UNIT 5

Greedy Technique

IDEA

- Computer scientists consider it a general design technique despite the fact that it is applicable to optimization problems only.

- 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

IDEA

- These requirements explain the technique's name: on each step, it suggests a "greedy" grab of the best alternative available in the hope that a sequence of locally optimal choices will yield a (globally) optimal solution to the entire problem.

SCENARIO 1

You have recently started playing a brand new computer game called "Mr. President". The game is about ruling a country, building infrastructures and developing it.

Your country consists of **N** cities and **M** bidirectional roads connecting them. Each road has assigned a cost of its maintenance. The greatest achievement in the game is called "Great Administrator" and it is given to a player who manages to have all cities in the country connected by roads in such a way that it is possible to travel between any two cities and that the sum of maintenance costs of these roads is the least.

How will you solve this problem?

SCENARIO 2

You are the owner of a company with branch offices in several cities; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost.

How will you solve this problem?

PROBLEM

- Given *n* points, connect them in the cheapest possible way so that there will be a path between every pair of points.

- We can represent the points by vertices of a graph, possible connections by the graph's edges, and the connection costs by the edge weights.

- Then the question can be posed as the minimum spanning tree problem, defined formally as follows:
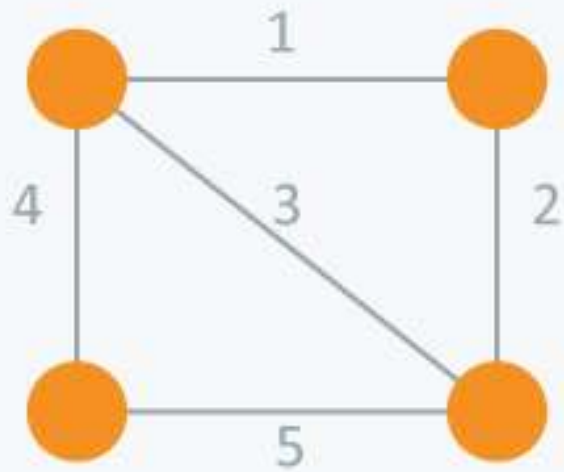
MINIMUM SPANNING TREE

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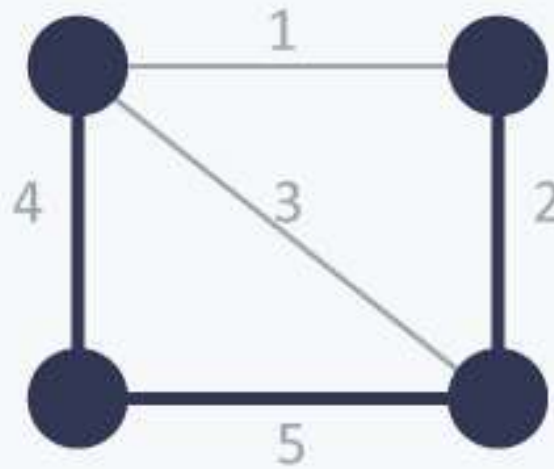
A *minimum spanning tree* of a weighted connected graph is its spanning tree of the smallest weight, where the *weight* of a tree is defined as the sum of the weights on all its edges.

The *minimum spanning tree problem* is the problem of finding a minimum spanning tree for a given weighted connected graph.
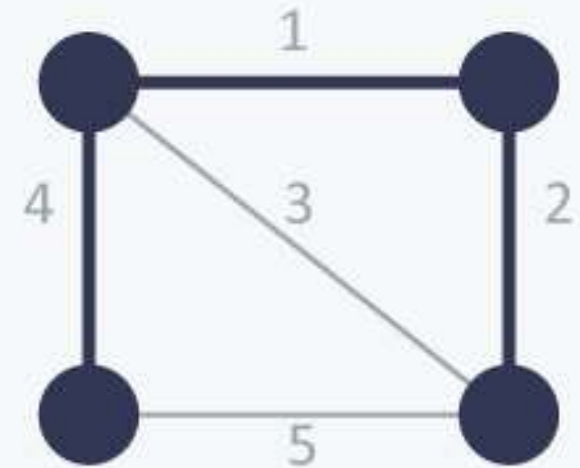
MINIMUM SPANNING TREE

Undirected Graph

Spanning Tree
Cost = 11(=4+5+2)

Minimum Spanning Tree
Cost = 7(=4+1+2)

PRIM'S ALGORITHM - IDEA

- 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.

