1697. Checking Existence of Edge Length Limited Paths

and v_i with distance dis_i . Note that there may be **multiple** edges between two nodes.

Given an array queries, where queries[j] = $[p_j, q_j, limit_j]$, your task is to determine for each queries[j] whether there is a path between pj and qj such that each edge on the path has a distance strictly less than $\lim_{i \to \infty} t_i$

An undirected graph of n nodes is defined by edgeList, where edgeList[i] = $[u_i, v_i, \operatorname{dis}_i]$ denotes an edge between nodes u_i

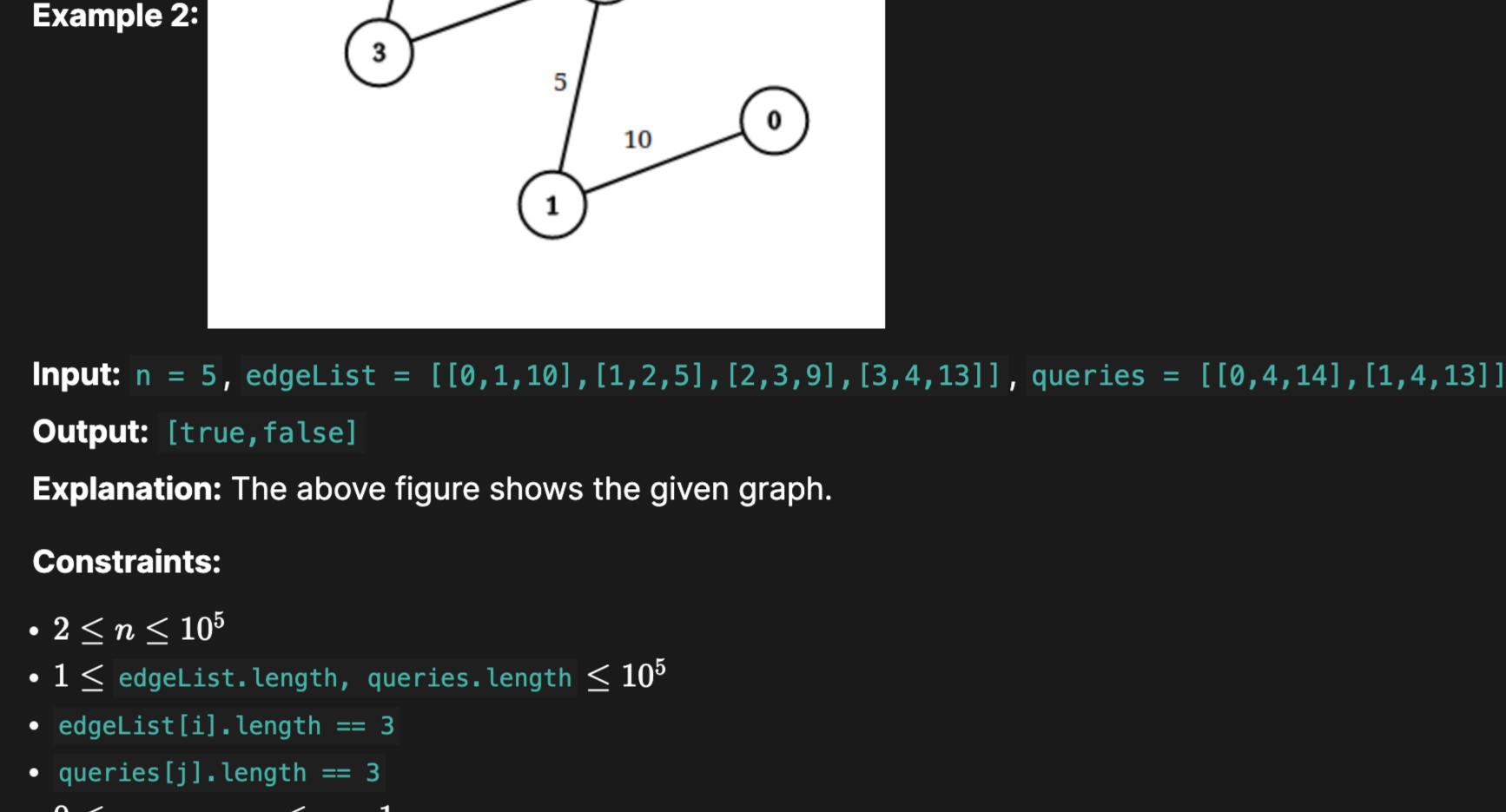
Return a **boolean array** answer, where answer length == queries length and the j^{th} value of answer is true if there is a path for queries[j], and false otherwise.

