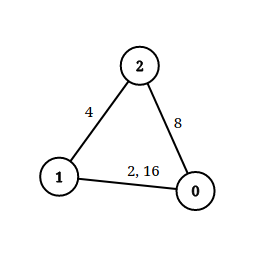
<https://leetcode.com/problems/checking-existence-of-edge-length-limited-paths/description/>

An undirected graph of n nodes is defined by edgeList, where edgeList[i] = [ui, vi, disi] denotes an edge between nodes ui and vi with distance disi. Note that there may be **multiple** edges between two nodes.

Given an array queries, where queries[j] = [pj, qj, limitj], 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** limitj .

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

**Example 1:**

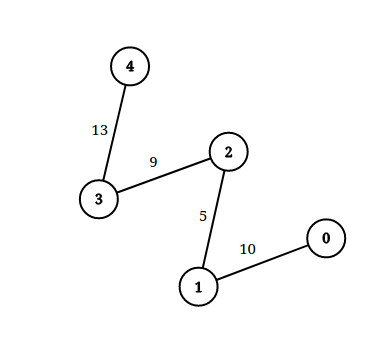


**Input:** n = 3, edgeList = [[0,1,2],[1,2,4],[2,0,8],[1,0,16]], queries = [[0,1,2],[0,2,5]]

**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 -> 1 -> 2) of two edges with distances less than 5, thus we return true for this query.

**Example 2:**



**Input:** n = 5, edgeList = [[0,1,10],[1,2,5],[2,3,9],[3,4,13]], queries = [[0,4,14],[1,4,13]]

**Output:** [true,false]

**Explanation:** The above figure shows the given graph.

**Constraints:**

2 <= n <= 10^5

1 <= edgeList.length, queries.length <= 10^5

edgeList[i].length == 3

queries[j].length == 3

0 <= ui, vi, pj, qj <= n - 1

ui != vi

pj != qj

1 <= disi, limitj <= 10^9

There may be **multiple** edges between two nodes.

**Attempt 1: 2025-04-13**

**Solution 1: Graph + Union Find (180 min)**

class Solution {

    public boolean[] distanceLimitedPathsExist(int n, int[][] edgeList, int[][] queries) {

        // Add original index to queries before sorting

        int[][] queriesWithIndex = new int[queries.length][4];

        for(int i = 0; i < queries.length; i++) {

            queriesWithIndex[i][0] = queries[i][0];

            queriesWithIndex[i][1] = queries[i][1];

            queriesWithIndex[i][2] = queries[i][2];

            queriesWithIndex[i][3] = i;

        }

        // Sort queries by limit

        Arrays.sort(queriesWithIndex, (a, b) -> a[2] - b[2]);

        // Sort edges by distance

        Arrays.sort(edgeList, (a, b) -> a[2] - b[2]);

        UnionFind uf = new UnionFind(n);

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

        int edgeIndex = 0;

        for(int[] query : queriesWithIndex) {

            int p = query[0];

            int q = query[1];

            int limit = query[2];

            int originalIndex = query[3];

            // Process all edges with distance < limit

            while(edgeIndex < edgeList.length && edgeList[edgeIndex][2] < limit) {

                int u = edgeList[edgeIndex][0];

                int v = edgeList[edgeIndex][1];

                uf.union(u, v);

                edgeIndex++;

            }

            // Check if p and q are connected

            result[originalIndex] = uf.find(p) == uf.find(q);

        }

        return result;

    }

}

class UnionFind {

    private int[] parent;

    private int[] rank;

    public UnionFind(int n) {

        parent = new int[n];

        rank = new int[n];

        for(int i = 0; i < n; i++) {

            parent[i] = i;

            rank[i] = 1;

        }

    }

    public int find(int x) {

        if(parent[x] == x) {

            return x;

        }

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

    }

    public void union(int x, int y) {

        int rootX = find(x);

        int rootY = find(y);

        if(rootX != rootY) {

            if(rank[x] > rank[y]) {

                parent[rootY] = rootX;

                rank[rootX] += rank[rootY];

            } else {

                parent[rootX] = rootY;

                rank[rootY] += rank[rootX];

            }

        }

    }

}

Time Complexity: O(ElogE + QlogQ).

Where E is the number of edges in edgeList and Q is the number of queries.