- (By the nearest vertex, we mean a vertex not in the tree connected to a vertex in the tree by an edge of the smallest weight. Ties can be broken arbitrarily.)

PRIM'S ALGORITHM - IDEA

- 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 - IDEA

- 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 length (the weight) of the corresponding edge.

- Vertices that are not adjacent to any of the tree vertices can be given the label ∞ indicating their "infinite" distance to the tree vertices and a null label for the name of the nearest tree vertex.
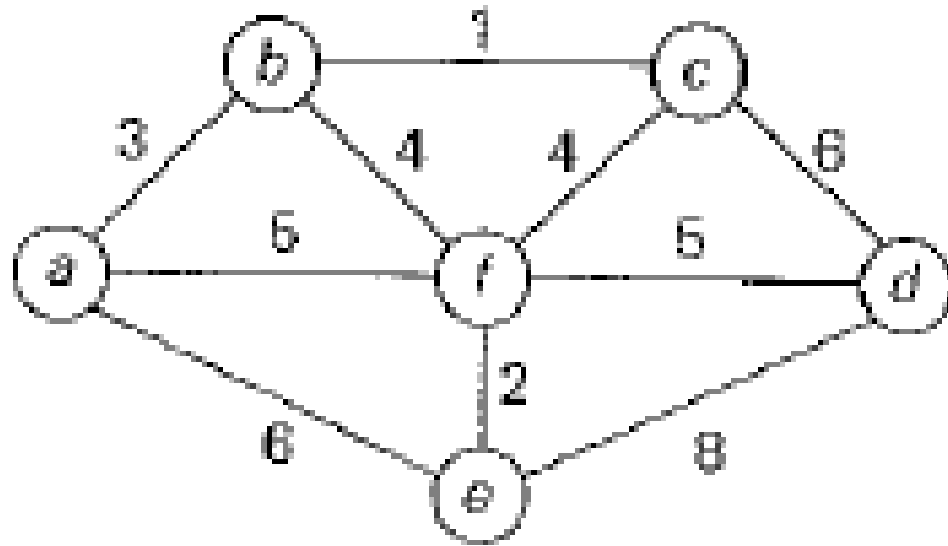
PRIM'S ALGORITHM - IDEA

- Alternatively, 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 the other vertices of the graph, called "unseen" because they are yet to be affected by the algorithm.)

- With such labels, finding the next vertex to be added to the current tree $T = (V_T, E_T)$ becomes a simple task of finding a vertex with the smallest distance label in the set $V- V_T$.
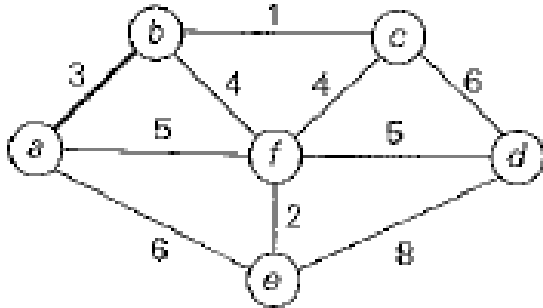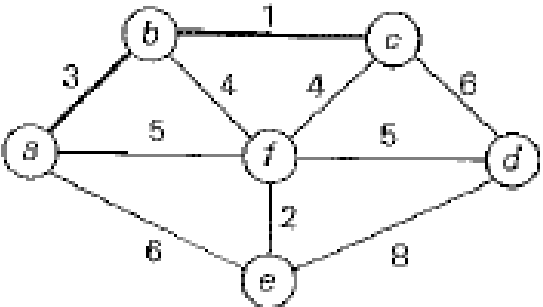
- Ties can be broken arbitrarily.

