[CS203] Design and Analysis of Algorithms

Course Instructor: Dr. Dibyendu Roy — Autumn 2023-24
Scribed by: Prakhar Jain(202251099) — Lecture (Week 12)

Introduction to Kruskal's Algorithm

1. **Objective:** Finds a minimal spanning tree for a connected, undirected graph.
2. **Greedy Approach:** Operates based on a greedy strategy, adding the edge with the lowest weight at each step.
3. **Safe Edge Selection:** Selects a safe edge by finding the edge with the lowest weight that connects two different trees in the growing forest.
4. **Connected Components:** Utilizes a disjoint-set data structure to maintain connected components (trees). Uses FIND-SET operation to determine whether two vertices belong to the same tree.
5. **Union Operation:** Merges two trees into one using the UNION procedure.
6. **Algorithm Steps:**
   (a) Initializes an empty set and creates individual trees for each vertex.
   (b) Examines edges in ascending order of weight.
   (c) Checks whether the edge's endpoints belong to the same tree.
   (d) If not, adds the edge to the set and merges the corresponding trees.
7. **Running Time:** Depends on the specific implementation of the disjoint-set data structure. Assumes disjoint-set-forest implementation with union-by-rank and path-compression heuristics. Time complexity is $O(E \log V)$, where E is the number of edges and V is the number of vertices.
8. **Algorithm Analysis:**
   (a) Initializes set and creates trees in $O(1)$ time.
   (b) Sorting edges takes $O(E \log E)$ time.
   (c) Disjoint-set operations take $O(E \log V)$ time.
   (d) Overall time complexity is $O(E \log V)$.
9. **Conclusion:** Kruskal's algorithm is a greedy approach that efficiently finds a minimal spanning tree by iteratively adding the lowest-weight edges. This ensures that the resulting spanning tree connects all vertices with the minimum possible total edge weight.

MST-KRUSKAL(G, w)

$A \leftarrow \emptyset$
for each vertex $v \in G.V$ **do**
    MAKE-SET(v)
end for
Create a single list of edges in $G.E$
Sort the list of edges into monotonically increasing order by weight w
for each edge (u, v) taken from the sorted list in order **do**
    if FIND-SET$(u) \neq$ FIND-SET(v) **then**
        $A \leftarrow A \cup \{(u, v)\}$
        UNION(u, v)
    end if
end for
return A

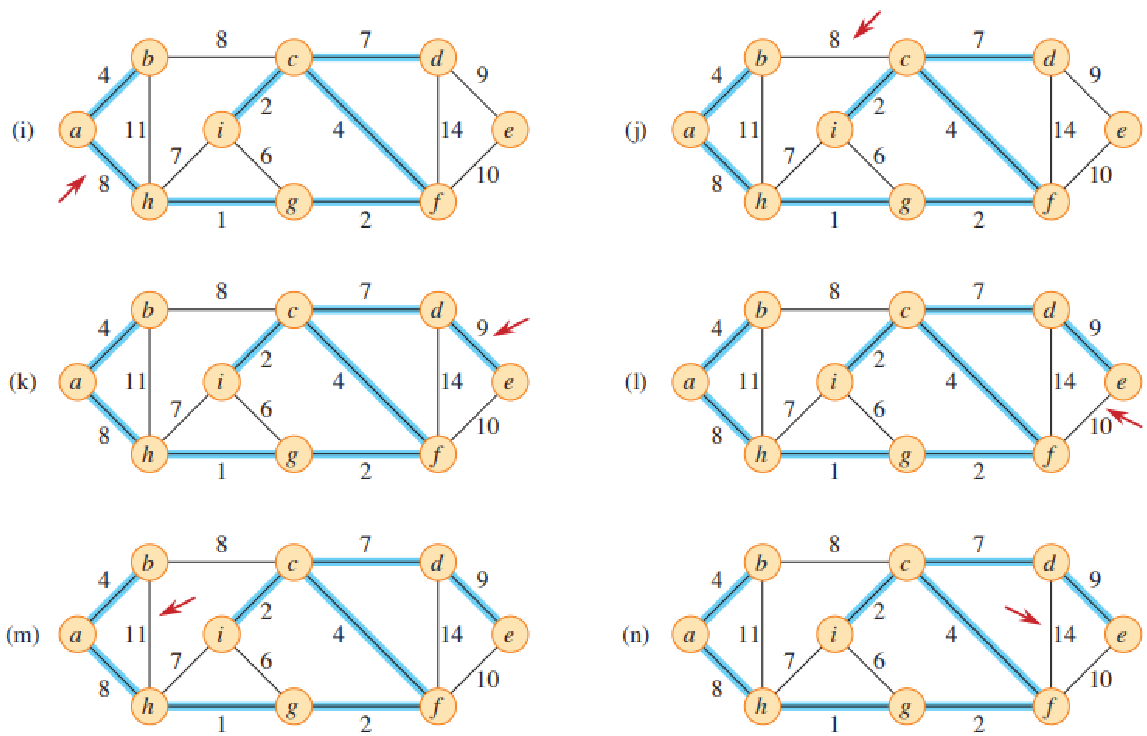

Figure 1: The execution of Kruskals algorithm on the graph

Prim's Algorithm - Key Points

Algorithm Overview:

- Prim's algorithm is a greedy algorithm for finding a minimum spanning tree in a connected, undirected graph.
- It starts from an arbitrary root vertex and grows the minimum spanning tree until it spans all vertices.

