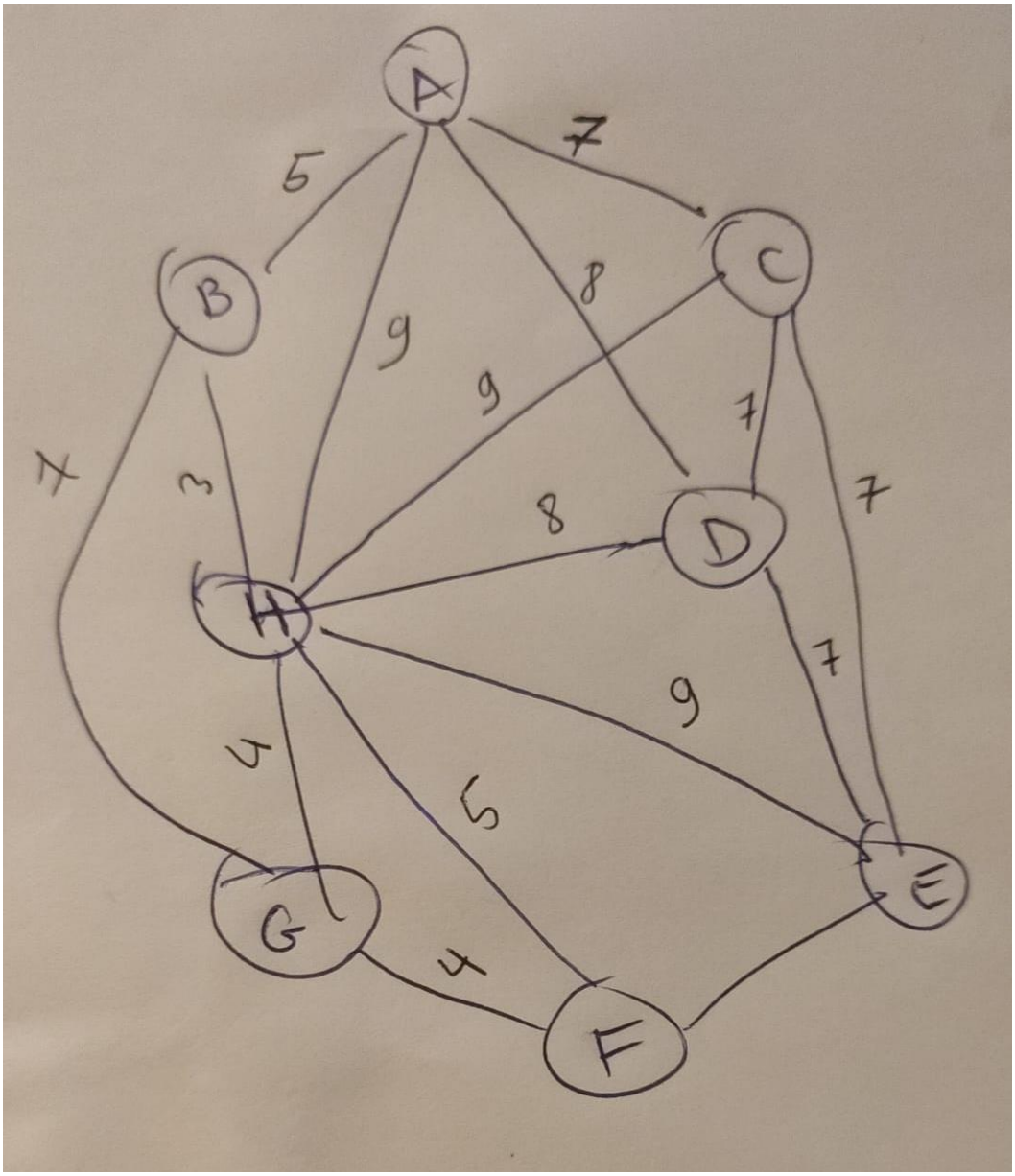


To apply the Dijkstra algorithm starting from node **A** and ending at node **F** for the graph,

1. Graph Details:

- The graph is represented by nodes A, B, C, D, E, F, G, H.
- Edge weights:
  - A → B: 5
  - A → C: 7
  - A → D: 9
  - B → H: 3

- C → D: 7
- C → E: 7
- D → E: 9
- D → H: 8
- E → F: 7
- H → F: 5
- H → G: 4
- G → F: 4

2. Step-by-Step Dijkstra Analysis (Starting at A, Destination F):

Initialization:

- Set the distance for the starting node (A) as 0, and all other nodes as infinity (∞).
- The "previous node" for each node is set to undefined.