PRIM'S ALGORITHM - IDEA

- After we have identified a vertex $u*$ to be added to the tree, we need to perform **two operations:**
  - Move $u*$ from the set $V - V_T$ to the set of tree vertices V_T.
  - For each remaining vertex u in $V - V_T$ that is connected to $u*$ by a shorter edge than the u's current distance label, update its labels by $u*$ and the weight of the edge between $u*$ and u, respectively.
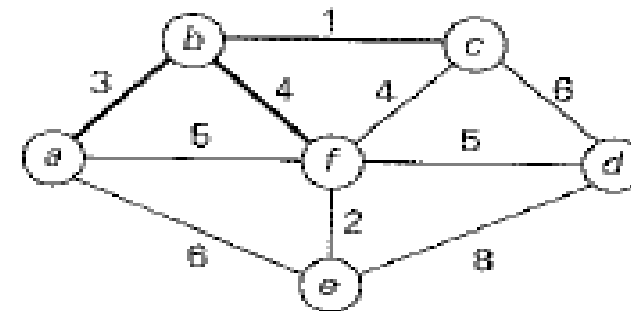
PRIM'S ALGORITHM - EXAMPLE

PRIM'S ALGORITHM - EXAMPLE

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)	

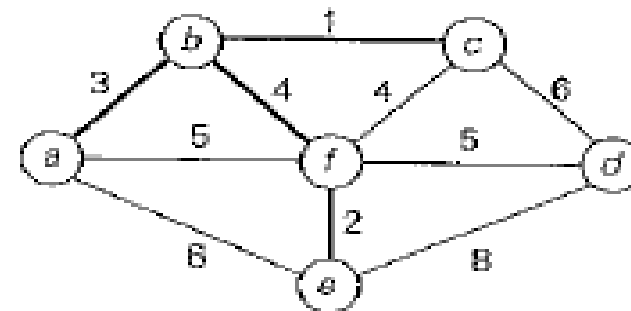# PRIM'S ALGORITHM - EXAMPLE

c(b, 1)   d(c, 6) e(a, 6) **f(b, 4)**

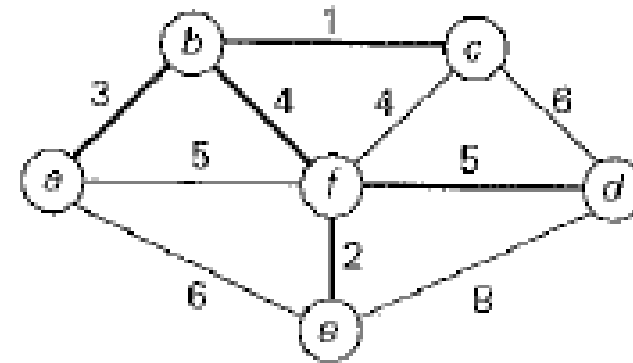
f(b, 4)   d(f, 5) **e(f, 2)**
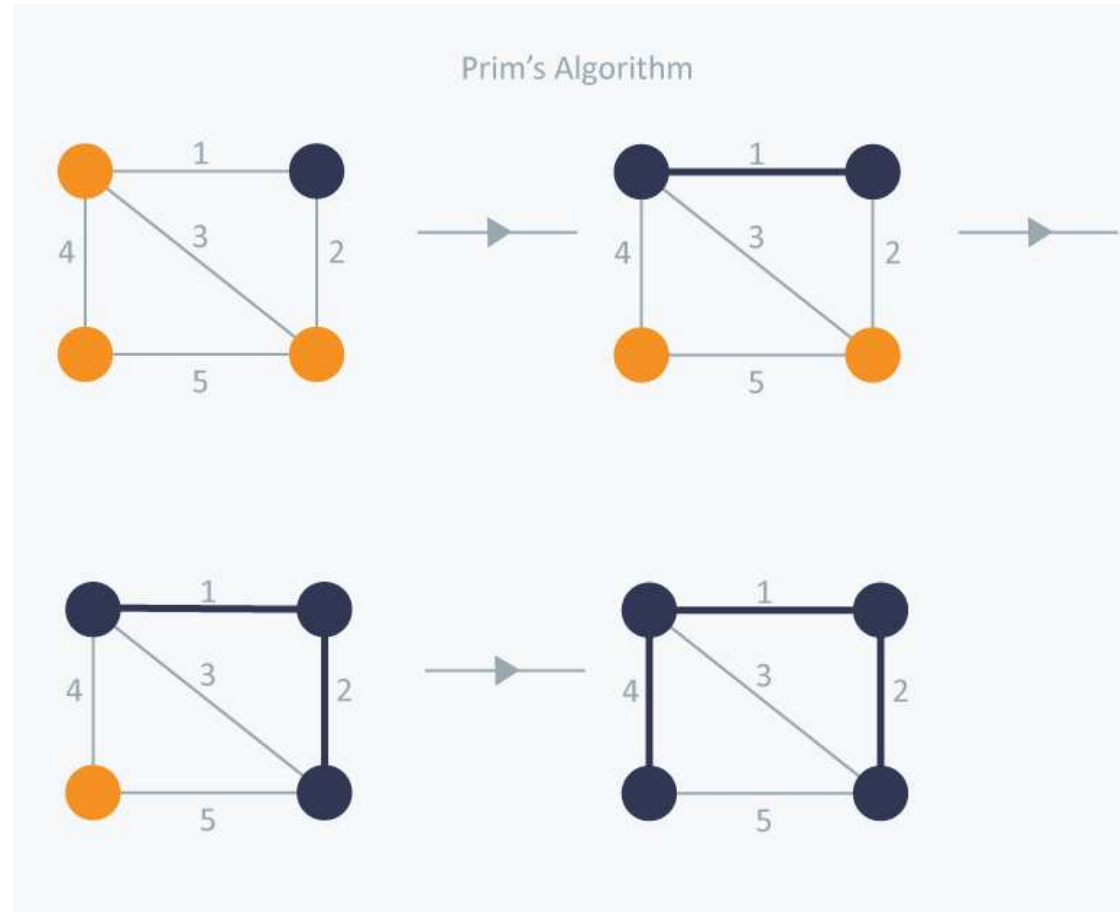
PRIM'S ALGORITHM - EXAMPLE

e(f, 2)   d(f, 5)

d(f, 5)

PRIM'S ALGORITHM - EXAMPLE

PRIM'S ALGORITHM – PROOF OF CORRECTNESS

- Let us prove by induction that each of the subtrees T_i, $i = 0, \ldots, n-1$, generated by Prim's algorithm is a part (i.e., a subgraph) of some minimum spanning tree.

- This immediately implies, of course, that the last tree in **the sequence, T_{n-1}, is a minimum spanning tree itself because it contains all n** vertices of the graph.

- The basis of the induction is trivial, since T_0 consists of a single vertex and hence must be a part of any minimum spanning tree.

- For the **inductive step, let us assume that** T_{i-1} **is part of some minimum spanning tree T.**

- We need to prove that T_i, generated from T_{i-1} by Prim's algorithm, is also a part of a minimum spanning tree.

PRIM'S ALGORITHM – PROOF OF CORRECTNESS

- We prove this by contradiction by assuming that no minimum spanning tree of the graph can contain T_i

- Let $e_1 = (v, u)$ be the minimum weight edge from a vertex in T_{i-1} to a vertex not in T_{i-1} used by Prim's algorithm to expand T_{i-1} to T_i.

- By our assumption, e_1 cannot belong to the minimum spanning tree T. Therefore, if we add e_1 to T, a cycle must be formed.

- In addition to edge $e_1 = (v, u)$, this cycle must contain another edge (v', u') connecting a vertex v' belonging to T_{i-1} to a vertex u' that is not in T_{i-1}.

