ANALYSIS AND DESIGN OF ALGORITHMS

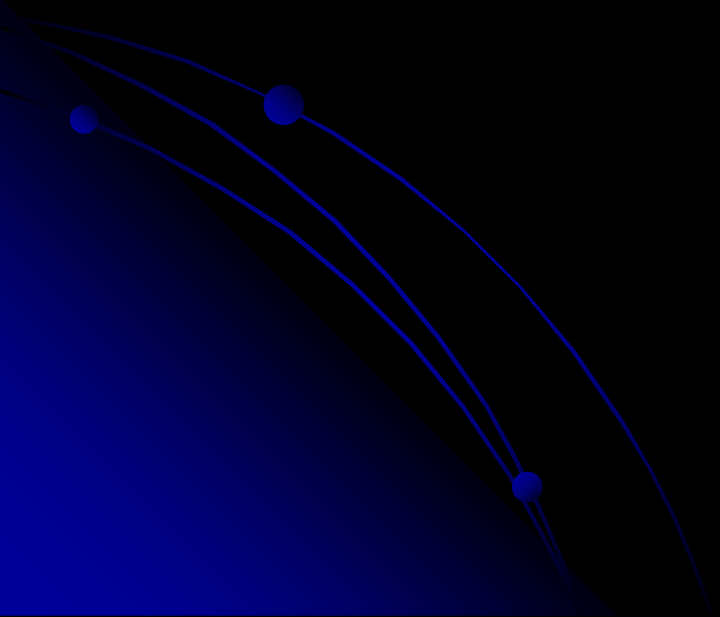
UNIT-IV

CHAPTER 9:

GREEDY TECHNIQUE

OUTLINE

- **Greedy Technique**
  - Prim's Algorithm
  - Kruskal's Algorithm
  - Dijkstra's Algorithm



Minimum Spanning Tree

Definition: A spanning tree of a connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph. A minimum spanning tree of a weighted connected graph is its spanning tree of smallest weight, where the weight of a tree is defined as the sum of the weights on all its edges. The ***minimum spanning tree problem is the problem of finding a minimum spanning tree for a given weighted connected graph.***

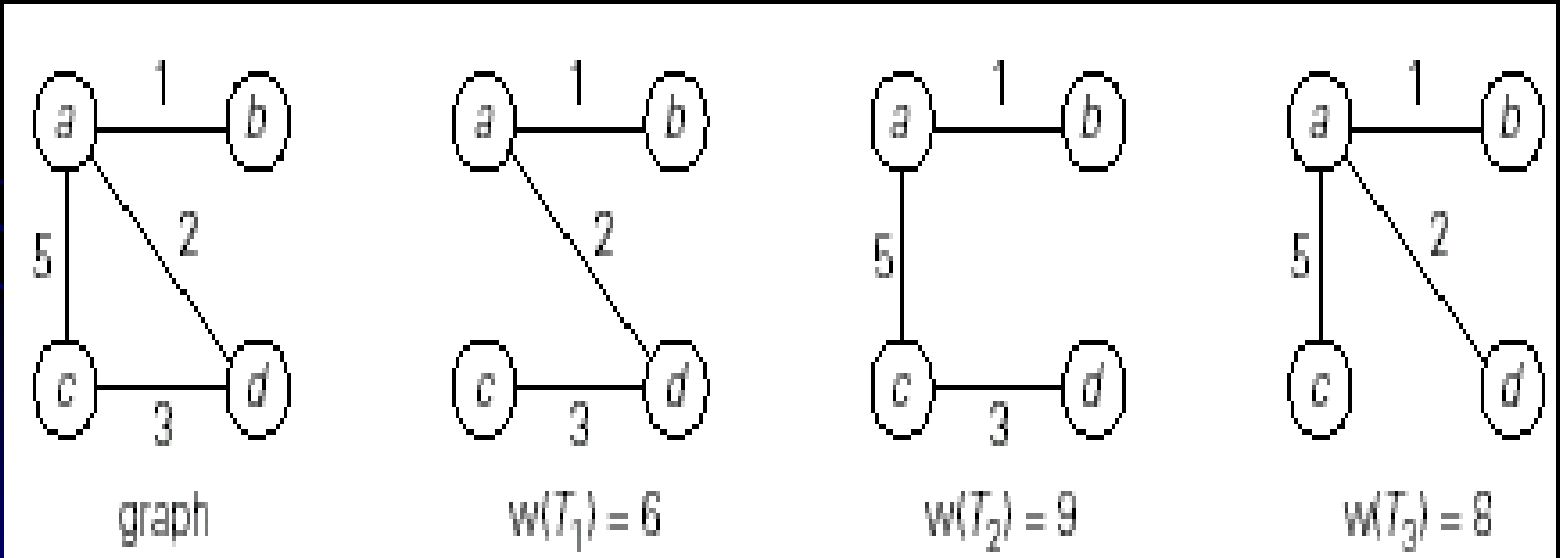


Figure: Graph and its spanning trees: T_1 is the minimum spanning tree.