Tree Growing Strategy:

- The edges in the set A always form a single tree throughout the algorithm's execution.
- At each step, a light edge connecting A to an isolated vertex (one on which no edge of A is incident) is added.

Greedy Nature:

- The algorithm adds edges to the tree with the minimum possible weight at each step.
- This ensures that the edges added contribute the minimum amount possible to the tree's weight.

Procedure Description (MST-PRIM):

1. Maintain a min-priority queue Q of vertices not in the tree, based on a key attribute.
2. Attributes include *key* (minimum weight of any edge connecting the vertex to the tree) and $v : \pi$ (parent of the vertex in the tree).
3. The algorithm implicitly maintains the set A as $A = \{(v, v : \pi) \mid v \in V \setminus Q\}$.

Loop Invariant:

- Prior to each iteration of the loop, A is correctly maintained as described above.
- Vertices in $V \setminus Q$ are already in the minimum spanning tree.
- For vertices $v \in Q$, if $v : \pi \neq \text{NIL}$, then $v : key < 1$ and $v : key$ is the weight of a light edge $(v, v : \pi)$.

Edge Selection and Update:

- Identify a vertex $u \in Q$ incident on a light edge that crosses the cut $(V \setminus Q, Q)$.
- Add u to the tree $(V \setminus Q)$, adding the edge $(u, u : \pi)$ to A.
- Update key and π attributes of vertices adjacent to u but not in the tree.
- DECREASE-KEY is called to inform the min-priority queue of the change in key.