- It is possible that v' coincides with v or u' coincides with u but not both.

PRIM'S ALGORITHM – PROOF OF CORRECTNESS

- If we now delete the edge *(v', u')* from this cycle, we obtain another spanning tree of the entire graph whose weight is less than or equal to the weight *of T* since the weight of *e1* is less than or equal to the weight of (*v', u')*.

- Hence, this spanning tree is a minimum spanning **tree, which contradicts the assumption that no minimum spanning tree contains** T_i.

PRIM'S ALGORITHM – EFFICIENCY

- If a graph is represented by its weight matrix and the priority queue is implemented as an unordered array, the algorithm's running time will be in $\Theta(|V|^2)$.

- If a graph is represented by its adjacency lists and the priority queue is implemented as a min-heap, the running time of the algorithm is in $\Theta(|E| \log |V|)$.

KRUSKAL'S ALGORITHM - IDEA

- The algorithm begins by sorting the graph's edges in non decreasing 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 sub graph if such an inclusion does not create a cycle and simply skipping the edge otherwise.

KRUSKAL'S ALGORITHM - IDEA

- The algorithm begins by sorting the graph's edges in non decreasing 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 sub graph if such an inclusion does not create a cycle and simply skipping the edge otherwise.
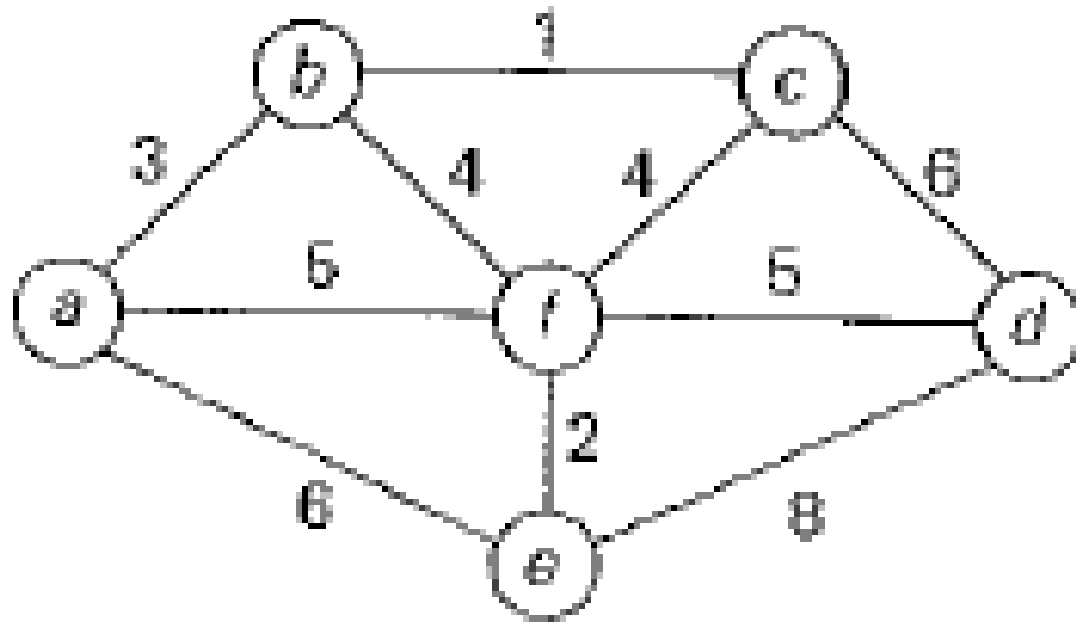
KRUSKAL'S ALGORITHM - IDEA

ALGORITHM $Kruskal(G)$

//Kruskal's algorithm for constructing a minimum spanning tree

//Input: A weighted connected graph $G = (V, E)$

//Output: E_T, the set of edges composing a minimum spanning tree of G

sort E in nondecreasing order of the edge weights $w(e_{i_1}) \leq \ldots \leq w(e_{i_{|E|}})$

