NAME : SUMALATHA D K

REG NO : 192424023

COURSE CODE : CSA0613

COURSE NAME : DESIGN AND ANALYSIS OF ALGORITHMS FOR OPTIMAL APPLICATIONS

SLOT : A

TOPIC 5 GREEDY

1. **There are 3n piles of coins of varying size, you and your friends will take piles of coins as follows: In each step, you will choose any 3 piles of coins (not necessarily consecutive). Of your choice, Alice will pick the pile with the maximum number of coins. You will pick the next pile with the maximum number of coins. Your friend Bob will pick the last pile. Repeat until there are no more piles of coins. Given an array of integers piles where piles[i] is the number of coins in the ith pile. Return the maximum number of coins that you can have.**

AIM:

To maximize the number of coins you can collect from **3n piles** when picking coins in groups of three, following the order: Alice picks the largest, you pick the second-largest, and Bob picks the smallest.

ALGORITHM:

1. Sort the piles array in **descending order**.
2. Since Alice always picks the largest pile, and Bob the smallest, you should pick the **second-largest** pile in each triplet.
3. After sorting, your coins will be every second pile starting from the second pile and skipping the last pile taken by Bob.
4. Sum up your coins.

CODE:

```
def maxCoins(piles):
    piles.sort(reverse=True)
    n = len(piles) // 3
    coins = 0
    for i in range(1, 2*n+1, 2):
        coins += piles[i]
    return coins
```

INPUT:

```
# Test Cases
print(maxCoins([2,4,1,2,7,8]))
print(maxCoins([2,4,5]))
```

OUTPUT:

```
9
4
```

2. **You are given a 0-indexed integer array coins, representing the values of the coins available, and an integer target. An integer x is obtainable if there exists a subsequence of coins that sums to x. Return the minimum number of coins of any value that need to be added to the array so that every integer in the range [1, target] is obtainable. A subsequence of an array is a new non-empty array that is formed from the original array by deleting some (possibly none) of the elements without disturbing the relative positions of the remaining elements.**

AIM:

To determine the **minimum number of coins** that need to be added to an array so that every integer in the range [1, target] can be formed as a subsequence sum of the array.

ALGORITHM:

1. Sort the coins array in ascending order.
2. Initialize miss = 1 (smallest value not obtainable yet) and added = 0 (number of coins added).
3. Traverse the sorted coins:
    - If the current coin c ≤ miss, then we can form sums up to miss + c - 1 by including c. Update miss = miss + c.
    - If c > miss, we **add a coin with value miss**, increment added, and update miss = 2*miss.

4. Repeat until miss > target.

CODE:

```
def minPatches(coins, target):
    coins.sort()
    miss = 1
    added = 0
    i = 0
    while miss <= target:
        if i < len(coins) and coins[i] <= miss:
            miss += coins[i]
            i += 1
        else:
            # add coin of value 'miss'
            added += 1
            miss *= 2
    return added
```

INPUT:

```
# Test Cases
print(minPatches([1,4,10], 19))
print(minPatches([1,4,10,5,7,19], 19))
```

OUTPUT:

```
2
1
```

3. **You are given an integer array jobs, where jobs[i] is the amount of time it takes to complete the ith job. There are k workers that you can assign jobs to. Each job should be assigned to exactly one worker. The working time of a worker is the sum of the time it takes to complete all jobs assigned to them. Your goal is to devise an optimal assignment such that the maximum working time of any worker is minimized. Return the minimum possible maximum working time of any assignment.**

AIM:

To assign n jobs to k workers such that the **maximum working time among all workers is minimized**.

ALGORITHM:

This is a **job scheduling problem** (minimizing maximum workload), solved using **Binary Search + Backtracking (DFS)**:

1. **Binary Search Range**:
    - Lower bound (low) = max(jobs) → at least one worker must do the largest job.

- Upper bound (high) = sum(jobs) → if one worker does all jobs.

2. **Check feasibility** (DFS / backtracking):
   - Try to assign jobs to workers without exceeding a target mid.
   - If all jobs can be assigned under mid, it's feasible.
3. **Binary Search**:
   - If mid is feasible, search for smaller max (high = mid).
   - If mid is not feasible, search higher (low = mid + 1).
4. Continue until low == high. This value is the **minimum possible maximum working time**.

