**Problem: Telecommunication Network Optimization**

**Scenario**

You are the network engineer for a telecommunication company responsible for optimizing the network connections between several cities. Each city is connected to other cities through fiber optic cables. Each connection has a certain cost associated with laying the cable.

Your objective is to ensure that all cities are connected with the minimum possible cost while including at least one high-priority connection that has been mandated by the company. These high-priority connections are critical for strategic reasons.

**Problem Statement**

Given a list of cities and the costs to connect them, find the Minimum Spanning Tree (MST) that includes at least one high-priority connection. If it's not possible to connect all cities while including at least one high-priority connection, print -1.

**Input Format**

* The first line contains three integers n, m, and k representing the number of cities, the number of connections, and the number of high-priority connections, respectively.
* The next m lines contain three integers each, u, v, and w, describing a connection between cities u and v with a cost w.
* The following k lines contain two integers each, a and b, indicating that the connection between cities a and b is a high-priority connection.

**Constraints**

* 2 <= n <= 1000
* n - 1 <= m <= 5000
* 1 <= w <= 10^5
* Each city is labeled from 1 to n.
* The network is connected.
* 1 <= k <= m

**Output Format**

Print the total cost of the MST that includes at least one high-priority connection. If no such MST exists, print -1.

**Sample Input**

4 5 2

1 2 1

1 3 2

1 4 3

2 3 4

3 4 5

1 2

3 4

**Sample Output**

6

**Explanation**

The MST of the graph including at least one high-priority connection (1, 2) or (3, 4) is:

* Include connection (1, 2) with cost 1
* Include connection (1, 3) with cost 2
* Include connection (1, 4) with cost 3 The total cost of the MST is 1 + 2 + 3 = 6.

**Solution**

We can solve this problem using a modified version of Kruskal's algorithm. The algorithm can be adjusted to ensure that at least one high-priority connection is included in the MST.

Here's the Python function to solve the problem:

python

class UnionFind:

def \_\_init\_\_(self, size):

self.parent = list(range(size))

self.rank = [1] \* size

def find(self, x):

if self.parent[x] != x:

self.parent[x] = self.find(self.parent[x])

return self.parent[x]

def union(self, x, y):

rootX = self.find(x)

rootY = self.find(y)

if rootX != rootY:

if self.rank[rootX] > self.rank[rootY]:

self.parent[rootY] = rootX

elif self.rank[rootX] < self.rank[rootY]:

self.parent[rootX] = rootY

else:

self.parent[rootY] = rootX

self.rank[rootX] += 1

return True

return False

def minimumSpanningTreeWithHighPriorityEdges(n, edges, highPriorityEdges):

edges.sort(key=lambda x: x[2]) # Sort edges based on cost

highPriorityEdgesSet = set((a, b) for a, b in highPriorityEdges)

minCost = float('inf')

for highPriority in highPriorityEdges:

uf = UnionFind(n + 1)

totalCost = 0

highPriorityIncluded = False

for edge in edges:

u, v, w = edge

if uf.union(u, v):

totalCost += w

if (u, v) in highPriorityEdgesSet or (v, u) in highPriorityEdgesSet:

highPriorityIncluded = True

if highPriorityIncluded and all(uf.find(i) == uf.find(1) for i in range(1, n + 1)):

minCost = min(minCost, totalCost)

return minCost if minCost != float('inf') else -1

# Input

n, m, k = map(int, input().split())

edges = []

for \_ in range(m):

u, v, w = map(int, input().split())

edges.append((u, v, w))

highPriorityEdges = []

for \_ in range(k):

a, b = map(int, input().split())

highPriorityEdges.append((a, b))

# Output

print(minimumSpanningTreeWithHighPriorityEdges(n, edges, highPriorityEdges))

**Additional Test Cases**

**Test Case 1**

**Input:**

5 7 3

1 2 1

1 3 2

1 4 3

2 3 4

2 5 5

3 4 6

4 5 7

1 2

3 4

4 5

**Output:**

11

**Test Case 2**

**Input:**

3 3 1

1 2 1

2 3 2

1 3 3

1 3

**Output:**

-1

**Test Case 3**

**Input:**

6 9 2

1 2 3

1 3 1

1 4 6

2 3 5

2 5 4

3 4 2

3 6 7

4 5 8

5 6 9

2 5

3 6

**Output:**

17

**Test Case 4**

**Input:**

6 8 2

1 2 3

1 3 1

1 4 6

2 3 5

2 5 4

3 4 2

3 6 7

4 5 8

2 5

3 6

**Output:**

17

**Test Case 5**

**Input:**

4 3 2

1 2 3

1 3 1

1 4 6

2 5

3 6

**Output:**

-1