GREEDY APPROACH

- The **greedy approach** suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. On each step—and this is the central point of this technique—the choice made must be
  - *feasible*, i.e., it has to satisfy the problem's constraints.
  - *locally optimal*, i.e., it has to be the best local choice among all feasible choices available on that step.
  - *irrevocable*, i.e., once made, it cannot be changed on subsequent steps of the algorithm.

PRIM'S ALGORITHM

- Prim's algorithm constructs a minimum spanning tree through a sequence of expanding subtrees.
- The initial subtree in such a sequence consists of a single vertex selected arbitrarily from the set V of the graph's vertices.
- On each iteration, we expand the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree.
- The algorithm stops after all the graph's vertices have been included in the tree being constructed.
- Since the algorithm expands a tree by exactly one vertex on each of its iterations, the total number of such iterations is n-1, where n is the number of vertices in the graph.

PRIM'S ALGORITHM

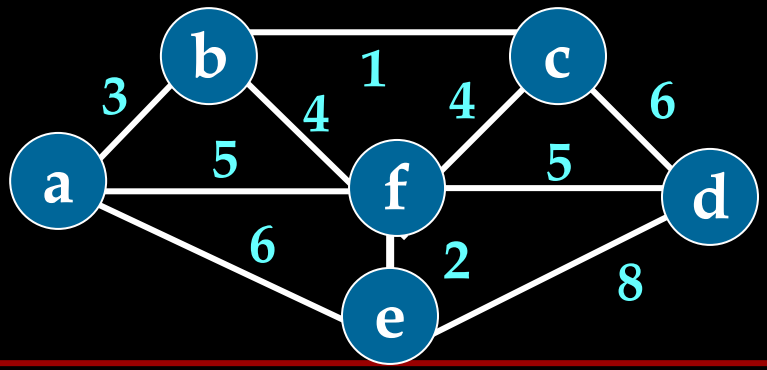
ALGORITHM *Prim*(G)

//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex
$E_T \leftarrow \emptyset$
for $i \leftarrow 1$ **to** $|V| - 1$ **do**
    find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u)
    such that v is in V_T and u is in $V - V_T$
    $V_T \leftarrow V_T \cup \{u^*\}$
    $E_T \leftarrow E_T \cup \{e^*\}$
return E_T

PRIM'S ALGORITHM

- The nature of Prim's algorithm makes it necessary to provide each vertex not in the current tree with ***the information about the shortest edge connecting the vertex to a tree vertex***.
- We can provide such information by **attaching two labels** to a vertex: the **name of the nearest tree vertex** and the **weight** (length) of the corresponding edge.
- Vertices that are not adjacent to any of the tree vertices can be given the ∞ **label** and a **null label** for the name of the nearest tree vertex.
- We can split the vertices that are not in the tree into two sets, the "**fringe**" and the "**unseen**".
  - The **fringe** contains only the vertices that are not in the tree but are adjacent to at least one tree vertex. These are the candidates from which the next tree vertex is selected.
  - The **unseen** vertices are all other vertices of the graph.

Figure: Application of Prim's Algorithm. The parenthesized labels of a vertex in the middle column indicate the nearest tree vertex and edge weight; selected vertices and edges are shown in bold.