CODE:

```
def minimumTime(jobs, k):
    def can_assign(mid):
        workers = [0]*k
        jobs_sorted = sorted(jobs, reverse=True)
        def dfs(index):
            if index == len(jobs_sorted):
                return True
            for i in range(k):
                if workers[i] + jobs_sorted[index] <= mid:
                    workers[i] += jobs_sorted[index]
                    if dfs(index + 1):
                        return True
                    workers[i] -= jobs_sorted[index]
                if workers[i] == 0:
                    break
            return False
        return dfs(0)
    low, high = max(jobs), sum(jobs)
    while low < high:
        mid = (low + high) // 2
        if can_assign(mid):
            high = mid
        else:
            low = mid + 1
    return low
```

INPUT:

```
# Test Cases
print(minimumTime([3,2,3], 3))
print(minimumTime([1,2,4,7,8], 2))
```

OUTPUT:

```
3
11
```

4. **We have n jobs, where every job is scheduled to be done from startTime[i] to endTime[i], obtaining a profit of profit[i]. You're given the startTime, endTime and profit arrays, return the maximum profit you can take such that there are no two jobs in the subset with overlapping time range. If you choose a job that ends at time X you will be able to start another job that starts at time X.**

AIM:

To schedule jobs such that **no two jobs overlap** and the **total profit is maximized**.

ALGORITHM:

This is the **Weighted Job Scheduling Problem**, solved efficiently using **Dynamic Programming with Binary Search**:

1. **Combine and sort jobs by endTime**:
   - Pair jobs as (start, end, profit) and sort by end.
2. **DP array**:
   - dp[i] = maximum profit considering jobs up to index i.
3. **Binary search for non-overlapping job**:
   - For job i, find the **last job j that ends before job i starts**.
   - dp[i] = max(dp[i-1], dp[j] + profit[i])
4. **Return** dp[n-1] as maximum profit.

CODE:

```
from bisect import bisect_right
def jobScheduling(startTime, endTime, profit):
    # Combine jobs and sort by endTime
    jobs = sorted(zip(startTime, endTime, profit), key=lambda x: x[1])
    dp = []
    ends = []
    for s, e, p in jobs:
        # Find the last job that ends before current job starts
        i = bisect_right(ends, s) - 1
        if i != -1:
            p += dp[i]
        if dp:
            dp.append(max(dp[-1], p))
        else:
            dp.append(p)
        ends.append(e)
    return dp[-1]
```

INPUT: **OUTPUT:**

```
# Test Cases
print(jobScheduling([1,2,3,3],[3,4,5,6],[50,10,40,70]))
print(jobScheduling([1,2,3,4,6],[3,5,10,6,9],[20,20,100,70,60]))
```

```
120
150
```

5. **Given a graph represented by an adjacency matrix, implement Dijkstra's Algorithm to find the shortest path from a given source vertex to all other vertices in the graph. The graph is represented as an adjacency matrix where graph[i][j] denote the weight of the edge from vertex i to vertex j. If there is no edge between vertices i and j, the value is Infinity (or a very large number).**

AIM:

To find the **shortest path from a given source vertex to all other vertices** in a weighted graph using **Dijkstra's Algorithm**.

ALGORITHM:

1. **Initialize distances**:
   - Create a distance array dist[] and set all distances to infinity except the source (dist[source] = 0).
   - Keep a set of visited vertices to avoid revisiting.
2. **Iterate over all vertices**:
   - Pick the **unvisited vertex with the smallest distance** (u).
   - Mark u as visited.
3. **Update distances of neighbors**:
   - For every neighbor v of u, if v is unvisited and dist[u] + graph[u][v] < dist[v], then update dist[v].
4. **Repeat until all vertices are visited**.
5. **Return** the distance array dist[].

