Minimal Cost Spanning Trees

Roll No: 220001043, 220001044, 220001045

The term "minimal" is used as the mst may not be unique.
Let G be an undirected graph.

1. G is connected.
2. G is acyclic.
3. G has exactly $n-1$ edges.

If any of the two statements hold, then the third statement holds.
To prove statement 3 from statements 1 and 2, consider the following:

- **Statement 1:** G has at least $n-1$ edges.

  Proof. Basis: A graph with a single vertex clearly has at least 0 edges(Trivial).

  Inductive Hypothesis: A connected graph with k vertices have at least $k-1$ edges for any positive integer k

  Induction Step: Now, we should show that the inductive hypothesis is true for k+1 vertices.

  Let G be a connected graph with $k+1$ vertices and let v be a vertex of G. Remove v and all incident edges to v from G. The resulting graph, which we will call G_0, may or may not be connected, but we do know it has k vertices. Suppose G_0 has s components where $1 \leq s \leq k$. Each component is a connected graph with k or fewer vertices, so we may apply the inductive hypothesis to each component. For each component, if the component has q vertices then it has at least $q-1$ edges. Thus G_0 must have at least $k-s$ edges. When v was removed from G we only removed edges that connect v to obtain G_0. For each component in G_0 there must be an edge from v to a vertex in that component since G was connected. Thus there must be at least s more edges in G than there are in G_0, i.e., the number of edges in G is at least $(k-s)+s=k$.

  Therefore, from the induction step, we can say that G has atleast n-1 edges.

- **Statement 2:** G has at most $n-1$ edges.

  Proof: Assume for contradiction that there exists a graph with n vertices and n edges that does not contain a cycle. Then, we can pick any start vertex and move to one of its neighbors via an edge. We can assume that without loss of generality every vertex has at least one incident edge via pigeonhole principle (n vertices, n edges), therefore every vertex has at least one neighbor. Since we moved to a neighboring vertex from our start vertex via one edge we have now connected two vertices but only used one edge. We can move to a neighbor of one of the visited vertices by another incident edge or visit a new vertex and move to its neighbor (thus also connecting two vertices but using only one edge) and inductively continue this process until we have visited all n vertices of the graph. It is trivial to see that we will only have used at max $n-1$ edges.. But since the graph has n edges there must exist another edge which we have not used. The only possibility for such an edge to exist is if it connects to a vertex that has already been visited. Therefore this n-th edge must complete a cycle and therefore the graph is cyclic. We have thus reached a contradiction. Hence, the graph G should have atmost n-1 edges.

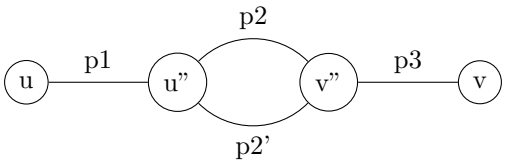
  Since G has at least $n-1$ edges and at most $n-1$ edges, it follows that G has exactly $n-1$ edges.

Therefore, statement 3 holds true.

Suppose G is a graph with n vertices that is connected, acyclic, and undirected.

Statement: If all three conditions hold, there exists exactly one path between any two vertices u and v in G.

Proof: Let u and v be any two vertices in G. Since G is connected, there exists at least one path between u and v. We aim to show that there cannot be more than one path between u and v.



Assume, for contradiction, that there are two distinct paths between u and v, denoted by P($p_1p_2p_3$) and Q($p_1p_2'p_3$) respectively. As, the two paths are not equivalent, they must diverge at a particular vertex, say u"(may be u). And as, these two paths end at a particular node v, they must converge after they diverge. Let the convergence vertex be v"(may be v). Since G is connected, there must be a path between u'' and v''. However, there cannot be two disjoint paths between u'' and v'' as it forms a cycle, contradicting the assumption that G is acyclic. Therefore, there cannot be two distinct paths between u and v.

Conclusion: Since there cannot be more than one path between any two vertices in G, there exists exactly one path between any two vertices u and v in G.