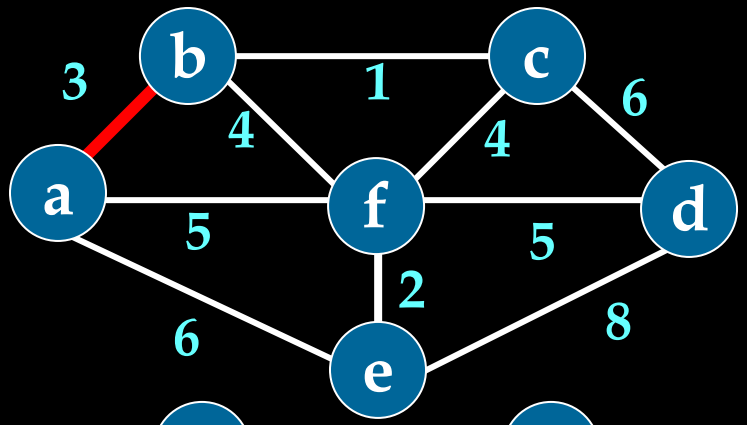
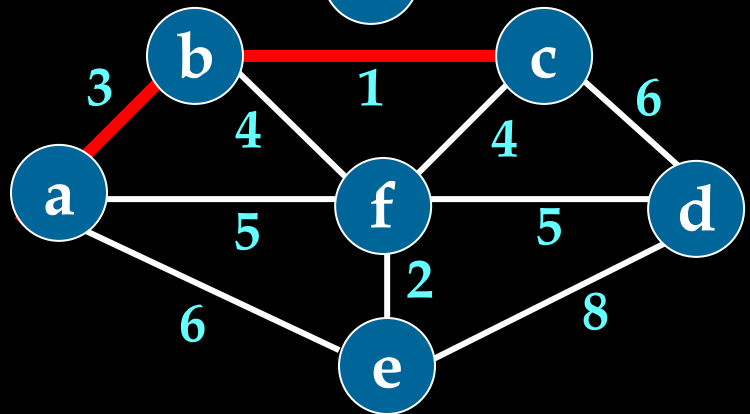
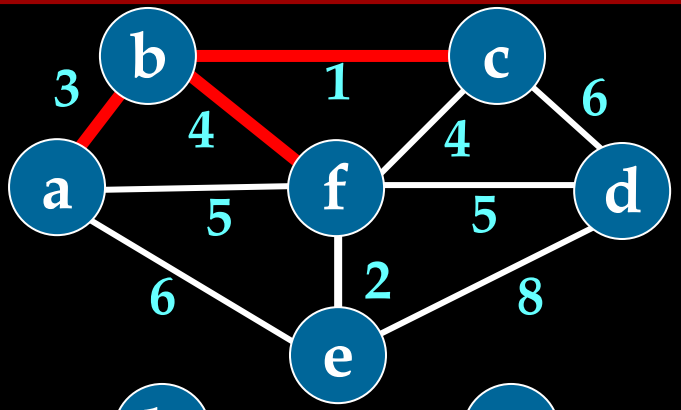
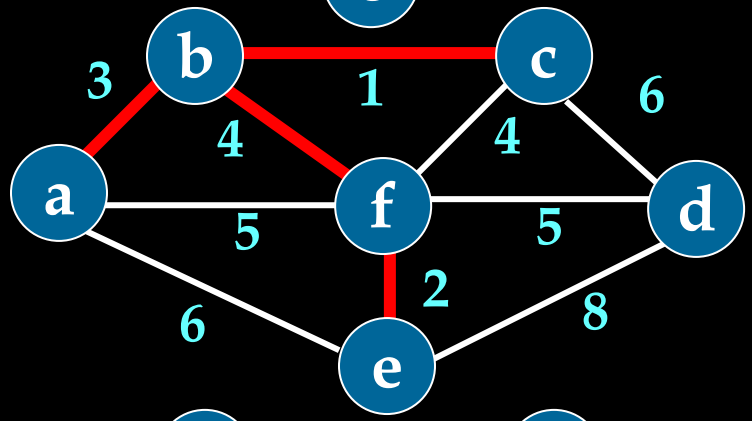
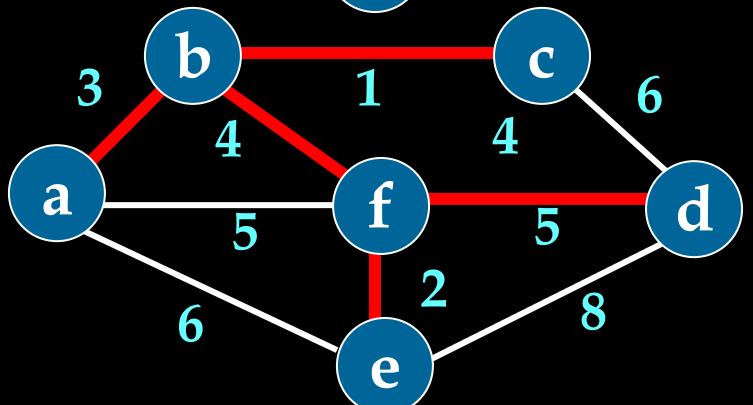
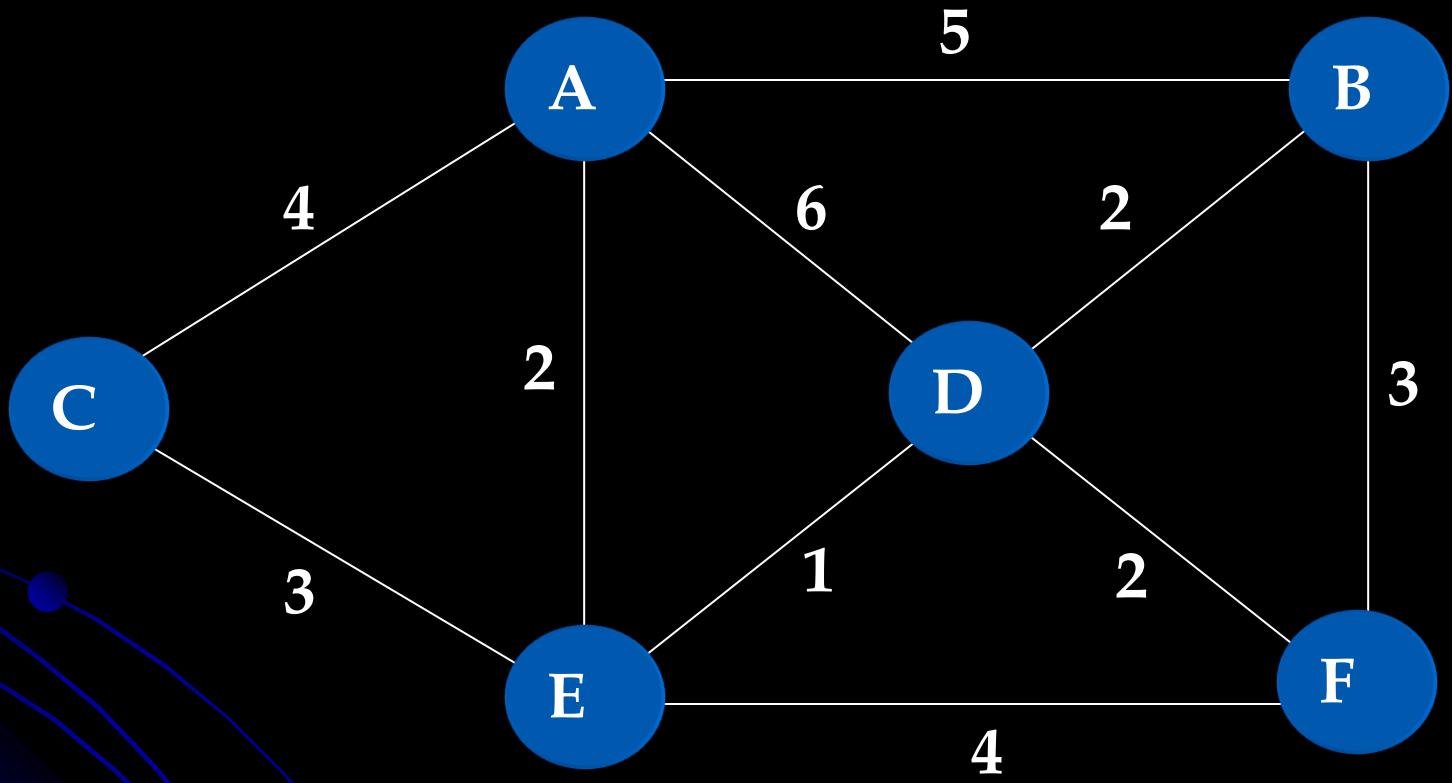
Tree vertices	Remaining vertices	Illustration
a(-, -)	**b(a, 3)** c(-, ∞) d(-, ∞) e(a, 6) f(a, 5)	
b(a, 3)	**c(b, 1)** **d(-, ∞)** e(a, 6) f(b, 4)	

