<https://leetcode.com/problems/minimum-weighted-subgraph-with-the-required-paths/description/>

You are given an integer n denoting the number of nodes of a **weighted directed** graph. The nodes are numbered from 0 to n - 1.

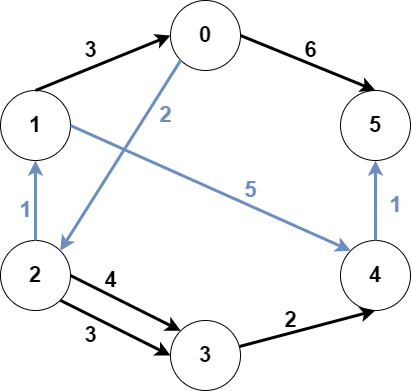
You are also given a 2D integer array edges where edges[i] = [fromi, toi, weighti] denotes that there exists a **directed** edge from fromi to toi with weight weighti.

Lastly, you are given three **distinct** integers src1, src2, and dest denoting three **distinct** nodes of the graph.

Return the **minimum weight** of a subgraph of the graph such that it is **possible** to reach dest from both src1 and src2 via a set of edges of this subgraph. In case such a subgraph does not exist, return -1.

A **subgraph** is a graph whose vertices and edges are subsets of the original graph. The **weight** of a subgraph is the sum of weights of its constituent edges.

**Example 1:**



**Input:** n = 6, edges = [[0,2,2],[0,5,6],[1,0,3],[1,4,5],[2,1,1],[2,3,3],[2,3,4],[3,4,2],[4,5,1]], src1 = 0, src2 = 1, dest = 5**Output:** 9**Explanation:**The above figure represents the input graph.The blue edges represent one of the subgraphs that yield the optimal answer.Note that the subgraph [[1,0,3],[0,5,6]] also yields the optimal answer. It is not possible to get a subgraph with less weight satisfying all the constraints.

**Example 2:**



**Input:** n = 3, edges = [[0,1,1],[2,1,1]], src1 = 0, src2 = 1, dest = 2**Output:** -1**Explanation:**The above figure represents the input graph.It can be seen that there does not exist any path from node 1 to node 2, hence there are no subgraphs satisfying all the constraints.

**Constraints:**

3 <= n <= 10^5

0 <= edges.length <= 10^5

edges[i].length == 3

0 <= fromi, toi, src1, src2, dest <= n - 1

fromi != toi

src1, src2, and dest are pairwise distinct.

1 <= weight[i] <= 10^5

**Attempt 1: 2024-12-16**

**Solution 1: Dijkstra (120 min)**

class Solution {

    // Define infinity as the maximum value a long can hold.

    private static final long INFINITY = Long.MAX\_VALUE;

    public long minimumWeight(int n, int[][] edges, int src1, int src2, int dest) {

        // Prepare graph and reverse graph

        // We don't need to use map because the nodes are numbered from 0 to n - 1,

        // just mapping it to list's natural index is fine

        List<List<Node>> graph = new ArrayList<>();

        List<List<Node>> reverseGraph = new ArrayList<>();

        for(int i = 0; i < n; i++) {

            graph.add(new ArrayList<>());

            reverseGraph.add(new ArrayList<>());

        }

        for(int[] edge : edges) {

            int from = edge[0];

            int to = edge[1];

            long weight = (long) edge[2];

            graph.get(from).add(new Node(to, weight));

            reverseGraph.get(to).add(new Node(from, weight));

        }

        // Run Dijkstra's algorithm to find shortest paths from src1 and src2

        // in the graph (g) and from dest in the reverse graph (rg)

        long[] distFromSrc1 = helper(graph, src1);

        long[] distFromSrc2 = helper(graph, src2);

        long[] distToDest = helper(reverseGraph, dest);

        long result = INFINITY;

        for(int i = 0; i < n; i++) {

            // Skip if any one of the distances is infinity

            if(distFromSrc1[i] == INFINITY || distFromSrc2[i] == INFINITY || distToDest[i] == INFINITY) {

                continue;

            }

            // Calculate the total distance for the current node

            result = Math.min(result, distFromSrc1[i] + distFromSrc2[i] + distToDest[i]);

        }

        return result == INFINITY ? -1 : result;

    }

    private long[] helper(List<List<Node>> graph, int startNode) {

        int n = graph.size();

        long[] distances = new long[n];

// Based on Dijkstra algorithm, start with all distances set to infinity

        Arrays.fill(distances, INFINITY);

        // Distance to start node is zero

        distances[startNode] = 0;