Algorithms for creating MSTs

In the graph G, there will be a weight associated with the edges. Hence, we can form a minimum spanning tree (MST) by using those edge weights. Two popular algorithms for finding MSTs are Prim's algorithm and Kruskal's algorithm. These two algorithms follow greedy approach.

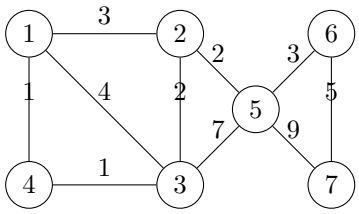
1. **Prim's Algorithm:** This algorithm starts from an arbitrary vertex and grows the MST by adding the shortest edge that connects the current MST to a vertex outside the MST until all vertices are included.
2. **Kruskal's Algorithm:** This algorithm initially treats each vertex as a separate component and iteratively adds the shortest edge that connects two different components and does not form a cycle until all vertices are part of a single connected component, forming the MST.

Kruskal's Algorithm

1. Initialize an empty set T to represent the minimum spanning tree.
2. Sort all the edges in non-decreasing order of their weights.
3. For each edge e in the sorted list:
   (a) If adding e to T does not create a cycle:
      i. Add e to T.
4. Return the set T as the minimum spanning tree.

Steps and Graphs

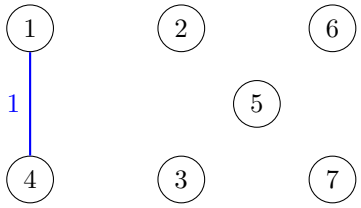
1. Initial graph:



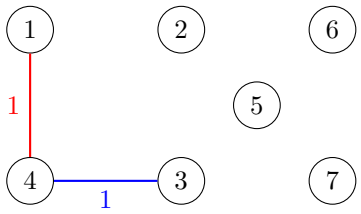
2. After sorting edges:

Edge	Weight
(1,4)	1
(3,4)	1
(2,3)	2
(2,5)	2
(5,6)	3
(1,2)	3
(1,3)	4
(6,7)	5
(3,5)	7
(5,7)	9

3. Adding edges to form the minimum spanning tree:

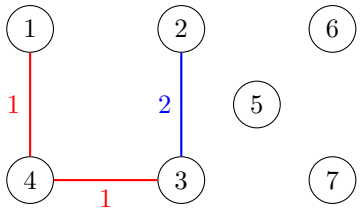


After adding edge (1,4), there is no cycle formed as this edge does not connect back to any of the previously added vertices (1), hence it's safe to add this edge.



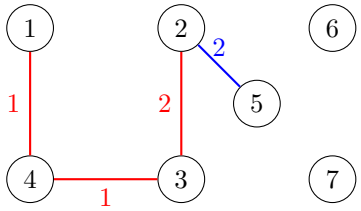
After adding edge (3,4), there is no cycle formed as this edge does not connect back to any of the previously added vertices (1), hence it's safe to add this edge.

4. After adding edge (2,3):



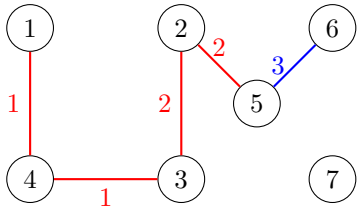
After adding edge (2,3), there is no cycle formed as this edge does not connect back to any of the previously added vertices (1,4), hence it's safe to add this edge.

5. After adding edge (2,5):



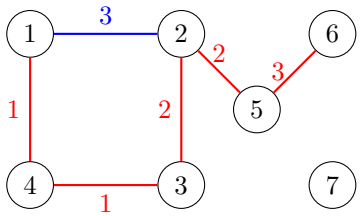
After adding edge (2,5), there is no cycle formed as this edge does not connect back to any of the previously added vertices (1,4,3), hence it's safe to add this edge.

6. After adding edge (5,6):



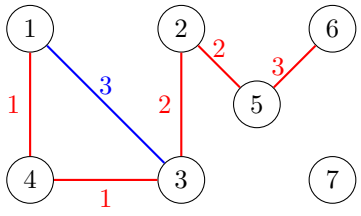
After adding edge (5,6), there is no cycle formed as this edge does not connect back to any of the previously added vertices (1,4,3,2,5), hence it's safe to add this edge.