Figure: Application of Prim's Algorithm. The parenthesized labels of a vertex in the middle column indicate the nearest tree vertex and edge weight; selected vertices and edges are shown in bold.

Tree vertices	Remaining vertices	Illustration
c(b, 1)	**d(c, 6)** e(a, 6) **f(b, 4)**	
f(b, 4)	**d(f, 5)** **e(f, 2)**	
e(f, 2)	**d(f, 5)**	
d(f, 5)		

PRIM'S ALGORITHM PROBLEM



PRIM'S ALGORITHM ANALYSIS

- Efficiency of Prim's algorithm depends on the data structures chosen for the graph.
- If a graph is represented by its weight matrix, the algorithm's running time will be in $\theta(|V|^2)$.
- If a graph is represented by its adjacency linked lists, the running time of the algorithm is in $O(|E| \log |V|)$.

KRUSKAL'S ALGORITHM

- This is another greedy algorithm for the minimum spanning tree problem that also always yields an optimal solution.
- It is named Kruskal's algorithm, after Joseph Kruskal, who discovered the algorithm.
- Kruskal's algorithm looks at a minimum spanning tree for a weighted connected graph G={V, E} as an acyclic subgraph with |V| - 1 edges for which the sum of the edge weights is the smallest.

KRUSKAL'S ALGORITHM

- The algorithm constructs a minimum spanning tree as an expanding sequence of subgraphs, which are always acyclic but are not necessarily connected on the intermediate stages of the algorithm.

- The algorithm begins by sorting the graph's edges in nondecreasing order of their weights.

