional space is required to store neighbors' values for each node, and in the worst case, each

• Greedy Algorithm: By sorting the neighbors' values in descending order and picking the top k elements, the solution uses a

The solution effectively combines these data structures and algorithms to efficiently solve the maximum star sum problem while

greedy approach where it always picks the local optimal choice (highest value) at each step to maximize the star sum in the end.

Example Walkthrough

Step-by-Step Process:

respecting the constraint of at most k edges.

**Patterns Used:** 

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] -> [1, 5] # Node 1 is connected to nodes 0 and 4 3 g[2] -> [1, 5] # Node 2 is connected to nodes 0 and 4

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)].

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

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)

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

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

```
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)
```

**Python Solution** 

class Solution:

sum.

10

12

14

15

16

17

18

19

20

21

22

23

24

25

26

9

10

11

12

13

14

15

16

17

18

19

20

(4 and 3) are included in the sum.

1 Maximum Star Sum =  $\max(8, 7, 8, 5, 8) = 8$ 

graph = defaultdict(list)

for node\_a, node\_b in edges:

if values[node\_b] > 0:

if values[node\_a] > 0:

for neighbors in graph.values():

neighbors.sort(reverse=True)

# Return the maximum star sum found

int numValues = values.length;

for (int[] edge : edges) {

**if** (values[to] > 0) {

if (values[from] > 0) {

return max\_star\_sum

valued neighbors:

1 from collections import defaultdict from typing import List

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

```
Java Solution
  class Solution {
      // Function to calculate the maximum star sum
```

max\_star\_sum = max(value + sum(graph[node\_index][:k]) for node\_index, value in enumerate(values))

```
22
23
           // Sort the adjacency lists in descending order
24
            for (List<Integer> adjacentValues : graph) {
25
                Collections.sort(adjacentValues, (a, b) -> b - a);
26
27
28
           // Initialize the answer with the lowest possible value
29
            int maxStarSum = Integer.MIN_VALUE;
30
           // Iterate through each node to calculate the sum of the stars
31
32
           for (int i = 0; i < numValues; ++i) {</pre>
33
                int nodeValue = values[i];
34
35
               // Add up the k highest values connected to the node
36
                for (int j = 0; j < Math.min(graph[i].size(), k); ++j) {</pre>
37
                    nodeValue += graph[i].get(j);
38
39
               // Update the answer with the maximum sum found so far
40
               maxStarSum = Math.max(maxStarSum, nodeValue);
41
42
43
44
           // Return the maximum star sum
45
           return maxStarSum;
46
47 }
48
C++ Solution
```

// Compute the maximum sum of star values with the center at each vertex considering 'k' arms

if (values[vertexB] > 0) adjacencyList[vertexA].emplace\_back(values[vertexB]);

if (values[vertexA] > 0) adjacencyList[vertexB].emplace\_back(values[vertexA]);

int maxStarSum(vector<int>& values, vector<vector<int>>& edges, int k) {

vector<vector<int>> adjacencyList(numVertices);

int vertexA = edge[0], vertexB = edge[1];

sort(neighbors.rbegin(), neighbors.rend());

// Variable to store the maximum star value found

starValue += adjacencyList[i][j];

maxStarValue = max(maxStarValue, starValue);

return maxStarValue; // Return the found maximum star value

// Iterate over each vertex to calculate the star value

for (auto& neighbors : adjacencyList) {

for (int i = 0; i < numVertices; ++i) {</pre>

for (auto& edge : edges) {

int maxStarValue = INT\_MIN;

1 // Define a VertexValue array type for clarity

for (let edge of edges) {

for (let neighbors of adjacencyList) {

neighbors.sort( $(a, b) \Rightarrow b - a$ );

for (let i = 0; i < numVertices; ++i) {</pre>

// Variable to store the maximum star value found

let maxStarValue: number = Number.MIN\_SAFE\_INTEGER;

// Iterate over each vertex to calculate the star value

int numVertices = values.size(); // Number of vertices in the graph

// Adjacency list to hold the positive values of connected vertices

// Build the adjacency list with only positive values from connected vertices

// Sort the adjacency list in descending order to prioritize larger values

int starValue = values[i]; // Start with the value of the current vertex

for (int j = 0; j < min(static\_cast<int>(adjacencyList[i].size()), k); ++j) {

// Add up to 'k' highest positive values from connected vertices

// Update the maximum star value if the current one is higher

function maxStarSum(values: VertexValueArray, edges: EdgeArray, k: number): number {

// Adjacency list to hold the positive values of connected vertices

let vertexA: number = edge[0], vertexB: number = edge[1];

const numVertices: number = values.length; // Number of vertices in the graph

// Build the adjacency list with only positive values from connected vertices

if (values[vertexB] > 0) adjacencyList[vertexA].push(values[vertexB]);

if (values[vertexA] > 0) adjacencyList[vertexB].push(values[vertexA]);

// Sort the adjacency list in descending order to prioritize larger values

let adjacencyList: VertexValueArray[] = new Array(numVertices).fill(0).map(() => []);

let starValue: number = values[i]; // Start with the value of the current vertex

### 2 type VertexValueArray = number[]; 4 // Define an EdgeArray type for clarity type EdgeArray = number[][]; // Function to compute the maximum sum of star values with the center at each vertex considering 'k' arms

Typescript Solution

1 class Solution {

2 public:

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

41

40 };

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

```
32
 33
            // Add up to 'k' highest positive values from connected vertices
            for (let j = 0; j < Math.min(adjacencyList[i].length, k); ++j) {</pre>
 34
 35
                starValue += adjacencyList[i][j];
 36
 37
 38
            // Update the maximum star value if the current one is higher
 39
            maxStarValue = Math.max(maxStarValue, starValue);
 40
 41
 42
         return maxStarValue; // Return the found maximum star value
 43
 44
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

3. Calculating the max star sum: The maximum star sum is computed in a single for loop that iterates over the nodes once. The

However, the sorting step's time complexity will often be dominated by the number of edges (sorting smaller adjacency lists): 0 (E +

1. Storing the graph g: The graph is stored using adjacency lists, which will require 0(N + E) space: 0(N) for storing N keys and

The space complexity involves:

D log D + Nk).

is the average degree of a node.

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:  $O(E + N \log N + Nk)$ .

- Space Complexity
  - 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 0(1) per node resulting in O(N) in total. The overall space complexity is O(N + E).