7. After adding edge (1,2):



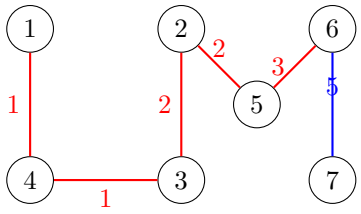
After adding edge (1,2), a cycle is formed as this edge connects back to vertex 1, which is part of the previously added vertices (1,4). Hence, it's not safe to add this edge.

8. After adding edge (1,3):



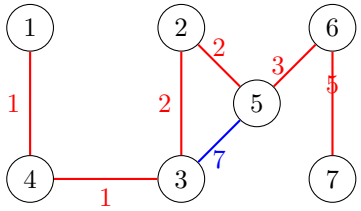
After adding edge (1,3), a cycle is formed as this edge connects back to vertex 1, which is part of the previously added vertices (1,4). Hence, it's not safe to add this edge.

9. After adding edge (6,7):



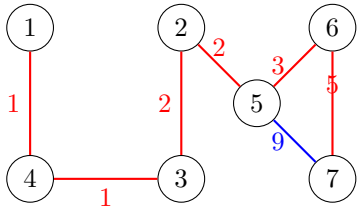
After adding edge (6,7), there is no cycle formed as this edge does not connect back to any of the previously added vertices (1,4,2,3,5), hence it's safe to add this edge.

10. After adding edge (3,5):



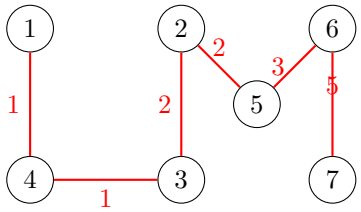
After adding edge (3,5), a cycle is formed as this edge connects back to vertex 3, which is part of the previously added vertices (1,4,2,3). Hence, it's not safe to add this edge.

11. After adding edge (5,7):



After adding edge (5,7), a cycle is formed as this edge connects back to vertex 5, which is part of the previously added vertices (1,4,2,3,5). Hence, it's not safe to add this edge.

12. Final graph: Minimum Spanning Tree formed by Kruskal's algorithm



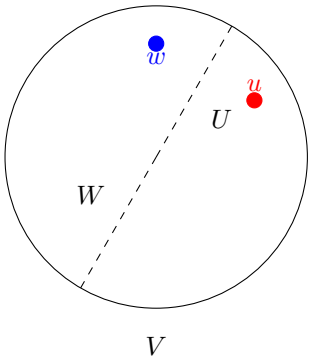
- Using both Prim's and Kruskal's algorithms, we can construct a minimum spanning tree.
- Both algorithms are greedy approaches; hence, we may not obtain a globally optimal solution.

Minimum Separator Lemma

Assumption: All the edges in the graph have unique weights.

Statement:

Let $U \subseteq V$ and $W = V \setminus U$. Let (u, w) be the minimum cost edge that connects U and W, where $u \in U$ and $w \in W$. Then, the minimum cost edge between $u \in U$ and $w \in W$ will be part of every minimum spanning tree (MST).



Proof:

Assume T is a minimum spanning tree (MST) of G that does not contain the edge (u, w). Since T spans all vertices in G, it must contain a path from u to w.

Without loss of generality, let's assume the path crosses from u to w and let $U \subseteq V$ and $W = V \setminus U$.

Since T does not contain the edge (u, w), it must contain at least one other edge, say (u', w'), where $u' \in U$ and $w' \in W$.

Consider the cycle formed by adding the edge (u, w) to the path in T. This cycle contains the edge (u', w'), and here's the key point:

Because of the unique weight assumption, if (u, w) has the minimum weight across the cut separating U and W, then the weight of (u', w') must be strictly greater than the weight of (u, w).