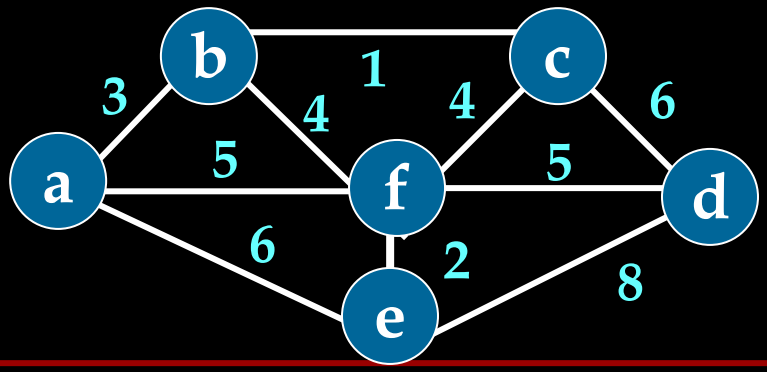
- Then, starting with the empty subgraph, it scans this sorted list adding the next edge on the list to the current subgraph if such an inclusion does not create a cycle and simply skipping the edge otherwise.

KRUSKAL'S ALGORITHM

ALGORITHM *Kruskal*(G)

//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
Sort E in nondecreasing order of the edge weights $w(e_{i_1}) \le \ldots \le w(e_{i_{|E|}})$
$E_T \leftarrow \emptyset$; $ecounter \leftarrow 0$ //initialize the set of tree edges and its size
$k \leftarrow 0$ //initialize the number of processed edges
while $ecounter < |V| - 1$
    $k \leftarrow k + 1$
    if $E_T \cup \{e_{i_k}\}$ is acyclic
        $E_T \leftarrow E_T \cup \{e_{i_k}\}$; $ecounter \leftarrow ecounter + 1$
return E_T

Figure: Application of Kruskal's Algorithm. Selected edges are shown in bold.