CODE:

```
import math
def dijkstra(graph, source):
    n = len(graph)
    dist = [math.inf] * n
    dist[source] = 0
    visited = [False] * n
    for _ in range(n):
        # Select the unvisited vertex with minimum distance
        u = -1
        min_dist = math.inf
        for i in range(n):
            if not visited[i] and dist[i] < min_dist:
                min_dist = dist[i]
                u = i
        if u == -1:
            break
        visited[u] = True
        # Update distances of neighbors
        for v in range(n):
            if not visited[v] and graph[u][v] != math.inf:
                if dist[u] + graph[u][v] < dist[v]:
                    dist[v] = dist[u] + graph[u][v]
    return dist
```

INPUT:

```
# Test Case 1
graph1 = [
    [0, 10, 3, math.inf, math.inf],
    [math.inf, 0, 1, 2, math.inf],
    [math.inf, 4, 0, 8, 2],
    [math.inf, math.inf, math.inf, 0, 7],
    [math.inf, math.inf, math.inf, 9, 0]]
print(dijkstra(graph1, 0))
# Test Case 2
graph2 = [
    [0, 5, math.inf, 10],
    [math.inf, 0, 3, math.inf],
    [math.inf, math.inf, 0, 1],
    [math.inf, math.inf, math.inf, 0]]
print(dijkstra(graph2, 0))
```

OUTPUT:

```
[0, 7, 3, 9, 5]
[0, 5, 8, 9]
```

6. **Given a graph represented by an edge list, implement Dijkstra's Algorithm to find the shortest path from a given source vertex to a target vertex. The graph is represented as a list of edges where each edge is a tuple (u, v, w) representing an edge from vertex u to vertex v with weight w.**

AIM:

To find the **shortest path from a source vertex to a target vertex** in a weighted graph represented as an edge list using **Dijkstra's Algorithm**.

ALGORITHM:

1. **Convert edge list to adjacency list**:
    - For each edge (u, v, w) add (v, w) to adj[u].
2. **Initialize distances**:
    - Set dist[] for all vertices as infinity, except source (dist[source] = 0).
    - Use a **priority queue** (min-heap) to pick the vertex with the smallest distance.
3. **Process vertices**:
    - Pop vertex u from the queue.
    - For each neighbor (v, w) of u, if dist[u] + w < dist[v], update dist[v] and push (dist[v], v) into the queue.
4. **Stop when target is reached** (optional optimization).
5. **Return** dist[target] as the shortest path length.

CODE:

```
import heapq
import math
def dijkstra_edge_list(n, edges, source, target):
    # Create adjacency list
    adj = [[] for _ in range(n)]
    for u, v, w in edges:
        adj[u].append((v, w))
        adj[v].append((u, w))  # If graph is undirected
    dist = [math.inf] * n
    dist[source] = 0
    pq = [(0, source)]  # (distance, vertex)
    while pq:
        d, u = heapq.heappop(pq)
        if u == target:
            return dist[target]
        if d > dist[u]:
            continue
        for v, w in adj[u]:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                heapq.heappush(pq, (dist[v], v))
    return dist[target]
```

INPUT: **OUTPUT:**

```
# Test Case 1
n1 = 6
edges1 = [(0, 1, 7), (0, 2, 9), (0, 5, 14), (1, 2, 10), (1, 3, 15),
      (2, 3, 11), (2, 5, 2), (3, 4, 6), (4, 5, 9)]
source1 = 0
target1 = 4
print(dijkstra_edge_list(n1, edges1, source1, target1))

# Test Case 2
n2 = 5
edges2 = [(0, 1, 10), (0, 4, 3), (1, 2, 2), (1, 4, 4), (2, 3, 9), (3, 2, 7),
      (4, 1, 1), (4, 2, 8), (4, 3, 2)]
source2 = 0
target2 = 3
print(dijkstra_edge_list(n2, edges2, source2, target2))
```

```
20

5
```

7. Given a set of characters and their corresponding frequencies, construct the Huffman Tree and generate the Huffman Codes for each character.

AIM:

To construct a **Huffman Tree** for a set of characters with given frequencies and generate the **Huffman Codes** for each character.

ALGORITHM:

1. **Create leaf nodes** for each character and its frequency.
2. **Insert all nodes into a priority queue** (min-heap) based on frequency.
3. **Repeat until only one node remains**:
    - Extract two nodes with the smallest frequencies.
    - Create a new internal node with frequency = sum of the two nodes.
    - Set the two extracted nodes as left and right children of the new node.
    - Insert the new node back into the priority queue.
4. **The remaining node is the root of the Huffman Tree**.
5. **Traverse the tree** to generate codes:
    - Assign '0' for left edges and '1' for right edges.
    - The code for a character is the path from the root to its leaf.