This comes from sorting both inputs.

Space: O(n). Where n is the number of nodes.

**Solution 2: DFS (30 min, TLE 18/24)**

class Solution {

    public boolean[] distanceLimitedPathsExist(int n, int[][] edgeList, int[][] queries) {

        List<int[]>[] graph = new ArrayList[n];

        for(int i = 0; i < n; i++) {

            graph[i] = new ArrayList<>();

        }

        for(int[] edge : edgeList) {

            int u = edge[0];

            int v = edge[1];

            int distance = edge[2];

            graph[u].add(new int[] {v, distance});

            graph[v].add(new int[] {u, distance});

        }

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

        for(int i = 0; i < queries.length; i++) {

            int p = queries[i][0];

            int q = queries[i][1];

            int limit = queries[i][2];

            result[i] = helper(p, q, limit, graph, new boolean[n]);

        }

        return result;

    }

    private boolean helper(int current, int target, int limit, List<int[]>[] graph, boolean[] visited) {

        if(current == target) {

            return true;

        }

        visited[current] = true;

        for(int[] neighbor : graph[current]) {

            int next = neighbor[0];

            int distance = neighbor[1];

            if(!visited[next] && distance < limit) {

                if(helper(next, target, limit, graph, visited)) {

                    return true;

                }

            }

        }

        return false;

    }

}

DFS/BFS per Query: O(V + E) for each query, where V is the number of vertices and E is the number of edges.

Overall Complexity: O(Q \* (V + E)), where Q is the number of queries. This is less efficient than the DSU approach,

especially for large graphs or many queries, but it's straightforward and easy to understand.

**Solution 3: BFS (30 min, TLE 18/24)**

class Solution {

    public boolean[] distanceLimitedPathsExist(int n, int[][] edgeList, int[][] queries) {

        // Build adjacency list

        List<int[]>[] graph = new ArrayList[n];

        for (int i = 0; i < n; i++) {

            graph[i] = new ArrayList<>();

        }

        for (int[] edge : edgeList) {

            int u = edge[0];

            int v = edge[1];

            int distance = edge[2];

            graph[u].add(new int[]{v, distance});

            graph[v].add(new int[]{u, distance});

        }

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

        for (int i = 0; i < queries.length; i++) {

            int p = queries[i][0];

            int q = queries[i][1];

            int limit = queries[i][2];

            result[i] = helper(p, q, limit, graph);

        }

        return result;

    }

    private boolean helper(int start, int end, int limit, List<int[]>[] graph) {

        Queue<Integer> q = new LinkedList<>();

        boolean[] visited = new boolean[graph.length];

        q.offer(start);

        visited[start] = true;

        while(!q.isEmpty()) {

            int cur = q.poll();

            if(cur == end) {

                return true;

            }

            for(int[] neighbor : graph[cur]) {

                int next = neighbor[0];

                int distance = neighbor[1];

                if(!visited[next] && distance < limit) {

                    visited[next] = true;

                    q.offer(next);

                }

            }

        }

        return false;

    }

}

DFS/BFS per Query: O(V + E) for each query, where V is the number of vertices and E is the number of edges.

Overall Complexity: O(Q \* (V + E)), where Q is the number of queries. This is less efficient than the DSU approach,

especially for large graphs or many queries, but it's straightforward and easy to understand.

**Refer to**

<https://leetcode.com/problems/checking-existence-of-edge-length-limited-paths/solutions/978450/c-dsu-two-pointers/>

**Observation**

The key here is to notice that the queries are offline which means that we can reorganize them however we want.

Now to answer the question, whether there is a path between any two nodes where the maximum edge length or weight is less than limit, we can join all the edges whose weight is less than limit and if we are still not able to reach one node from the other it essentially means that there is no path between them where edge weight is less than limit.

Which is the best data structure that can help us join edges as we want and answer whether in that structure, node a and node b are connected ?

That's right! **DSU**.

Let's try and use these facts to solve the question.

**Solution**

First we need to sort the input queries and edgeList by edge length or weight.

We can now simply use a two pointer approach to Union all the nodes whose edges have weight less than query[i].

To know if there is a path between them all we need is to know whether their parents (in DSU) are same.