        PriorityQueue<Node> minPQ = new PriorityQueue<>((a, b) -> Long.compare(a.weight, b.weight));

        minPQ.offer(new Node(startNode, 0));

        while(!minPQ.isEmpty()) {

            Node cur = minPQ.poll();

            int curNode = cur.nodeId;

            long curDistance = cur.weight;

            // Skip if we have already found a shorter path to this node previously

            if(curDistance > distances[curNode]) {

                continue;

            }

            // For each neighbor, update the shortest distance found and add it to the queue

            for(Node neighbor : graph.get(curNode)) {

                int neighborNode = neighbor.nodeId;

                long neighborWeight = neighbor.weight;

                if(neighborWeight + distances[curNode] < distances[neighborNode]) {

                    distances[neighborNode] = neighborWeight + distances[curNode];

                    minPQ.offer(neighbor);

                }

            }

        }

        return distances;

    }

}

class Node {

    int nodeId;

    long weight;

    public Node(int nodeId, long weight) {

        this.nodeId = nodeId;

        this.weight = weight;

    }

}

Time Complexity: O(E + VlogV)

Space Complexity: O(E + V)

where E is the space for storing the edges and V is the space for storing the nodes and the maximum space for the priority queue

**Refer to**

<https://leetcode.com/problems/minimum-weighted-subgraph-with-the-required-paths/solutions/1844091/c-dijkstra-3-times-with-illustration/>

**Solution 1. Dijkstra**

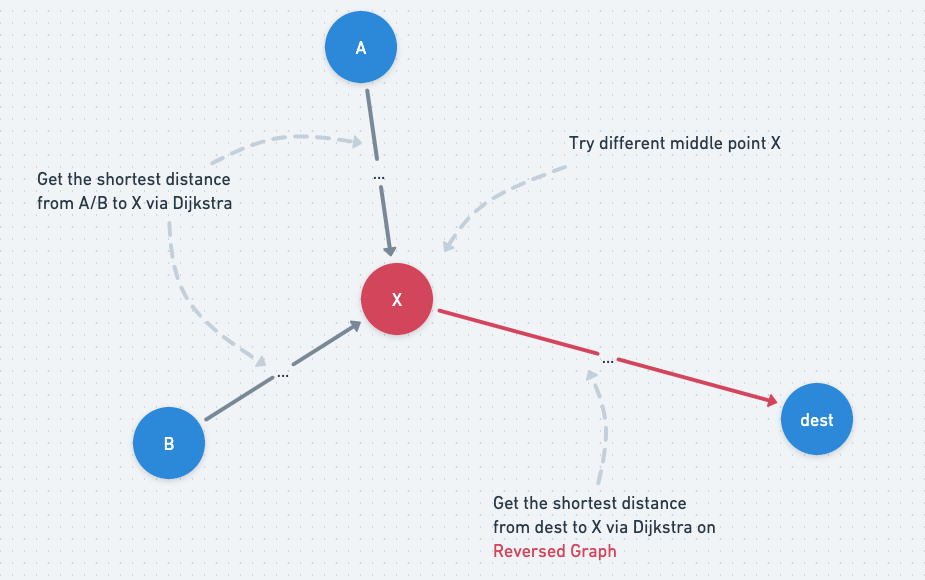
Do Dijkstra 3 times.

**First time:** store the shortest distance from node a to all other nodes in array da.

**Second time:** store the shortest distance from node b to all other nodes in array db.

**Third time:** store the shortest distance from node dest to all other nodes via **Reversed Graph** in array dd.

The answer is the minimum da[i] + db[i] + dd[i] (0 <= i < N).



**Python code**

// OJ: https://leetcode.com/contest/weekly-contest-284/problems/minimum-weighted-subgraph-with-the-required-paths/

// Author: github.com/lzl124631x

// Time: O(ElogE + N)

// Space: O(E)

class Solution {

typedef pair<long, long> ipair;

public:

long long minimumWeight(int n, vector<vector<int>>& E, int a, int b, int dest) {

vector<vector<ipair>> G(n), R(n); // `G` is the original graph. `R` is the reversed graph

for (auto &e : E) {

long u = e[0], v = e[1], w = e[2];

G[u].emplace\_back(v, w);

R[v].emplace\_back(u, w);

}

vector<long> da(n, LONG\_MAX), db(n, LONG\_MAX), dd(n, LONG\_MAX);

priority\_queue<ipair, vector<ipair>, greater<ipair>> pq;

dist[a] = 0;

pq.emplace(0, a);

while (pq.size()) {

auto [cost, u] = pq.top();

pq.pop();

if (cost > dist[u]) continue;

for (auto &[v, c] : G[u]) {

if (dist[v] > dist[u] + c) {

dist[v] = dist[u] + c;

pq.emplace(dist[v], v);