Example 1:

Input: n = 3, edgeList = [[0,1,2],[1,2,4],[2,0,8],[1,0,16]], queries = [[0,1,2],[0,2,5]] Output: [false, true] **Explanation:** The above figure shows the given graph. Note that there are two overlapping edges between 0 and 1 with distances 2 and 16.

For the first query, between 0 and 1 there is no path where each distance is less than 2, thus we return false for this query. For the second query, there is a path $(0 \to 1 \to 2)$ of two edges with distances less than 5, thus we return true for this query.

13



• $0 \leq u_i, v_i, p_j, q_i \leq n-1$

connected. An alternative is to build the graph and run a $\overline{ ext{BFS}}$ on it to check for connectivity from p_j to q_j .

Let M denote the size of edgeList and let Q denote the size of queries.

• $u_i \neq v_i$ • $p_j \neq q_j$

Brute Force

Solution

• $1 \leq \operatorname{dis}_i, \operatorname{limit}_i \leq 10^9$

• There may be multiple edges between two nodes.

When we see problems about whether or not two nodes are connected, we should think about DSU as it answers queries related

With the DSU solution, we'll answer each query in $\mathcal{O}(M\log N)$. This leads to a total time complexity of $\mathcal{O}(QM\log N)$. The BFS solution answers queries in $\mathcal{O}(N+M)$ in the worst case which leads to a total time complexity of $\mathcal{O}(Q(N+M))$.

Full Solution

queries.

for the remaining queries.

answers to the queries in the order they were asked.

because we rebuild the <u>DSU</u> for every query. Let's find a way to answer all the queries without rebuilding the <u>DSU</u> each time. We can accomplish this by answering the queries by **non-decreasing** $\lim_i t_i$. If we answer queries in this order, we'll only need to

We will focus on the brute force <u>DSU</u> solution and extend it to obtain the full solution. The reason why this solution is inefficient is

build the DSU once. Why is this the case? First, we'll take a step back to our brute force solution. Let's look at what edges each

<u>DSU</u> contains for each $limit_i$. Our queries are sorted by the value of $limit_i$ so $limit_{i+1}$ will always be greater or equal to $limit_i$.

Since $\mathrm{limit}_{j+1} \geq \mathrm{limit}_j$, we can observe that all edges in the DSU for query j will always be part of the DSU for query j+1. We

can also observe that this means once an edge is added into the DSU for some query, it will stay in the DSU for the remaining

to connectivity very quickly. For each query queries[j], we'll initialize a DSU and merge nodes connected by edges with a

distance **strictly less than** ${
m limit}_i$. We can then check if p_i and q_i have the same id/leader in the DSU to determine if they are

Example For example, let's say we had queries with the following limits: [1,3,5,7,9] If edgeList had an edge with distance 4, it will be added in the DSU for queries with limits of 5, 7, and 9 as $4 \le 5, 7, 9$. We can see that once we reach the query with the limit of 5, we will include the edge with distance 4 into our <u>DSU</u> and it will stay there

For each query, we should include another variable which is the index of that query. This is important as we have to return the

Since we are adding edges in order from least distance to greatest distance, it would be convenient to sort the edges by the

Each edge is added at most once so this contributes $\mathcal{O}(M\log N)$ to our time complexity. Answering all Q queries will contribute

Some details for implementation

Time Complexity

distance value and have some pointer to indicate which edges have been added so far.