CODE:

```
import heapq
class Node:
    def __init__(self, char=None, freq=0):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None
    def __lt__(self, other):
        return self.freq < other.freq
def huffman_codes(characters, frequencies):
    heap = [Node(c, f) for c, f in zip(characters, frequencies)]
    heapq.heapify(heap)
    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(freq=left.freq + right.freq)
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)
    root = heap[0]
    codes = {}
    def generate_codes(node, current_code):
        if node is None:
            return
        if node.char is not None:
            codes[node.char] = current_code
        generate_codes(node.left, current_code + '0')
        generate_codes(node.right, current_code + '1')
    generate_codes(root, "")
    return sorted(codes.items())
```

INPUT:

```
# Test Case 1
characters1 = ['a', 'b', 'c', 'd']
frequencies1 = [5, 9, 12, 13]
print(huffman_codes(characters1, frequencies1))

# Test Case 2
characters2 = ['f', 'e', 'd', 'c', 'b', 'a']
frequencies2 = [5, 9, 12, 13, 16, 45]
print(huffman_codes(characters2, frequencies2))
```

OUTPUT:

```
[('a', '00'), ('b', '01'), ('c', '10'), ('d', '11')]
[('a', '0'), ('b', '111'), ('c', '101'), ('d', '100'), ('e', '1101'), ('f', '1100')]
```

8. Given a Huffman Tree and a Huffman encoded string, decode the string to get the original message.

AIM:

To decode a given Huffman encoded string using the Huffman Tree constructed from characters and their frequencies.

ALGORITHM:

1. **Construct the Huffman Tree** from the given characters and frequencies (same as encoding).
2. **Start from the root of the tree** and read the encoded string bit by bit:
    - '0' → move to the left child.
    - '1' → move to the right child.
3. **When a leaf node is reached**, record its character as part of the decoded message and return to the root.
4. **Repeat until the entire encoded string is processed.**

CODE:

```
import heapq
class Node:
    def __init__(self, char=None, freq=0):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None
    def __lt__(self, other):
        return self.freq < other.freq
def build_huffman_tree(characters, frequencies):
    heap = [Node(c, f) for c, f in zip(characters, frequencies)]
    heapq.heapify(heap)
    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(freq=left.freq + right.freq)
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)
    return heap[0]  # root
def decode_huffman(root, encoded_string):
    decoded = []
    node = root
    for bit in encoded_string:
        node = node.left if bit == '0' else node.right
        if node.char is not None:
            decoded.append(node.char)
            node = root
    return ''.join(decoded)
```

INPUT:

```
# Test Case 1
characters1 = ['a', 'b', 'c', 'd']
frequencies1 = [5, 9, 12, 13]
encoded_string1 = '1101100111110'
root1 = build_huffman_tree(characters1, frequencies1)
print(decode_huffman(root1, encoded_string1))

# Test Case 2
characters2 = ['f', 'e', 'd', 'c', 'b', 'a']
frequencies2 = [5, 9, 12, 13, 16, 45]
encoded_string2 = '1100110111001011111001011'
root2 = build_huffman_tree(characters2, frequencies2)
print(decode_huffman(root2, encoded_string2))
```

OUTPUT:

dbcbdd

fefcbaac

9. Given a list of item weights and the maximum capacity of a container, determine the maximum weight that can be loaded into the container using a greedy approach. The greedy approach should prioritize loading heavier items first until the container reaches its capacity.

AIM:

To determine the maximum weight that can be loaded into a container using a greedy approach, prioritizing heavier items first until the container reaches its maximum capacity.

ALGORITHM:

1. **Sort the list of item weights in descending order**.
2. Initialize a variable current_weight = 0 to keep track of the total loaded weight.
3. **Iterate through the sorted weights**:
    - If adding the current item does not exceed max_capacity, add it to current_weight.
    - Otherwise, skip the item.
4. Return current_weight as the maximum weight that can be loaded.

CODE:

```
def max_loaded_weight(weights, max_capacity):
    weights.sort(reverse=True)  # sort in descending order
    current_weight = 0

    for weight in weights:
        if current_weight + weight <= max_capacity:
            current_weight += weight

    return current_weight
```

INPUT:

```
# Test Case 1
weights1 = [10, 20, 30, 40, 50]
max_capacity1 = 60
print(max_loaded_weight(weights1, max_capacity1))

# Test Case 2
weights2 = [5, 10, 15, 20, 25, 30]
max_capacity2 = 50
print(max_loaded_weight(weights2, max_capacity2))
```

OUTPUT:

```
60
50
```

10. Given a list of item weights and a maximum capacity for each container, determine the minimum number of containers required to load all items using a greedy approach. The greedy approach should prioritize loading items into the current container until it is full before moving to the next container.