}

}

}

};

solve(G, a, da);

solve(G, b, db);

solve(R, dest, dd);

long ans = LONG\_MAX;

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

if (da[i] == LONG\_MAX || db[i] == LONG\_MAX || dd[i] == LONG\_MAX) continue;

ans = min(ans, da[i] + db[i] + dd[i]);

}

return ans == LONG\_MAX ? -1 : ans;

}

};

**Java version**

class Solution {

public long minimumWeight(int n, int[][] edges, int src1, int src2, int dest) {

List<List<Pair<Integer, Long>>> graph = new ArrayList<>();

List<List<Pair<Integer, Long>>> rgraph = new ArrayList<>();

// init graph

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

graph.add(new ArrayList<>());

rgraph.add(new ArrayList<>());

}

// build graph

for(int[] e : edges){

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

long w = (long) e[2];

graph.get(from).add(new Pair<Integer, Long>(to, w));

rgraph.get(to).add(new Pair<Integer, Long>(from, w));

}

// init cost array for 3 Dijkstra's

long[] cost1 = new long[n], cost2 = new long[n], costDes = new long[n];

Arrays.fill(cost1, Long.MAX\_VALUE);

Arrays.fill(cost2, Long.MAX\_VALUE);

Arrays.fill(costDes, Long.MAX\_VALUE);

// 3 Dijkstra's

bfs(graph, src1, cost1);

bfs(graph, src2, cost2);

bfs(rgraph, dest, costDes);

// find min cost

long result = Long.MAX\_VALUE;

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

if(cost1[i] == Long.MAX\_VALUE || cost2[i] == Long.MAX\_VALUE ||

costDes[i] == Long.MAX\_VALUE) {

continue;

}

result = Math.min(result, cost1[i] + cost2[i] + costDes[i]);

}

return result < Long.MAX\_VALUE? result : -1;

}

// Dijkstra

private void bfs(List<List<Pair<Integer, Long>>> g, int from, long[] cost) {

PriorityQueue<Pair<Integer, Long>> heap = new PriorityQueue<>((a, b) -> Long.compare(a.getValue(), b.getValue()));

cost[from] = 0;

heap.add(new Pair<Integer, Long>(from, cost[from]));

while(!heap.isEmpty()){

Pair<Integer, Long> curr = heap.poll();

int nodeCurr = curr.getKey();

long costCurr = curr.getValue();

if(cost[nodeCurr] < costCurr){

continue;

}

if(g.get(nodeCurr).isEmpty()){

continue;

}

for(Pair<Integer, Long> entry : g.get(nodeCurr)){

int next = entry.getKey();

long costNext = entry.getValue();

if(cost[next] > cost[nodeCurr] + costNext){

cost[next] = cost[nodeCurr] + costNext;

heap.add(new Pair<Integer, Long>(next, cost[next]));

}

}

}

}

}

**Why reversed graph in Step 3 ?**

**Refer to**

<https://leetcode.com/problems/minimum-weighted-subgraph-with-the-required-paths/solutions/1844091/c-dijkstra-3-times-with-illustration/comments/1306175>

oYu need to find shortest distance from i to dest. If you move from i to dest in the same graph then you need to compute shortest distance from all the i's. Which will take O((n+Elogn)\*n) time and we can't afford that in given constraints.

So, how to improve???

As you can see that we always need to find shortest distance from i to dest. Here dest is not changing So we can find shortest distance from dest to all other nodes, which will take O(n + Elogn) time and O(n) space, but we can't do that in original graph as in original graph edges are from i to dest so we reverse the edges. And now dd[i] actually means distance from i to dest.

**Refer to**

<https://algo.monster/liteproblems/2203>

**Problem Description**

The given LeetCode problem presents a scenario where you have a weighted directed [graph](https://algo.monster/problems/graph_intro) with n nodes numbered from 0 to n - 1. The connections between nodes are given by a list of edges, where each edge is represented by a triplet [from, to, weight], indicating a directed edge from node from to node to with a given weight.

Alongside this [graph](https://algo.monster/problems/graph_intro), you're provided with three distinct nodes: src1, src2, and dest. The goal is to find the minimum weight of a subgraph that allows both src1 and src2 to reach the node dest. If no such subgraph exists that allows this, then the result should be -1.

The problem is akin to finding the shortest paths in a directed [graph](https://algo.monster/problems/graph_intro), but with the added complexity of needing to ensure that two distinct source nodes can reach a common destination while minimizing the total weight of the paths used.