$E_T \leftarrow \emptyset$; $ecounter \leftarrow 0$ //initialize the set of tree edges and its size

$k \leftarrow 0$ //initialize the number of processed edges

while $ecounter < |V| - 1$ **do**

    $k \leftarrow k + 1$

    if $E_T \cup \{e_{i_k}\}$ is acyclic

        $E_T \leftarrow E_T \cup \{e_{i_k}\}$; $ecounter \leftarrow ecounter + 1$
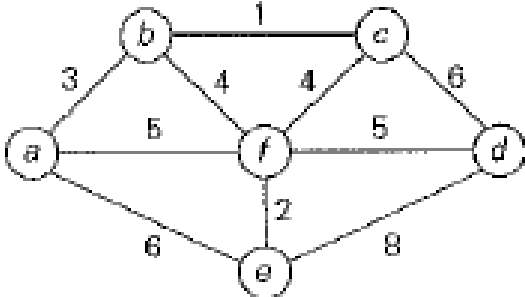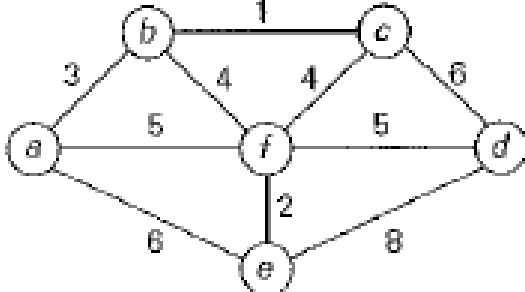
return E_T

KRUSKAL'S ALGORITHM – PROOF OF CORRECTNESS

- The correctness of Kruskal's algorithm can be proved by repeating the essential steps of the proof of Prim's algorithm given in the previous section.

KRUSKAL'S ALGORITHM – EXAMPLE
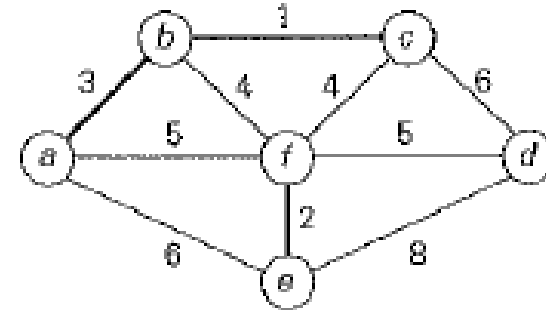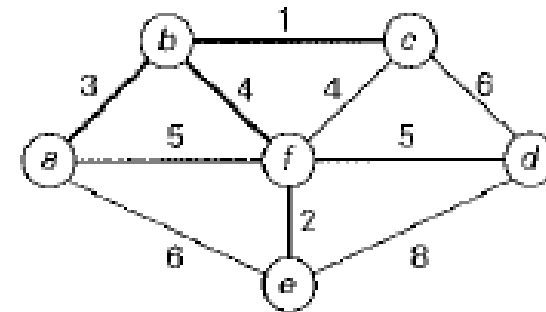
KRUSKAL'S ALGORITHM – EXAMPLE