AIM:

To determine the minimum number of containers required to load all items using a greedy approach, filling each container to its maximum capacity before moving to the next one.

ALGORITHM:

1. **Sort the item weights in descending order** (heaviest items first).
2. Initialize a variable containers = 0 to count the containers.
3. Initialize a variable i = 0 to iterate over the weights.

4. **While i < n** (items remain):
    - Initialize current_capacity = 0 for the current container.
    - **Fill the container greedily**:
        - Add items to current_capacity until adding another item exceeds max_capacity.
        - Move to the next item when full or cannot add without exceeding capacity.
    - Increment containers by 1.

Return containers as the minimum number of containers required.

CODE:

```
def min_containers(weights, max_capacity):
    weights.sort(reverse=True)  # sort in descending order
    containers = 0
    n = len(weights)
    used = [False] * n  # track loaded items

    for i in range(n):
        if used[i]:
            continue
        current_capacity = weights[i]
        used[i] = True
        for j in range(i + 1, n):
            if not used[j] and current_capacity + weights[j] <= max_capacity:
                current_capacity += weights[j]
                used[j] = True
        containers += 1

    return containers
```

INPUT:

```
# Test Case 1
weights1 = [5, 10, 15, 20, 25, 30, 35]
max_capacity1 = 50
print(min_containers(weights1, max_capacity1))

# Test Case 2
weights2 = [10, 20, 30, 40, 50, 60, 70, 80]
max_capacity2 = 100
print(min_containers(weights2, max_capacity2))
```

OUTPUT:

```
3
4
```

11. Given a graph represented by an edge list, implement Kruskal's Algorithm to find the Minimum Spanning Tree (MST) and its total weight.

AIM:

To implement **Kruskal's Algorithm** to find the Minimum Spanning Tree (MST) of a given weighted, undirected graph and calculate its total weight.

ALGORITHM:

1. **Sort all edges** of the graph in ascending order of their weights.
2. Initialize a **Union-Find (Disjoint Set)** data structure to detect cycles.
3. Initialize an empty list mst_edges to store edges of the MST.
4. **Iterate through the sorted edges**:
    - For each edge (u, v, w), check if u and v belong to the same set (i.e., would adding this edge form a cycle?).
    - If **not in the same set**, add the edge to mst_edges and **union** the sets of u and v.
    - If they are in the same set, skip the edge.
5. Stop when **MST has n-1 edges** (n = number of vertices).
6. Compute the **total weight** by summing the weights of edges in mst_edges.
7. Return the MST edges and total weight.

CODE:

```
class DisjointSet:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0]*n
    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])  # path compression
        return self.parent[x]
    def union(self, x, y):
        xroot = self.find(x)
        yroot = self.find(y)
        if xroot == yroot:
            return False
        if self.rank[xroot] < self.rank[yroot]:
            self.parent[xroot] = yroot
        elif self.rank[xroot] > self.rank[yroot]:
            self.parent[yroot] = xroot
        else:
            self.parent[yroot] = xroot
            self.rank[xroot] += 1
        return True
def kruskal(n, edges):
    edges.sort(key=lambda x: x[2])  # sort by weight
    ds = DisjointSet(n)
    mst_edges = []
    total_weight = 0
    for u, v, w in edges:
        if ds.union(u, v):
            mst_edges.append((u, v, w))
            total_weight += w
        if len(mst_edges) == n-1:
            break
    return mst_edges, total_weight
```

INPUT:

```
# Test Case 1
n1 = 4
edges1 = [ (0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4) ]
mst1, total1 = kruskal(n1, edges1)
print("Edges in MST:", mst1)
print("Total weight of MST:", total1)

# Test Case 2
n2 = 5
edges2 = [ (0, 1, 2), (0, 3, 6), (1, 2, 3), (1, 3, 8), (1, 4, 5), (2, 4, 7), (3, 4, 9) ]
mst2, total2 = kruskal(n2, edges2)
print("Edges in MST:", mst2)
print("Total weight of MST:", total2)
```

OUTPUT:

```
Edges in MST: [(2, 3, 4), (0, 3, 5), (0, 1, 10)]
Total weight of MST: 19
Edges in MST: [(0, 1, 2), (1, 2, 3), (1, 4, 5), (0, 3, 6)]
Total weight of MST: 16
```

12. Given a graph with weights and a potential Minimum Spanning Tree (MST), verify if the given MST is unique. If it is not unique, provide another possible MST.