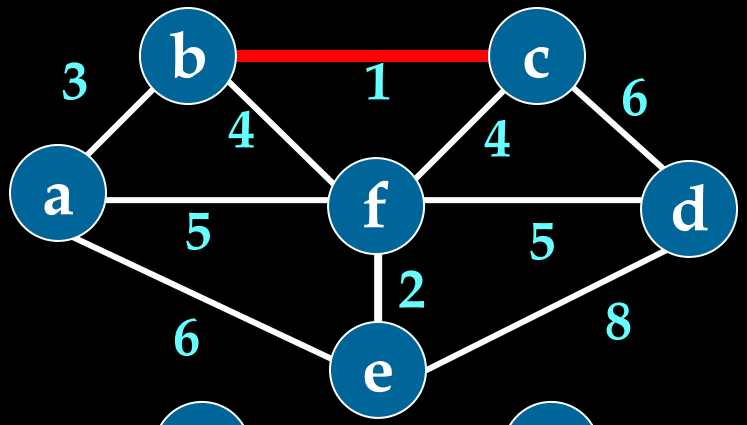
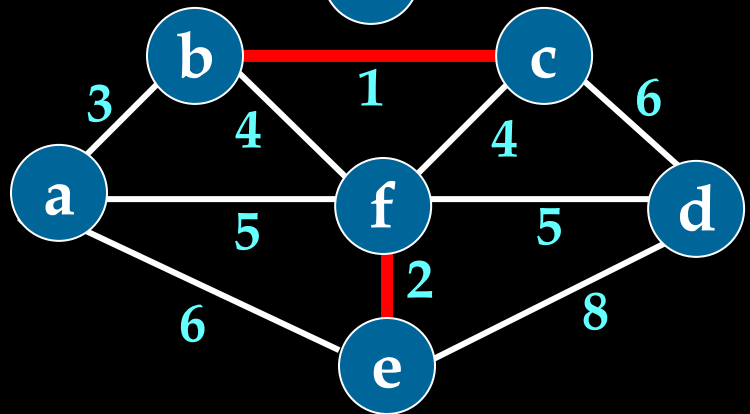
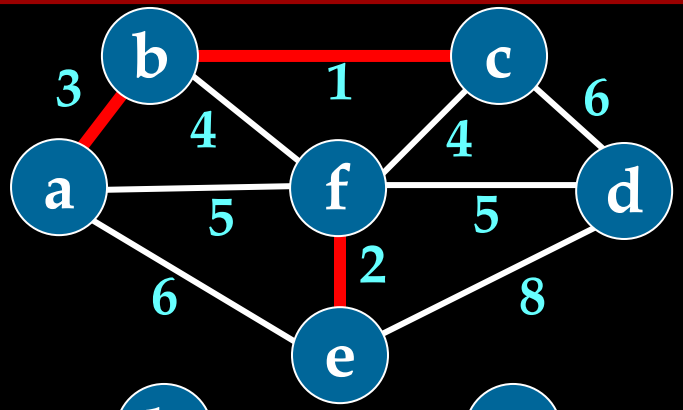
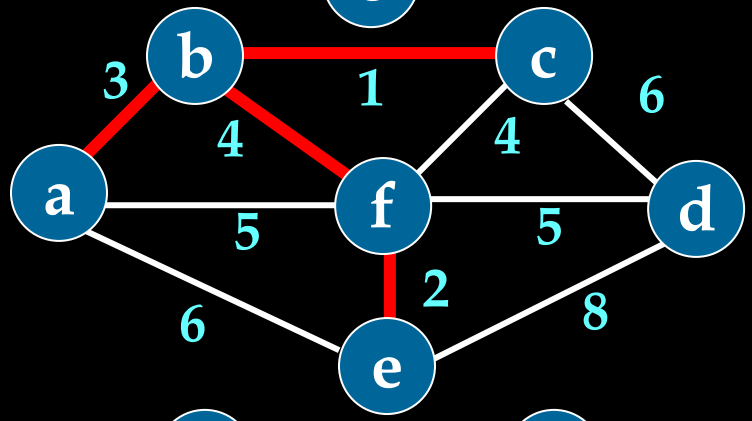
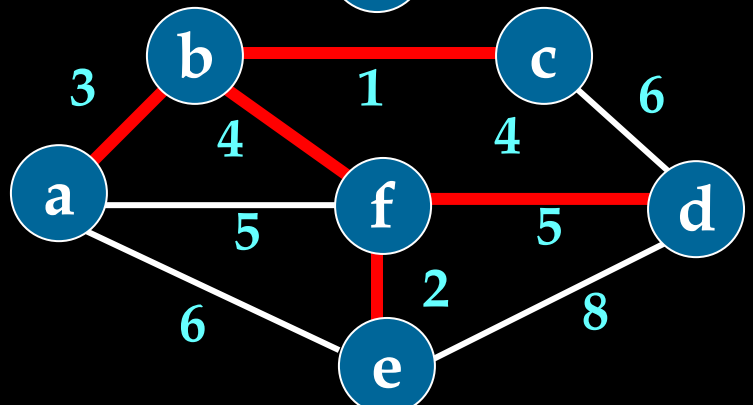
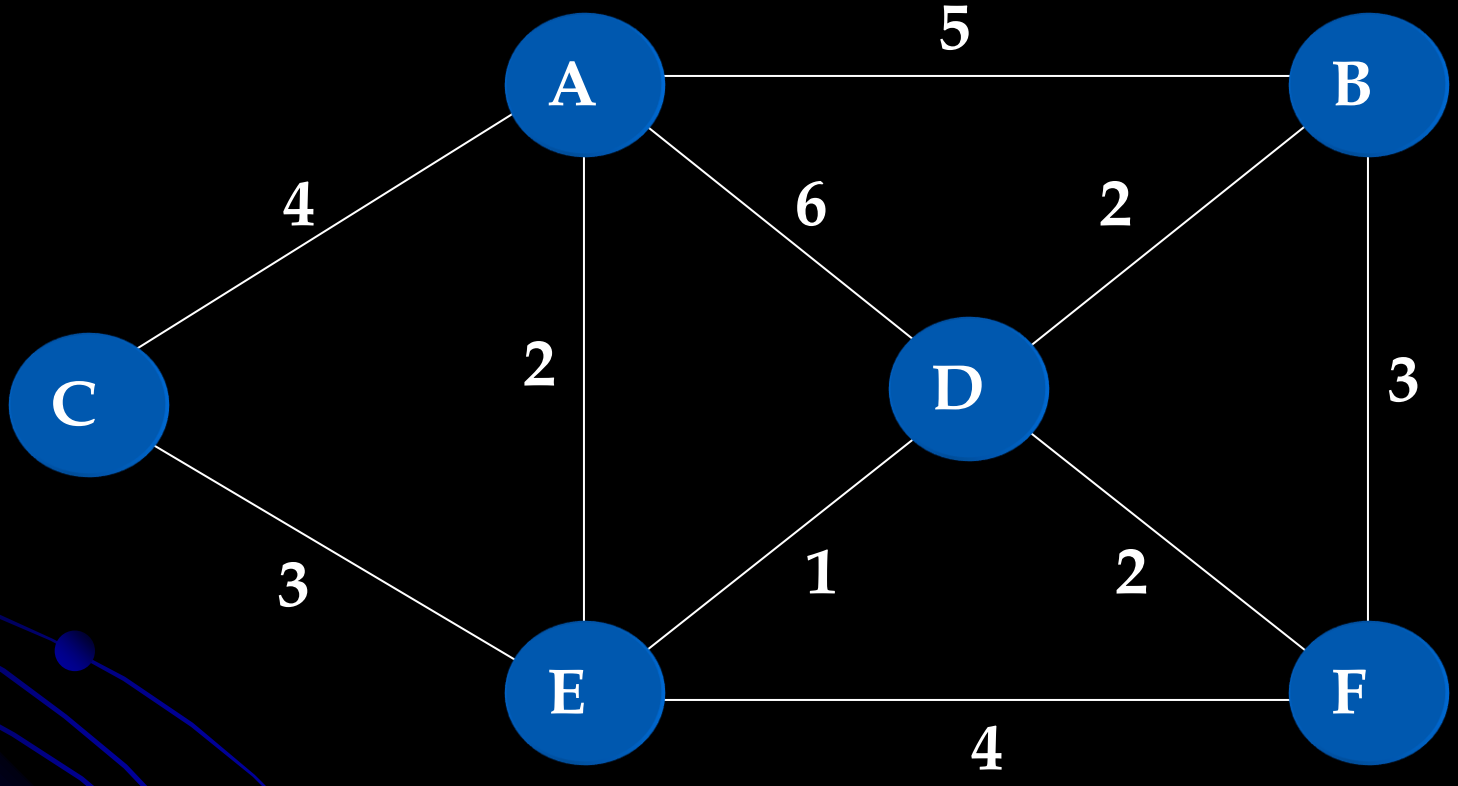
Tree vertices	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de **1** 2 3 4 4 5 5 6 6 8	
bc **1**	**bc** **ef** ab bf cf af df ae cd de **1** **2** 3 4 4 5 5 6 6 8	