**Intuition**

To solve this problem, we can use Dijkstra's algorithm to find the [shortest path](https://algo.monster/problems/shortest_path_unweight) from each source node (src1 and src2) to dest. Dijkstra's algorithm is a classic algorithm for finding the shortest paths from a single source node to all other nodes in a [graph](https://algo.monster/problems/graph_intro) with non-negative edge weights.

The solution takes advantage of a modified version of Dijkstra's algorithm. Normally, Dijkstra's algorithm is used to compute the shortest paths from a single source to all other nodes. However, our problem requires shortest paths from two sources src1 and src2 to one destination dest, and the paths must be part of some common subgraph.

The approach involves calculating three sets of shortest paths:

Shortest paths from src1 to all other nodes.

Shortest paths from src2 to all other nodes.

**Shortest paths from dest to all other nodes (this requires reversing the edge directions to treat dest as a source).**

**Step 3's reverse**[**graph**](https://algo.monster/problems/graph_intro)**computation is key for determining the**[**shortest path**](https://algo.monster/problems/shortest_path_unweight)**from dest to other nodes in the original direction.**

After computing the [shortest path](https://algo.monster/problems/shortest_path_unweight) arrays, a simple linear pass can combine the paths from src1 and src2 to any intermediary node and from that intermediary node to dest (using reverse paths calculated in step 3). Adding these three path weights provides the total weight for each possible pair of paths reaching dest from each source. The minimum of these sums, if it is finite, gives us the minimum weight of the subgraph meeting the problem's criteria. If there's no subgraph where both src1 and src2 can reach dest, we return -1.

The intuition behind this triple shortest-path computation is to exploit the fact that if the optimal paths from src1 and src2 to dest share any common segment, those segments should be counted only once in the total weight of the subgraph.

**Solution Approach**

The solution approach to this problem involves several important steps which can be broken down into the following:

**Graph Representation**: The implementation uses a defaultdict from Python's collections module to represent the graph g and the reverse graph rg. The defaultdict allows for easy appending of edges without worrying about key errors. For instance, g[f].append((t, w)) adds a directed edge from f to t with weight w to the graph g.

**Dijkstra's Algorithm:** Dijkstra's algorithm is used to compute the shortest paths. A helper function dijkstra(g, u) is defined, which performs Dijkstra's on the [graph](https://algo.monster/problems/graph_intro) g from the source node u. This function initializes a distance list dist with inf (infinity). This list holds the shortest distance from u to every other node. The distance from u to itself is set to 0. A priority queue q is then used to select the next node with the smallest distance, which makes the process efficient by always considering the closest non-visited node. The queue begins with the source node u.

**Priority Queue:** Python's heapq module is used to implement the priority queue in Dijkstra's algorithm. It ensures the selection of the minimum distance which is not yet processed. This is an efficient way to get the next node to process for the shortest paths (heappop(q) retrieves and removes the node with the smallest distance from the queue).

**Computing Shortest Paths:** The algorithm computes three sets of shortest paths: one from src1 to all other nodes as d1, one from src2 to all other nodes as d2, and one from dest to all other nodes using the reverse [graph](https://algo.monster/problems/graph_intro) rg as d3. The reverse paths are necessary as we want to compute the [shortest path](https://algo.monster/problems/shortest_path_unweight) from dest to any node efficiently, emulating a shortest path to dest.

**Finding the Minimal Subgraph:** Once we have the shortest paths from src1, src2, and dest to all other nodes, we iterate through all the nodes to calculate the sum of distances from src1 to an intermediary node i, from src2 to the same intermediary i, and from i to dest (using the reverse paths d3). ans = min(sum(v) for v in zip(d1, d2, d3)) goes through all these sums and finds the minimum.

**Returning the Result:** Finally, if the minimum found in the previous step is infinite, then there is no valid subgraph, and the function returns -1. Otherwise, the minimum value is the answer which represents the minimum weight of the subgraph where src1 and src2 can both reach dest.

In summary, the solution effectively combines [graph](https://algo.monster/problems/graph_intro) representation, Dijkstra's algorithm, efficient data structures like heapq, and algorithmic ingenuity to tackle this path-finding problem and guarantee the minimal subgraph weight that satisfies the given conditions.