AIM:

To verify if a given Minimum Spanning Tree (MST) for a weighted graph is **unique**, and if not, provide another possible MST.

ALGORITHM:

1. **Compute the MST weight** of the given graph using Kruskal's or Prim's algorithm.
2. **Compare** the total weight of the given MST with the computed MST weight:
    - If the weights are different, the given MST is invalid.
    - If weights are equal, continue to check uniqueness.
3. **Check for uniqueness**:
    - Sort edges by weight.
    - Construct the MST using Kruskal's algorithm.
    - If there exists another edge with the same weight that can replace an edge in MST **without changing the total weight**, then the MST is **not unique**.
4. **Return**:
    - True if the MST is unique.
    - False and another possible MST if it's not unique.

CODE:

```
class DisjointSet:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0]*n
    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):
        xroot = self.find(x)
        yroot = self.find(y)
        if xroot == yroot:
            return False
        if self.rank[xroot] < self.rank[yroot]:
            self.parent[xroot] = yroot
        elif self.rank[xroot] > self.rank[yroot]:
            self.parent[yroot] = xroot
        else:
            self.parent[yroot] = xroot
            self.rank[xroot] += 1
        return True
def kruskal(n, edges):
    edges.sort(key=lambda x: x[2])
    ds = DisjointSet(n)
    mst_edges = []
    total_weight = 0
    for u, v, w in edges:
        if ds.union(u, v):
            mst_edges.append((u, v, w))
            total_weight += w
        if len(mst_edges) == n-1:
            break
    return mst_edges, total_weight

def is_mst_unique(n, edges, given_mst):
    given_weight = sum([w for u,v,w in given_mst])
    mst_edges, mst_weight = kruskal(n, edges)
    if mst_weight != given_weight:
        return False, None
    from itertools import permutations
    weight_groups = {}
    for u,v,w in edges:
        weight_groups.setdefault(w, []).append((u,v))
    alternative_msts = []
    def dfs(curr_edges, ds, remaining_edges):
        if len(curr_edges) == n-1:
            alternative_msts.append(list(curr_edges))
            return
        for i, (u,v,w) in enumerate(remaining_edges):
            if ds.find(u) != ds.find(v):
                ds2 = DisjointSet(n)
                ds2.parent = ds.parent[:]
                ds2.rank = ds.rank[:]
                ds2.union(u,v)
                dfs(curr_edges + [(u,v,w)], ds2, remaining_edges[i+1:])
    ds = DisjointSet(n)
    dfs([], ds, edges)
    unique = all(sorted(mst) == sorted(given_mst) for mst in alternative_msts)
    # If not unique, return another MST
    for mst in alternative_msts:
        if sorted(mst) != sorted(given_mst):
            return False, mst
    return True, None
```

INPUT:

```
# Test Case 1
n1 = 4
edges1 = [ (0, 1, 10), (0, 2, 6), (0, 3, 5), (1, 3, 15), (2, 3, 4) ]
given_mst1 = [(2, 3, 4), (0, 3, 5), (0, 1, 10)]
unique1, alt1 = is_mst_unique(n1, edges1, given_mst1)
print("Is the given MST unique?", unique1)
if alt1: print("Another possible MST:", alt1)

# Test Case 2
n2 = 5
edges2 = [ (0, 1, 1), (0, 2, 1), (1, 3, 2), (2, 3, 2), (3, 4, 3), (4, 2, 3) ]
given_mst2 = [(0, 1, 1), (0, 2, 1), (1, 3, 2), (3, 4, 3)]
unique2, alt2 = is_mst_unique(n2, edges2, given_mst2)
print("Is the given MST unique?", unique2)
if alt2: print("Another possible MST:", alt2)
```

OUTPUT:

```
Is the given MST unique? False
Another possible MST: [(2, 3, 4), (0, 3, 5), (1, 3, 15)]
Is the given MST unique? False
Another possible MST: [(0, 1, 1), (0, 2, 1), (1, 3, 2), (4, 2, 3)]
```