 $\mathcal{O}(Q\log N)$. Thus, our final time complexity will be $\mathcal{O}((Q+M)\log N)$.

void Union(int x, int y) { // merges two disjoint sets into one set

static bool comp(vector<int>& a, vector<int>& b) { // sorting comparator

Bonus: We can use union by rank mentioned here to improve the time complexity of DSU operations from $\mathcal{O}(\log N)$ to $\mathcal{O}(lpha(N))$

Our <code>DSU</code> takes $\mathcal{O}(N)$ space and storing the answers to all Q queries takes $\mathcal{O}(Q)$ space so our final space complexity is $\mathcal{O}(N+1)$

C++ Solution class Solution {

return parent[x];

x = find(x);

y = find(y);

parent[x] = y;

return a[2] < b[2];</pre>

parent[x] = find(parent[x]);

vector<int> parent;

Space Complexity: $\mathcal{O}(N+Q)$

Time Complexity: $\mathcal{O}((Q+M)\log N)$.

int find(int x) { // finds the id/leader of a node if (parent[x] == x) { return x;

Space Complexity

Q).

```
public:
    vector<bool> distanceLimitedPathsExist(int n, vector<vector<int>>& edgeList,
                                            vector<vector<int>>& queries) {
        parent.resize(n);
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        sort(edgeList.begin(), edgeList.end(), comp); // sort edges by value for distance
        for (int j = 0; j < queries.size();</pre>
             i++) { // keeps track of original index for each query
            queries[j].push_back(j);
        sort(queries.begin(), queries.end(), comp); // sort queries by value for limit
        int i = 0;
        vector<bool> ans(queries.size());
        for (vector<int> query : queries) {
            int limit = query[2];
            while (i < edgeList.size() &&</pre>
                   edgeList[i][2] < limit) { // keeps adding edges with distance < limit</pre>
                int u = edgeList[i][0];
                int v = edgeList[i][1];
                Union(u, v);
                i++;
            int p = query[0];
            int q = query[1];
            int queryIndex = query[3];
            if (find(p) == find(q)) { // checks if p and q are connected
                ans[queryIndex] = true;
        return ans;
};
Java Solution
class Solution {
   private
    int find(int x, int[] parent) { // finds the id/leader of a node
        if (parent[x] == x) {
            return x;
        parent[x] = find(parent[x], parent);
        return parent[x];
   private
    void Union(int x, int y, int[] parent) { // merges two disjoint sets into one set
        x = find(x, parent);
        v = find(v, parent);
        parent[x] = y;
   public
    boolean[] distanceLimitedPathsExist(int n, int[][] edgeList, int[][] queries) {
        int[] parent = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        Arrays.sort(edgeList, (a, b)->a[2] - b[2]); // sort edges by value for distance
        for (int j = 0; j < queries.length;</pre>
             j++) { // keeps track of original index for each query
```

queries[j] = new int[]{queries[j][0], queries[j][1], queries[j][2], j};

Arrays.sort(queries, (a, b)->a[2] - b[2]); // sort queries by value for limit

edgeList[i][2] < limit) { // keeps adding edges with distance < limit</pre>

int i = 0;

boolean[] ans = new boolean[queries.length];

for (int j = 0; j < queries.length; j++) {</pre>

int[] query = queries[j];

while (i < edgeList.length &&</pre>

int u = edgeList[i][0];

int v = edgeList[i][1];

Union(u, v, parent);

int queryIndex = query[3];

int limit = query[2];

i++;

int p = query[0];

int q = query[1];

```
if (find(p, parent) == find(q, parent)) { // checks if p and q are connected
                ans[queryIndex] = true;
        return ans;
Python Solution
class Solution:
    def distanceLimitedPathsExist(
        self, n: int, edgeList: List[List[int]], queries: List[List[int]]
    ) -> List[bool]:
        parent = [i for i in range(n)]
        def find(x): # finds the id/leader of a node
            if parent[x] == x:
                return x
            parent[x] = find(parent[x])
            return parent[x]
        def Union(x, y): # merges two disjoint sets into one set
            x = find(x)
            v = find(v)
            parent[x] = y
        edgeList.sort(key=lambda x: x[2]) # sort edges by value for distance
        for i in range(len(queries)): # keeps track of original index for each query
            queries[i].append(i)
        queries.sort(key=lambda x: x[2]) # sort queries by value for limit
        i = 0
        ans = [False for j in range(len(queries))]
        for query in queries:
            limit = query[2]
            while (
                i < len(edgeList) and edgeList[i][2] < limit</pre>
            ): # keeps adding edges with distance < limit</pre>
                u = edgeList[i][0]
                v = edgeList[i][1]
                Union(u, v)
                i += 1
            p = query[0]
            q = query[1]
            queryIndex = query[3]
            if find(p) == find(q): # checks if p and q are connected
                ans[queryIndex] = True
        return ans
```