Node	Distance	Previous Node
A	0	-
B	∞	-
C	∞	-
D	∞	-
E	∞	-
F	∞	-
G	∞	-
H	∞	-

Step 1: Starting at A :

- For each neighbor of A, update distances:
  - A → B = 5 (Update B's distance)
  - A → C = 7 (Update C's distance)
  - A → D = 9 (Update D's distance)

Node	Distance	Previous Node
A	0	-
B	5	A
C	7	A
D	9	A
E	∞	-
F	∞	-
G	∞	-
H	∞	-

Step 2: Visit B (smallest distance)

- Explore B's neighbors:
    - B → H = 3, total distance to H is 5 (from A to B) + 3 = 8 (Update H's distance)

Node	Distance	Previous Node
A	0	-
B	5	A
C	7	A
D	9	A
E	∞	-
F	∞	-
G	∞	-
H	8	B

Step 3: Visit C (next smallest distance)

- Explore C's neighbors:
    - C → D = 7, total distance to D is 7 + 7 = 14 (no update because current distance to D is shorter at 9)
    - C → E = 7, total distance to E is 7 + 7 = 14 (Update E's distance)

Node	Distance	Previous Node
A	0	-
B	5	A
C	7	A
D	9	A
E	14	C
F	∞	-
G	∞	-
H	8	B

Step 4: Visit H (next smallest distance)

- Explore H's neighbors:
    - H → F = 5, total distance to F is 8 + 5 = 13 (Update F's distance)
    - H → G = 4, total distance to G is 8 + 4 = 12 (Update G's distance)

Node	Distance	Previous Node
A	0	-
B	5	A
C	7	A
D	9	A
E	14	C
F	13	H
G	12	H
H	8	B

Step 5: Visit D :

- Explore D's neighbors:
    - D → E = 9, total distance to E is 9 + 9 = 18 (no update because current distance to E is shorter at 14)

- D → H = 8, total distance to H is 9 + 8 = 17 (no update because current distance to H is shorter at 8)

Node Distance Previous Node

A	0	-
B	5	A
C	7	A
D	9	A
E	14	C
F	13	H
G	12	H
H	8	B

Step 6: Visit G

- Explore G's neighbors:
  - G → F = 4, total distance to F is 12 + 4 = 16 (no update because current distance to F is shorter at 13)

Node Distance Previous Node

A	0	-
B	5	A
C	7	A
D	9	A
E	14	C
F	13	H
G	12	H
H	8	B

Step 7: Visit F (destination reached)