Tree edges	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

KRUSKAL'S ALGORITHM – EXAMPLE

ef		bc	ef	**ab**	bf	cf	af	df	ae	cd	de
2		1	2	3	4	4	5	5	6	6	8

ab		bc	ef	ab	**bf**	cf	af	df	ae	cd	de
3		1	2	3	4	4	5	5	6	6	8

KRUSKAL'S ALGORITHM – EXAMPLE

bf
4

bc	ef	ab	bf	cf	af	**df**	ae	cd	de
1	2	3	4	4	5	5	6	6	8

df
5
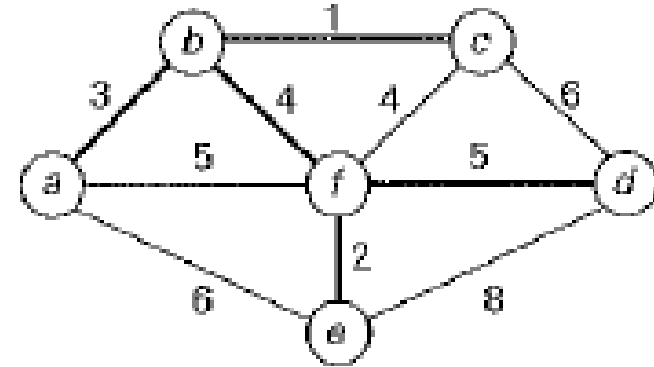
KRUSKAL'S ALGORITHM

- We can consider the algorithm's operations as a progression through a series of forests containing *all* the vertices of a given graph and *some* of its edges.

- The initial forest consists of |V| trivial trees, each comprising a single vertex of the graph.

- The final forest consists of a single tree, which is a minimum spanning tree of the graph.

- On each iteration, the algorithm takes the next edge *(u, v)* from the sorted list of the graph's edges, finds the trees **containing the vertices *u* and *v*, and, if these trees are not the same, unites them** in a larger tree by adding the edge *(u, v)*.

- Fortunately, there are efficient algorithms for doing so, including the crucial check whether two vertices belong to the same tree. They are called *union-find* algorithms.

KRUSKAL'S ALGORITHM – EXAMPLE
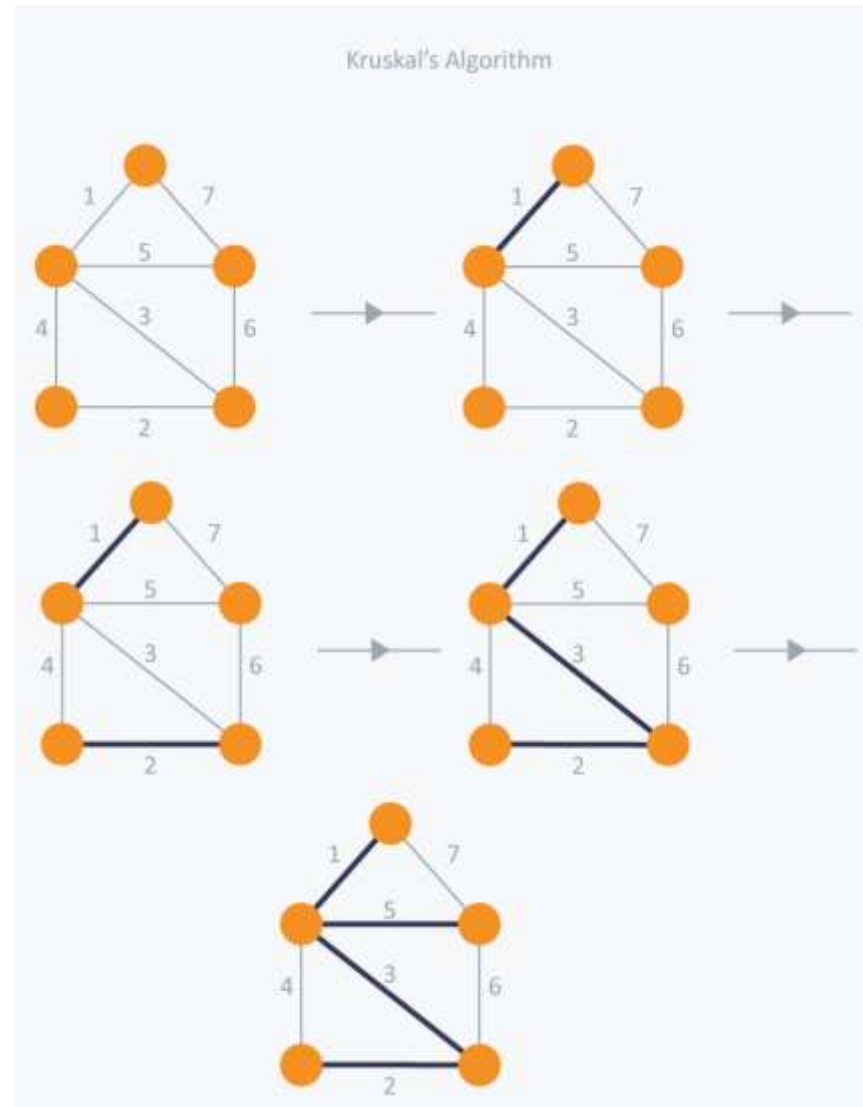
DIJKSTRA'S ALGORITHM

- *Solves the 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.