Figure: Application of Kruskal's Algorithm. Selected edges are shown in bold.

Tree vertices	Sorted list of edges	Illustration
ef 2	**bc ef ab** bf cf af df ae cd de **1 2 3** 4 4 5 5 6 6 8	
ab 3	**bc ef ab bf** cf af df ae cd de **1 2 3 4** 4 5 5 6 6 8	
bf 4	**bc ef ab bf** cf af **df** ae cd de **1 2 3 4** 4 5 **5** 6 6 8	
df **5**		

KRUSKAL'S ALGORITHM



ANALYSIS

With an efficient sorting algorithm, efficiency of Kruskal's algorithm will be in $O(|E| \log |E|)$.

DIJKSTRA'S ALGORITHM

- ***single-source shortest-paths problem***: for a given vertex called the ***source*** in a weighted connected graph, find shortest paths to all its other vertices.
- The best-known algorithm for the single-source shortest-paths problem is ***Dijkstra's algorithm***.
- Edsger W. Dijkstra discovered this algorithm in mid-1950s.

DIJKSTRA'S ALGORITHM

- Assumptions:
  - the graph is connected
  - the edges are undirected (or directed)
  - the edge weights are nonnegative

DIJKSTRA'S ALGORITHM

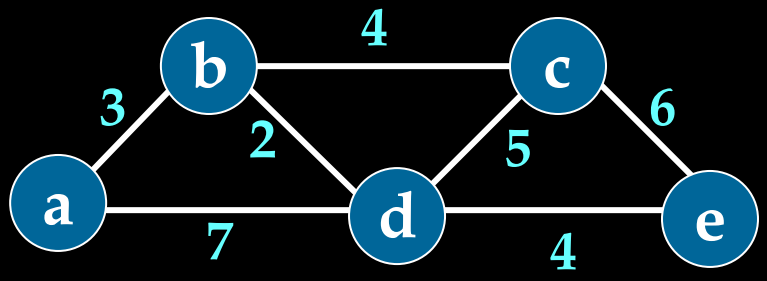
- First, it finds the shortest path from the source to a vertex nearest to it, then to a second nearest, and so on.
- Before its i^{th} iteration commences, the algorithm has already identified the shortest paths to i - 1 other vertices nearest to the source.
- These vertices, the source, and the edges of the shortest paths leading to them from the source form a subtree T_i of the given graph.
- The set of vertices adjacent to the vertices in T_i can be referred to as "fringe vertices"; they are the candidates from which Dijkstra's algorithm selects the next vertex nearest to the source.

DIJKSTRA'S ALGORITHM

- To identify the i^{th} nearest vertex, the algorithm computes, for every fringe vertex u, the sum of the distance to the nearest tree vertex v (given by the weight of the edge (v, u)) and the length d_v of the shortest path from the source to v.

- Then selects the vertex with the smallest such sum.

Figure: Application of Dijkstra's Algorithm. The next closest vertex is shown in bold.