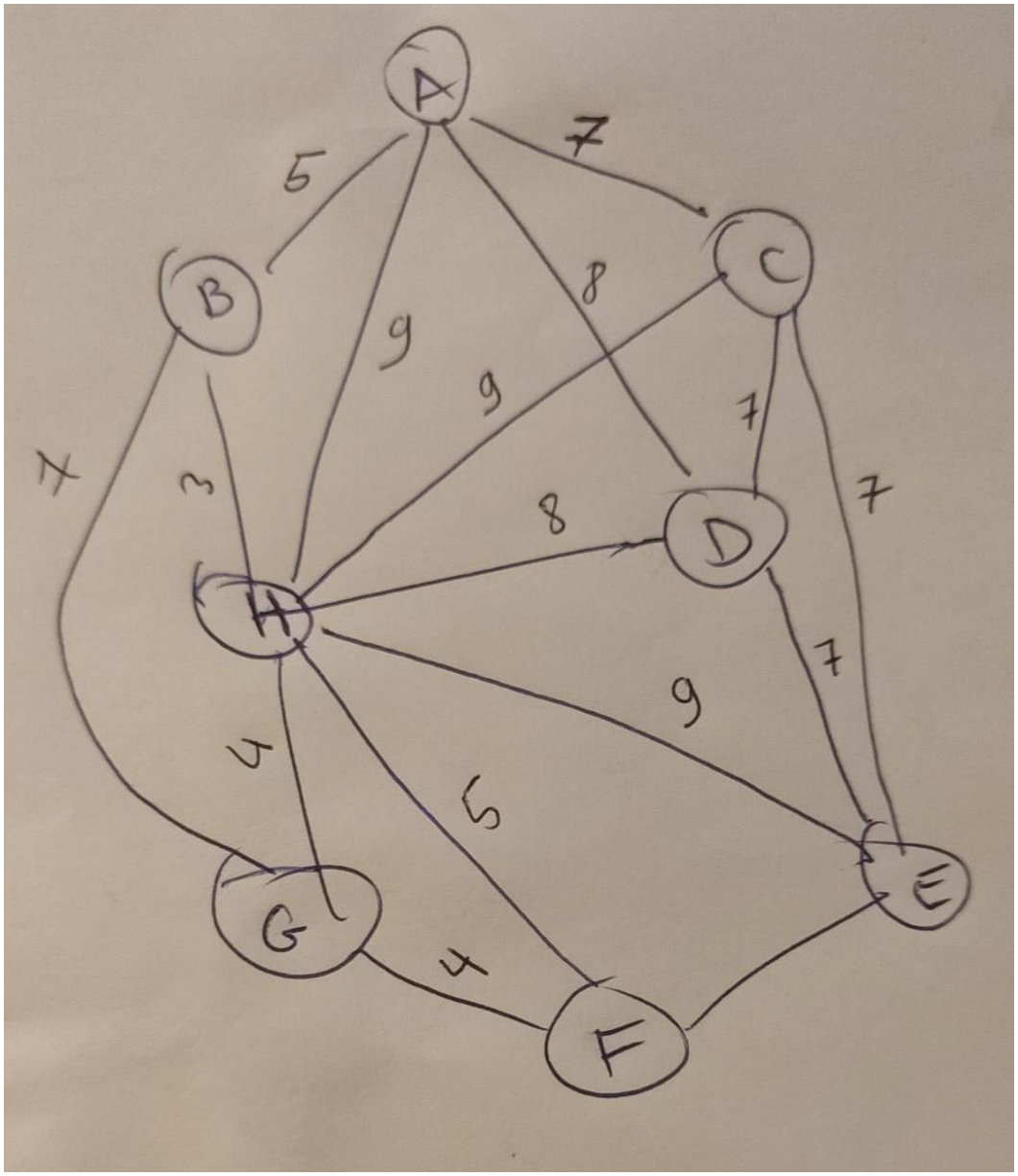
- Distance to F from A = **13**

- Shortest path: **A → B → H → F**

3. Dijkstra Algorithm Pseudocode:

```
function Dijkstra(graph, source):
  # Initialization
  dist = {}
  prev = {}
  for each vertex v in graph:
    dist[v] = infinity   // Set all distances to infinity
    prev[v] = undefined    //No previous node yet
  dist[source] = 0         // Distance to source is 0

  //Priority queue to store nodes to visit
  Q = set of all vertices in graph

  while Q is not empty:
    // Get the node with the smallest distance
    u = node in Q with smallest dist[u]
    Q.remove(u)

    # Check neighbors of u
    for each neighbor v of u:
      alt = dist[u] + length(u, v)  // Calculate the distance to v
      if alt < dist[v]:           // If a shorter path is found
        dist[v] = alt           // Update the shortest distance
        prev[v] = u             // Update the previous node

  return dist, prev
```

Assignment 14b – Minimum Spanning Tree



the execution of Prim-Jarvik's algorithm on this graph, I'll walk through each step using the visual of the graph you've provided.

Prim's Algorithm Explanation:

1. **Choose a starting vertex**. I'll start with vertex A as an example.
2. **Add the smallest edge that connects a vertex in the tree to a vertex outside the tree**.
3. **Repeat** this step until all vertices are included in the tree.

Execution of Prim's Algorithm on the Graph:

Step 1: Starting at vertex A

- Available edges from A:
  A-B (weight 5), A-H (weight 9), A-C (weight 7)
- Choose the smallest edge: **A-B (weight 5)**

Current tree: A - B

Step 2: Add the smallest edge connected to A or B

- Available edges from A and B:
  A-H (weight 9), A-C (weight 7), B-H (weight 3)
- Choose the smallest edge: **B-H (weight 3)**

Current tree: A - B - H

Step 3: Add the smallest edge connected to A, B, or H

- Available edges from A, B, H:
  A-H (weight 9), A-C (weight 7), H-C (weight 8), H-D (weight 9), H-G (weight 4), H-F (weight 5)
- Choose the smallest edge: **H-G (weight 4)**

Current tree: A - B - H - G

Step 4: Add the smallest edge connected to A, B, H, or G

- Available edges from A, B, H, G:
  A-H (weight 9), A-C (weight 7), H-C (weight 8), H-D (weight 9), H-F (weight 5), G-F (weight 4)
- Choose the smallest edge: **G-F (weight 4)**

Current tree: A - B - H - G - F

Step 5: Add the smallest edge connected to A, B, H, G, or F

- Available edges from A, B, H, G, F:
  A-H (weight 9), A-C (weight 7), H-C (weight 8), H-D (weight 9), H-F (weight 5), F-E (weight 5), D-F (weight 8)
- Choose the smallest edge: **F-E (weight 5)**

Current tree: A - B - H - G - F - E

Step 6: Add the smallest edge connected to A, B, H, G, F, or E

- Available edges from A, B, H, G, F, E:
  A-H (weight 9), A-C (weight 7), H-C (weight 8), H-D (weight 9), D-F (weight 8), D-E (weight 7)
- Choose the smallest edge: **A-C (weight 7)**