**Solution Implementation**

class Solution {

// Define infinity as the maximum value a long can hold.

private static final long INFINITY = Long.MAX\_VALUE;

// The minimumWeight method is responsible for finding the minimum total weight

// of paths from two sources (src1 and src2) to one destination (dest).

public long minimumWeight(int n, int[][] edges, int src1, int src2, int dest) {

// Initialize adjacency lists for the graph (g) and the reverse graph (rg)

List<Pair<Integer, Long>>[] graph = new List[n];

List<Pair<Integer, Long>>[] reverseGraph = new List[n];

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

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

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

}

// Populate the adjacency list for each edge in the graph.

for (int[] edge : edges) {

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

long weight = edge[2];

graph[from].add(new Pair<>(to, weight));

reverseGraph[to].add(new Pair<>(from, weight));

}

// Run Dijkstra's algorithm to find shortest paths from src1 and src2

// in the graph (g) and from dest in the reverse graph (rg).

long[] distancesFromSrc1 = dijkstra(graph, src1);

long[] distancesFromSrc2 = dijkstra(graph, src2);

long[] distancesToDest = dijkstra(reverseGraph, dest);

// Initialize answer as -1 to indicate no solution found yet.

long answer = -1;

// Iterate over all nodes to find the minimum combined distances.

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

// Skip if any one of the distances is infinity.

if (distancesFromSrc1[i] == INFINITY || distancesFromSrc2[i] == INFINITY || distancesToDest[i] == INFINITY) {

continue;

}

// Calculate the total distance for the current node.

long totalDistance = distancesFromSrc1[i] + distancesFromSrc2[i] + distancesToDest[i];

// Update the answer if totalDistance is smaller than the current answer.

if (answer == -1 || answer > totalDistance) {

answer = totalDistance;

}

}

// Return the minimum total weight of the paths found.

return answer;

}

// The dijkstra method implements Dijkstra's algorithm to find the shortest paths

// from a starting node (startNode) to all other nodes in the graph.

private long[] dijkstra(List<Pair<Integer, Long>>[] graph, int startNode) {

int n = graph.length;

long[] distances = new long[n];

Arrays.fill(distances, INFINITY); // Start with all distances set to infinity.

distances[startNode] = 0; // Distance to start node is zero.

PriorityQueue<Pair<Long, Integer>> queue = new PriorityQueue<>(Comparator.comparingLong(Pair::getKey));

queue.offer(new Pair<>(0L, startNode)); // Add the start node to the priority queue.

while (!queue.isEmpty()) {

Pair<Long, Integer> current = queue.poll();

long currentDistance = current.getKey();

int currentNode = current.getValue();

// Skip if we have already found a shorter path to this node.

if (currentDistance > distances[currentNode]) {

continue;

}

// For each neighbor, update the shortest distance found and add it to the queue.

for (Pair<Integer, Long> edge : graph[currentNode]) {

int neighbor = edge.getKey();

long edgeWeight = edge.getValue();

if (distances[neighbor] > distances[currentNode] + edgeWeight) {

distances[neighbor] = distances[currentNode] + edgeWeight;

queue.offer(new Pair<>(distances[neighbor], neighbor));

}

}

}

// Return the array of shortest distances from the start node.

return distances;

}

}

**Time and Space Complexity**

**Time Complexity:**

The given code implements the Dijkstra's algorithm three times. The time complexity of Dijkstra's algorithm is O(E + V log V) for each call, where E is the number of edges and V is the number of vertices (nodes) in the graph. In the code, we use a priority queue to implement the algorithm efficiently.

The first call of Dijkstra's algorithm calculates the shortest paths from src1 to all other nodes.

The second call calculates the shortest paths from src2 to all other nodes.

The third call is on the reversed graph to compute the shortest paths from dest to all other nodes.

Since we are calling this algorithm three times, the time complexity will be O(3 \* (E + V log V)), which simplifies to O(E + V log V) because constants are dropped in Big O notation.

**Space Complexity:**

The space complexity of the code is determined by the storage required for the graph representation and the auxiliary data structures used in Dijkstra's algorithm, such as the distance array and the priority queue.

Graph g and reversed graph rg use adjacency lists to store information, which require O(E) space.

For each call of Dijkstra's algorithm, a distance array of size V is created, thus 3 \* O(V) for three calls.

The priority queue can hold at most V elements at any time, which contributes O(V).

Combining these, the space complexity is O(E + V), where E is the space for storing the edges and V is the space for storing the nodes and the maximum space for the priority queue.

**Refer to**

[L1368.Minimum Cost to Make at Least One Valid Path in a Grid (Ref.L743,L2290)](note://WEB4b3aa9003dbf5915537540decf09b434)

[L1568.Minimum Number of Days to Disconnect Island (Ref.L200)](note://WEBe1f9dea2f2571376df84b64404041136)

[L2556.Disconnect Path in a Binary Matrix by at Most One Flip](note://WEB3322914f94be5e34556afdcf5b35b500)