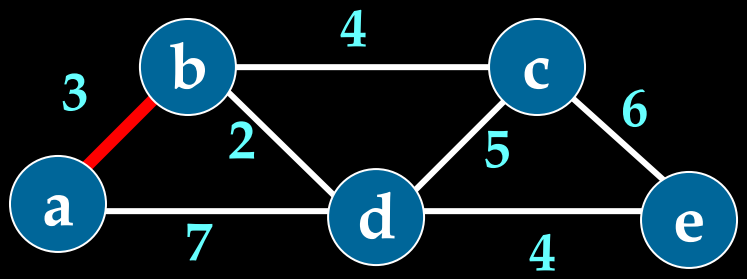
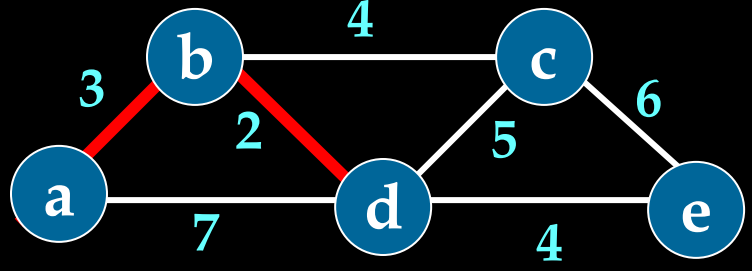
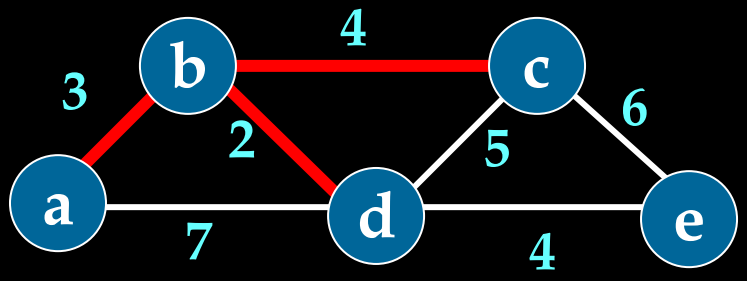
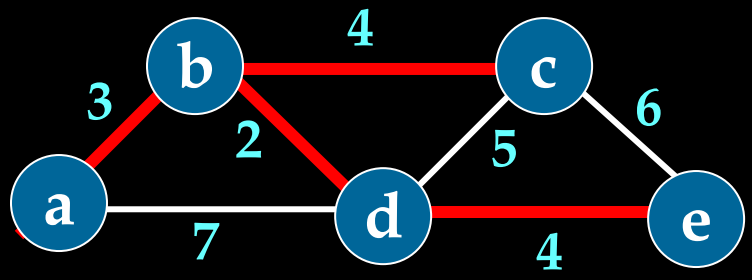
Tree vertices	Remaining vertices	Illustration
a(-, 0)	**b(a, 3)** c(-, ∞) d(a, 7) e(-, ∞)	
b(a, 3)	c(b, 3+4) **d(b, 3+2)** e(-, ∞)	

Figure: Application of Dijkstra's Algorithm. The next closest vertex is shown in bold.

Tree vertices	Remaining vertices	Illustration
d(b, 5)	**c(b, 7)** e(d, 5+4)	
c(b, 7)	**e(d, 9)**	
e(d, 9)		

The shortest paths and their lengths are:

from a to b: a - b **of length 3**

from a to d: a – b – d **of length 5**

from a to c: a – b – c **of length 7**

from a to e: a – b – d – e **of length 9**

DIJKSTRA'S ALGORITHM

ALGORITHM *Dijkstra*(G, s)

//Dijkstra's algorithm for single-source shortest paths
//Input: A weighted connected graph $G = \langle V, E \rangle$ and its vertex s
//Output: The length d_v of a shortest path from s to v
// and its penultimate vertex p_v for every vertex v in V
Initialize(Q) //initialize vertex priority queue to empty
for every vertex v in V **do**
    $d_v \leftarrow \infty$; $p_v \leftarrow$ **null**
    $Insert(Q, v, d_v)$ //initialize vertex priority in the priority queue
$d_s \leftarrow 0$; *Decrease*(Q, s, d_s) //update priority of s with d_s
$V_T \leftarrow \emptyset$
for $i \leftarrow 0$ **to** $|V| - 1$ **do**
    $u^* \leftarrow$ *DeleteMin*(Q) //delete the minimum priority element
    $V_T \leftarrow V_T \cup \{u^*\}$
    for every vertex u in $V - V_T$ that is adjacent to u^* **do**
        if $d_{u^*} + w(u^*, u) < d_u$
            $d_u \leftarrow d_{u^*} + w(u^*, u)$; $p_u \leftarrow u^*$
            Decrease(Q, u, d_u)

DIJKSTRA'S ALGORITHM ANALYSIS

- The time efficiency of Dijkstra's algorithm depends on the data structures used for representing an input graph.
- It is in $\theta(|V|^2)$ for graphs represented by their weight matrix.
- For graphs represented by their adjacency linked lists it is in $O(|E| \log |V|)$.

End of Chapter 9