Current tree: A - B - H - G - F - E - C

Step 7: Add the last remaining vertex

- The last remaining vertex is D.
- Available edges:
  H-D (weight 9), D-F (weight 8), D-E (weight 7), H-C (weight 8)
- Choose the smallest edge: **D-E (weight 7)**

Final tree: A - B - H - G - F - E - C - D

Resulting Minimum Spanning Tree (MST):

- The MST includes the edges:
  - A-B (weight 5)
  - B-H (weight 3)
  - H-G (weight 4)
  - G-F (weight 4)
  - F-E (weight 5)
  - A-C (weight 7)
  - D-E (weight 7)

The total weight of the Minimum Spanning Tree = 5 + 3 + 4 + 4 + 5 + 7 + 7 = **35**.

R-7-9 Repeat the previous problem for Baruvka's algorithm.

1. Consider the following potential MST algorithms based on the generic MST algorithm. Which, if any, successfully computes a MST? Hint: to show that an algorithm does not compute an MST, all you need to do is find a counterexample. If it does, you need to argue why based on the cycle property and/or the partition property.

1. Algorithm MST-a

This algorithm starts with all edges sorted in *nonincreasing* order (from the highest weight to the lowest). It then removes each edge, one by one, if removing that edge keeps the graph connected.

analyse
The approach contradicts the **cycle property** of MSTs, which states that in any cycle, the edge with the maximum weight can be removed to minimize the total weight. Since the algorithm starts with the largest edges and works downwards, it does the reverse of what is expected. It will likely remove smaller edges and keep larger edges, which is incorrect for finding an MST.

- **Counterexample**: Consider a triangle with edges of weights 1, 2, and 3. MST-a would start with the edge of weight 3, then consider removing edges of weights 2 and 1, keeping the edge with weight 3, which results in a higher total weight.

Conclusion: This algorithm does **not** compute an MST.

---

2. Algorithm MST-b

This algorithm adds edges arbitrarily, provided they do not form a cycle. It adds the edge if the union of the current tree and the edge does not result in a cycle.

- **Analysis**:
  This is essentially **Kruskal's Algorithm**, which is a known correct algorithm for computing the MST. As long as edges are added in non-decreasing order of weights, the algorithm will generate the MST by following the **cycle property** and **partition property**.

Conclusion: This algorithm **does** compute an MST, provided the edges are considered in non-decreasing order (as implied by Kruskal's algorithm).

---

3. Algorithm MST-c

This algorithm adds edges arbitrarily and removes an edge if it forms a cycle, removing the edge with the maximum weight in the cycle.

- **Analysis**:
  This approach works similarly to **Kruskal's Algorithm** with a slight variation. It

ensures that the highest-weight edge in any cycle is removed. By ensuring cycles are broken using the heaviest edge, the algorithm follows the **cycle property** of MSTs. This process will eventually build a valid MST, as long as cycles are managed correctly.

Conclusion: This algorithm **does** compute an MST.

The result :

- **MST-a**: Does **not** compute an MST.
- **MST-b**: Computes an MST (similar to Kruskal's algorithm).
- **MST-c**: Computes an MST (cycle-breaking strategy based on the maximum weight).

Problem A: Modified BFS for Edge Counting

```
function modifiedBFS(graph, startVertex)
  queue = new Queue()
  distance = [0] * |V|  // Initialize distances to 0

  queue.enqueue(startVertex)
  distance[startVertex] = 0

  while queue is not empty
    currentVertex = queue.dequeue()

    for neighbor in graph[currentVertex]
      if distance[neighbor] == 0
        distance[neighbor] = distance[currentVertex] + 1
        queue.enqueue(neighbor)
```

```
  return distance
```

Problem B: Checking for Tree Structure

```
function isTree(graph, edgeSequence)
  visited = [false] * |V|
  parent = [-1] * |V|

  for edge in edgeSequence
    u, v = edge.endpoints

    if visited[u] and visited[v]:
      return false  // Cycle detected

    if not visited[u]:
      visited[u] = true
      parent[u] = v

    if not visited[v]:
      visited[v] = true
      parent[v] = u

  // Check if all vertices are reachable
  for i in 1 to |V|
    if not visited[i]:
      return false

  return true
```

Problem C: Desert Crossing

```
function minStops(distances, capacity)
```

```
n = len(distances)
dp = [0] * n

for i in 1 to n - 1
   dp[i] = float('inf')
   for j in 0 to i - 1
      if distances[i] - distances[j] <= capacity:
         dp[i] = min(dp[i], dp[j] + 1)

return dp[n - 1]
```