// Standard Disjoint-set data structure implementation.

static class DSU {

vector<int> Parent, Rank;

public:

DSU(int n) {

Parent.resize(n);

Rank.resize(n, 0);

for (int i = 0; i < n; i++) Parent[i] = i;

}

int Find(int x) {

return Parent[x] = Parent[x] == x ? x : Find(Parent[x]);

}

bool Union(int x, int y) {

int xset = Find(x), yset = Find(y);

if (xset != yset) {

Rank[xset] < Rank[yset] ? Parent[xset] = yset : Parent[yset] = xset;

Rank[xset] += Rank[xset] == Rank[yset];

return true;

}

return false;

}

};

class Solution {

public:

{

DSU dsu(n);

//Add query indices to help with organizing/ordering results.

for(int i=0;i<queries.size();i++)

queries[i].push\_back(i);

//Sort inputs

sort(queries.begin(), queries.end(), [](auto &l, auto &r) { return l[2] < r[2]; });

sort(edgeList.begin(), edgeList.end(), [](auto &l, auto &r) { return l.back() < r.back(); });

int i=0;

vector<bool> result(queries.size());

for (vector<int> &q:queries)

{

// Two pointer approach. Join the edges till their weight is less than the current query.

while (i<edgeList.size()&&edgeList[i][2]<q[2])

dsu.Union(edgeList[i][0],edgeList[i++][1]);

//If parents are same we know that their is a path.

result[q.back()]=dsu.Find(q[0]) == dsu.Find(q[1]);

}

return result;

}

};

**Refer to Deepseek**

This problem requires us to determine for multiple queries whether there's a path between two nodes where all edges have lengths less than a given limit. An efficient solution involves using Union-Find (Disjoint Set Union) with offline processing.

**Approach**

Sort the queries by their limit values so we can process them in increasing order

Sort the edges by their weights in increasing order

Use Union-Find to gradually connect nodes as we process edges with weights less than the current query's limit

For each query, check if the nodes are connected after processing all relevant edges

**Solution Code**

class Solution {

public boolean[] distanceLimitedPathsExist(int n, int[][] edgeList, int[][] queries) {

// Add original index to queries before sorting

int[][] queriesWithIndex = new int[queries.length][4];

for (int i = 0; i < queries.length; i++) {

queriesWithIndex[i][0] = queries[i][0];

queriesWithIndex[i][1] = queries[i][1];

queriesWithIndex[i][2] = queries[i][2];

queriesWithIndex[i][3] = i;

}

// Sort queries by limit

Arrays.sort(queriesWithIndex, (a, b) -> a[2] - b[2]);

// Sort edges by distance

Arrays.sort(edgeList, (a, b) -> a[2] - b[2]);

UnionFind uf = new UnionFind(n);

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

int edgeIndex = 0;

for (int[] query : queriesWithIndex) {

int p = query[0];

int q = query[1];

int limit = query[2];

int originalIndex = query[3];

// Process all edges with distance < limit

while (edgeIndex < edgeList.length && edgeList[edgeIndex][2] < limit) {

int u = edgeList[edgeIndex][0];

int v = edgeList[edgeIndex][1];

uf.union(u, v);

edgeIndex++;

}

// Check if p and q are connected

result[originalIndex] = uf.find(p) == uf.find(q);

}

return result;

}

class UnionFind {

private int[] parent;

private int[] rank;

public UnionFind(int size) {

parent = new int[size];

rank = new int[size];

for (int i = 0; i < size; i++) {

parent[i] = i;

}

}

public int find(int x) {

if (parent[x] != x) {

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

}

return parent[x];

}

public void union(int x, int y) {

int rootX = find(x);

int rootY = find(y);

if (rootX != rootY) {

if (rank[rootX] > rank[rootY]) {

parent[rootY] = rootX;

} else if (rank[rootX] < rank[rootY]) {

parent[rootX] = rootY;

} else {

parent[rootY] = rootX;

rank[rootX]++;

}

}

}

}

}

**Explanation**

Query Processing: We first add the original index to each query so we can return results in the correct order after sorting.

Sorting: Both queries and edges are sorted by their limit/weight values to enable offline processing.

**Union-Find: We use the Union-Find data structure to efficiently track connected components. As we process each query:**

**We connect all nodes with edges smaller than the current query's limit**

**We check if the query's nodes are in the same connected component**

Efficiency: The algorithm runs in O(E log E + Q log Q) time due to sorting, and O(Q α(N)) for Union-Find operations, where α is the inverse Ackermann function (effectively constant).