Prim's Algorithm

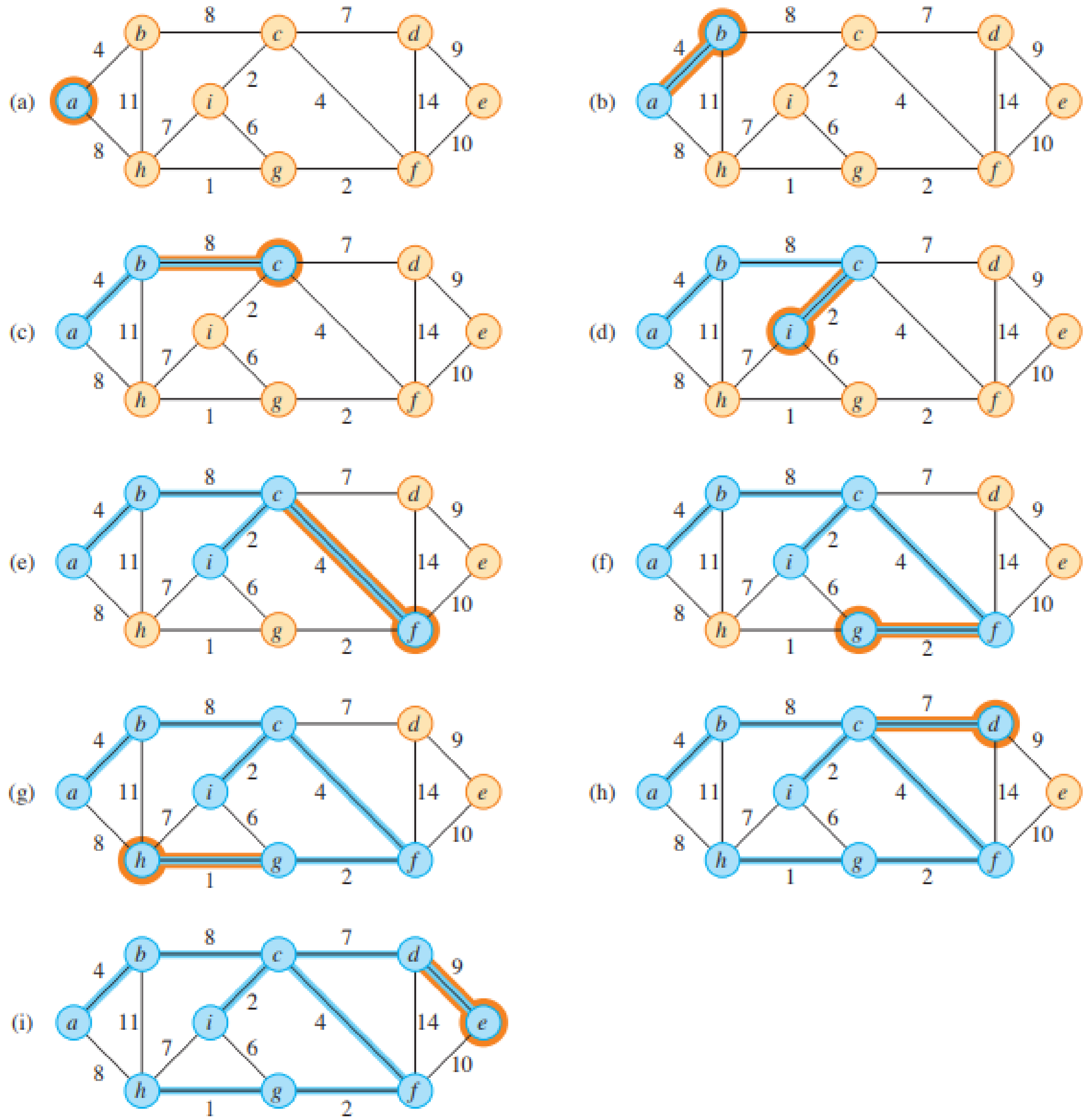

Figure 2: The execution of Prims algorithm on the graph

Input: Graph $G = (V, E)$, edge weights $w : E \to \mathbb{R}$, and a root vertex $r \in V$.

Output: Minimum Spanning Tree A.

```
procedure MST-PRIM(G, w, r)
    for each vertex u ∈ G : V do
        u : key ← 1
        u : π ← NIL
    end for
    r : key ← 0
    Q ← {}
    for each vertex u ∈ G : V do
        INSERT(Q, u)
    end for
    while Q ≠ {} do
        u ← EXTRACT-MIN(Q)                          ▷ Add u to the tree
        for each vertex v ∈ G : Adj(u) do
            if v ∈ Q  and  w(u, v) < v : key then
                v : π ← u
                v : key ← w(u, v)
                DECREASE-KEY(Q, v, w(u, v))
            end if
        end for
    end while
    return A
end procedure
```

Shortest Path Algorithm using BFS

Input: Graph G and a source vertex $start$.

Output: Shortest distances from $start$ to all other vertices.

```
procedure ShortestPathBFS(G, start)
    dist ← Dictionary to store shortest distances from start
    visited ← Set to keep track of visited vertices
    queue ← Queue data structure
    for each vertex v in G.vertices do
        dist[v] ← ∞
    end for
    dist[start] ← 0
    visited.add(start)
    queue.enqueue(start)
    while queue is not empty do
        u ← queue.dequeue()
        for each neighbor v of u do
            if v is not in visited then
                visited.add(v)
                dist[v] ← dist[u] + 1                ▷ Assuming unweighted edges
                queue.enqueue(v)
            end if
        end for
    end while
    return dist                                      ▷ Shortest distances from start
end procedure
```

Bellman-Ford Algorithm for Shortest Paths with Negative Cycle Detection

Input: Graph G with edge weights and a source vertex $start$.

Output: Shortest paths from $start$ to all other vertices or indication of a negative cycle.

```
procedure BellmanFord(G, start)
    dist ← Dictionary to store shortest distances from start
    prev ← Dictionary to store previous vertices in the shortest paths
    for each vertex v in G.vertices do
        dist[v] ← ∞
        prev[v] ← None
    end for
    for i ← 1 to |G.vertices| − 1 do
        for each edge (u, v) in G.edges do
            if dist[u] + weight(u, v) < dist[v] then
                dist[v] ← dist[u] + weight(u, v)
                prev[v] ← u
            end if
        end for
    end for
    for each edge (u, v) in G.edges do
        if dist[u] + weight(u, v) < dist[v] then
            return "Negative cycle detected"
        end if
    end for
    return dist, prev                    ▷ Shortest distances and previous vertices
end procedure
```

Dijkstra's Algorithm for Shortest Paths

Input: Graph G with non-negative edge weights and a source vertex *start*.

Output: Shortest paths from *start* to all other vertices.

```
procedure DIJKSTRA(G, start)
    dist ← Dictionary to store shortest distances from start
    prev ← Dictionary to store previous vertices in the shortest paths
    Q ← Priority queue
    for each vertex v in G.vertices do
        dist[v] ← ∞
        prev[v] ← None
        Q.insert(v, dist[v])
    end for
    dist[start] ← 0
    while Q is not empty do
        u ← Q.extractMin()
        for each vertex v adjacent to u do
            if Q.contains(v) and dist[u] + weight(u, v) < dist[v] then
                dist[v] ← dist[u] + weight(u, v)
                prev[v] ← u
                Q.decreaseKey(v, dist[v])
            end if
        end for
    end while
    return dist, prev                    ▷ Shortest distances and previous vertices
end procedure
```