Kruskal's algorithm is a classic algorithm used to find the **Minimum Spanning Tree (MST)** of a graph. The goal is to connect all vertices with the minimum possible total edge weight, without forming any cycles.

Part (a): Algorithms Needed to Find MST Using Kruskal's Algorithm

Kruskal's Algorithm works by sorting all the edges of the graph in non-decreasing order of their weights and then adding edges to the MST, ensuring no cycles are formed.

Here are the key algorithms and steps needed:

1. Union-Find Data Structure (Disjoint Set Union)

Kruskal's algorithm requires efficient data structures for checking whether two vertices are in the same connected component, which is done using the **Union-Find** (also called **Disjoint Set Union**, DSU) structure. This data structure supports two operations efficiently:

- **Find**: Determines which component a particular element is in.
- **Union**: Merges two components into one.

Union-Find Data Structure with Path Compression and Union by Rank

These optimizations ensure that both operations can be done in nearly constant time (amortized time complexity of $O(\alpha(n))$, where α is the inverse Ackermann function, which grows very slowly).

1. **Find Operation**:
    - Traverse the parent pointer recursively until we find the root of the set.
    - **Path Compression**: After finding the root, we update the parent pointers along the way to point directly to the root.
2. **Union Operation**:
    - Merge two sets by attaching the smaller set's root to the larger set's root.
    - **Union by Rank**: This ensures that the tree representing the disjoint sets remains shallow by attaching the smaller tree to the root of the larger tree.

python

Copy code

```python
class DisjointSet:
    def __init__(self, n):
        self.parent = list(range(n))  # Initially, each node is its own parent
        self.rank = [0] * n  # Initially, all trees have rank 0


    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):
        rootX = self.find(x)
        rootY = self.find(y)

        if rootX != rootY:
            # Union by rank
            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
```

2. Kruskal’s Algorithm for MST

Once we have the Union-Find data structure, Kruskal’s algorithm works by following these steps:

1. **Sort** all the edges in the graph in increasing order of their weights.
2. Initialize an empty list for the **MST**.
3. **Iterate** over the sorted edges:
    - For each edge, check if adding the edge to the MST forms a cycle using the **Union-Find** data structure.
    - If it doesn't form a cycle, add the edge to the MST and **union** the two vertices.
4. **Stop** when the MST contains n-1 edges, where n is the number of vertices.

python

Copy code

```
def kruskal(n, edges):
```

```
# Initialize the disjoint set
ds = DisjointSet(n)
mst = []  # To store the edges in the MST

# Step 1: Sort edges by weight
edges.sort(key=lambda x: x[2])  # Sort by edge weight (x[2] is the weight)

# Step 2: Iterate over the edges
for u, v, weight in edges:
  if ds.find(u) != ds.find(v):
    ds.union(u, v)
    mst.append((u, v, weight))  # Add the edge to MST
  if len(mst) == n - 1:  # If we've added n-1 edges, we're done
    break

return mst
```

- **Input**:
  - n: The number of vertices in the graph.
  - edges: A list of edges in the form of (u, v, weight) where u and v are the vertices connected by the edge and weight is the weight of the edge.
- **Output**: A list of edges that make up the MST.

Part (b): Algorithm Analysis

Let's break down the time complexity of each part of the algorithm:

1. **Sorting the edges**:
   - Sorting E edges takes **O(E log E)** time.
2. **Union-Find operations**:
   - The find and union operations are optimized using **path compression** and **union by rank**. Each operation takes **O(α(n))** time, where α(n) is the inverse Ackermann function, which grows extremely slowly.

- In the worst case, each edge requires two find operations and one union operation, so this part of the algorithm takes **O(E α(n))** time.

3. **Total Time Complexity**:
   - The total time complexity of Kruskal's algorithm is dominated by the sorting step, i.e., **O(E log E)**.
   - If E is large, this is the bottleneck of the algorithm. The Union-Find operations add an almost constant factor due to path compression and union by rank.

Thus, the overall time complexity of Kruskal's algorithm is:

mathematica

Copy code

$O(E \log E + E\,\alpha(n)) \approx O(E \log E)$

- **Space Complexity**:
  - We need to store the graph edges, which takes **O(E)** space.
  - The Union-Find data structure uses **O(V)** space, where V is the number of vertices.
  - Therefore, the total space complexity is **O(V + E)**.