Therefore, replacing (u', w') with (u, w) in the cycle creates a new spanning tree with a lower total weight. This contradicts the assumption that T is a minimum spanning tree.

Therefore, any minimum spanning tree T of G must contain the edge (u, w), where $w \in V \setminus \{u\}$. Hence, the minimum weight edge between any vertices separated by a cut must be part of every minimum spanning tree.

Correctness of Prim's Algorithm

To prove the correctness of Prim's algorithm using the Minimal Separator Lemma, let's consider the following:

Assumption: All the edges in the graph have unique weights.

Proof:

Prim's algorithm grows the minimum spanning tree (MST) by adding the shortest edge that connects the current MST to a vertex outside the MST until all vertices are included.

By the Minimal Separator Lemma, the minimum cost edge between any two sets of vertices separated by a cut must be part of every minimum spanning tree. Since Prim's algorithm always selects the

shortest edge that connects the current MST to a vertex outside the MST, it will always choose the minimum cost edge between any two sets of vertices separated by a cut. Therefore, Prim's algorithm is correct.

Correctness of Kruskal's Algorithm

To prove the correctness of Kruskal's algorithm using the Minimal Separator Lemma, let's consider the following:

Assumption: All the edges in the graph have unique weights.

Proof:

Kruskal's algorithm initially treats each vertex as a separate component and iteratively adds the shortest edge that connects two different components until all vertices are part of a single connected component, forming the MST.

By the Minimal Separator Lemma, the minimum cost edge between any two sets of vertices separated by a cut must be part of every minimum spanning tree.

Now, let's suppose we have an edge $e_{ij} = (u_i, w_j)$ where $u_i \in U$ and $u_j \in W$. This edge e_{ij} is used to connect the current set represented by U to the remaining vertices represented by W. We choose e_{ij} because it is the minimum cost edge that connects U to W, as proved by the Minimal Separator Lemma.

Since Kruskal's algorithm selects edges in non-decreasing order of weight, and e_{ij} is the minimum cost edge connecting U to W, Kruskal's algorithm will select e_{ij} at some stage of its execution. Therefore, Kruskal's algorithm will include the minimum cost edge between U and W in the MST, ensuring that the MST is formed correctly.

Hence, Kruskal's algorithm is correct.

Implementation of Prim's Algorithm

Prim's algorithm constructs the minimum spanning tree (MST) by starting from an arbitrary vertex and iteratively adding the shortest edge that connects the current MST to a vertex outside the MST until all vertices are included.

Variable Definitions:

- TV: Set containing the vertices seen so far.
- TE: Set containing the edges seen so far.
- n: Number of vertices in the graph.
- $visited[i]$: Boolean array to mark whether vertex i has been visited.
- $d[i]$: Array to store the distance between vertex i and the vertices in TV.
- $N[i]$: Array to store the neighbor of vertex i in TV that has the shortest distance.

Prim's Algorithm Implementation:

```
function MSTPrim
for i = 1 to n
    visited[i] = 0
    d[i] = infinity    // distance between i-th node to TV
    N[i] = -1         // neighbor of i-th node with TV
```

```
TV = {1}
visited[1] = 1
for (1, v) in E
    d[v] = w[1, v]
    N[v] = 1
// Create a priority queue with respect to d[v]
priority_queue pq
for i = 1 to n
    pq.push({d[i], i})

// Main loop
while !pq.empty()
    u = pq.top().second
    pq.pop()
    TV = TV U {u}   // Add u to the set of visited vertices
    if N[u] != -1
        TE = TE U {(u, N[u])}   // Add edge (u, N[u]) to the MST
    visited[u] = 1

    // Update distances and neighbors
    for (u, v) in E
        if !visited[v] and w[u,v] < d[v]
            d[v] = w[u,v]
            N[v] = u
            // Update the priority queue
            pq.push({d[v], v})
```

This implementation of Prim's algorithm constructs the minimum spanning tree by maintaining a set of visited vertices (TV) and a set of edges in the MST (TE), and updating the distances and neighbors of each vertex as it iterates through the vertices of the graph.