This approach efficiently answers all queries by processing them in a way that minimizes redundant work through sorting and incremental Union-Find operations.

**If we don't use DSU solution, instead of native solution (no need consider time complexity optimization), how will it looks like ?**

If we don't use the optimized Union-Find (DSU) approach and instead implement a native solution without considering time complexity optimization, we can use Depth-First Search (DFS) or Breadth-First Search (BFS) for each query to check if there's a valid path where all edges have lengths less than the given limit.

**Approach**

Graph Representation: Convert the edge list into an adjacency list for easier traversal.

Query Processing: For each query, perform a DFS or BFS starting from the source node while only traversing edges with lengths less than the query's limit.

Path Check: During the traversal, check if the destination node is reachable under the given constraints.

**Solution Code (Native DFS Approach)**

class Solution {

public boolean[] distanceLimitedPathsExist(int n, int[][] edgeList, int[][] queries) {

// Build adjacency list

List<int[]>[] graph = new ArrayList[n];

for (int i = 0; i < n; i++) {

graph[i] = new ArrayList<>();

}

for (int[] edge : edgeList) {

int u = edge[0];

int v = edge[1];

int distance = edge[2];

graph[u].add(new int[]{v, distance});

graph[v].add(new int[]{u, distance});

}

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

for (int i = 0; i < queries.length; i++) {

int p = queries[i][0];

int q = queries[i][1];

int limit = queries[i][2];

result[i] = dfs(p, q, limit, graph, new boolean[n]);

}

return result;

}

private boolean dfs(int current, int target, int limit, List<int[]>[] graph, boolean[] visited) {

if (current == target) {

return true;

}

visited[current] = true;

for (int[] neighbor : graph[current]) {

int next = neighbor[0];

int distance = neighbor[1];

if (!visited[next] && distance < limit) {

if (dfs(next, target, limit, graph, visited)) {

return true;

}

}

}

return false;

}

}

**Solution Code (Native BFS Approach)**

class Solution {

public boolean[] distanceLimitedPathsExist(int n, int[][] edgeList, int[][] queries) {

// Build adjacency list

List<int[]>[] graph = new ArrayList[n];

for (int i = 0; i < n; i++) {

graph[i] = new ArrayList<>();

}

for (int[] edge : edgeList) {

int u = edge[0];

int v = edge[1];

int distance = edge[2];

graph[u].add(new int[]{v, distance});

graph[v].add(new int[]{u, distance});

}

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

for (int i = 0; i < queries.length; i++) {

int p = queries[i][0];

int q = queries[i][1];

int limit = queries[i][2];

result[i] = bfs(p, q, limit, graph);

}

return result;

}

private boolean bfs(int start, int target, int limit, List<int[]>[] graph) {

Queue<Integer> queue = new LinkedList<>();

boolean[] visited = new boolean[graph.length];

queue.offer(start);

visited[start] = true;

while (!queue.isEmpty()) {

int current = queue.poll();

if (current == target) {

return true;

}

for (int[] neighbor : graph[current]) {

int next = neighbor[0];

int distance = neighbor[1];

if (!visited[next] && distance < limit) {

visited[next] = true;

queue.offer(next);

}

}

}

return false;

}

}

**Explanation**

Graph Construction: The edge list is converted into an adjacency list where each node points to its neighbors along with the edge distances.

DFS/BFS Traversal: For each query, we initiate a traversal (DFS or BFS) from the source node, only moving to neighbors if the connecting edge's distance is less than the query's limit.

Result Compilation: The result for each query is true if the destination node is reached during the traversal, otherwise false.

**Time Complexity**

DFS/BFS per Query: O(V + E) for each query, where V is the number of vertices and E is the number of edges.

Overall Complexity: O(Q \* (V + E)), where Q is the number of queries. This is less efficient than the DSU approach, especially for large graphs or many queries, but it's straightforward and easy to understand.

This native approach is conceptually simple but not optimal for large inputs, making the DSU solution preferable for performance-critical applications.

**Refer to**

[L2421.Number of Good Paths (Ref.L1671,L2506)](note://WEBe0a7e9243acd024b92c4cd8a899fb0d3)