- This algorithm is applicable to graphs with non negative weights only.

DIJKSTRA'S ALGORITHM – IDEA

- Dijkstra's algorithm finds the shortest paths to a graph's vertices in order of their distance from a given source.

- First, it finds the shortest path from the source to a vertex nearest to it, then to a second nearest, and so on.

- In general, before its ith 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.

DIJKSTRA'S ALGORITHM – IDEA

- The next vertex nearest to the source can be found among the vertices adjacent to the vertices of T_i.

- 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.

- To identify the ith 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 (previously determined by the algorithm) and then selects the vertex with the smallest such sum.

- The fact that it suffices to compare the lengths of such special paths is the central insight of Dijkstra's algorithm.

DIJKSTRA'S ALGORITHM – IDEA

- To facilitate the algorithm's operations, we label each vertex with two labels.

- The numeric label d indicates the length of the shortest path from the source to this vertex found by the algorithm so far; when a vertex is added to the tree, d indicates the length of the shortest path from the source to that vertex.

- The other label indicates the name of the next-to-last vertex on such a path, i.e., the parent of the vertex in the tree being constructed.

DIJKSTRA'S ALGORITHM – IDEA

- With such labeling, finding the next nearest vertex $u*$ becomes a simple task of finding a fringe vertex with the smallest d value. Ties can be broken arbitrarily.

- After we have identified a vertex $u*$ to be added to the tree, we need to perform two operations:

  - Move $u*$ from the fringe to the set of tree vertices.
  - For each remaining fringe vertex u that is connected to $u*$ by an edge of weight $w(u*, u)$ such that $d_{.,.} + w(u*, u) < d_{.,,}$ update the labels of u by $u*$ and $du* + w(u*, u)$, respectively.

DIJKSTRA'S ALGORITHM

ALGORITHM   *Dijkstra(G, s)*

//Dijkstra's algorithm for single-source shortest paths

//Input: A weighted connected graph $G = (V, E)$ with nonnegative weights

//   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
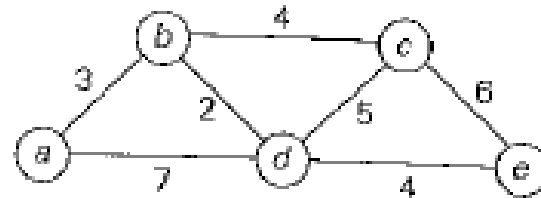
   $V_T \leftarrow V_T \cup \{u^*\}$

   for every vertex u in $V - V_T$ that is adjacent to u^* **do**

      if $d_{u^*} + w(u^*, u) < d_u$

         $d_u \leftarrow d_{u^*} + w(u^*, u)$; $p_u \leftarrow u^*$

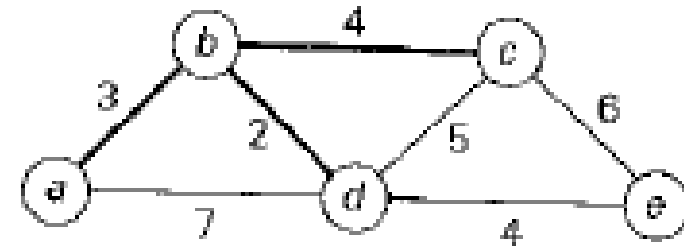   *Decrease*(Q, u, d_u)

DIJKSTRA'S ALGORITHM - EXAMPLE

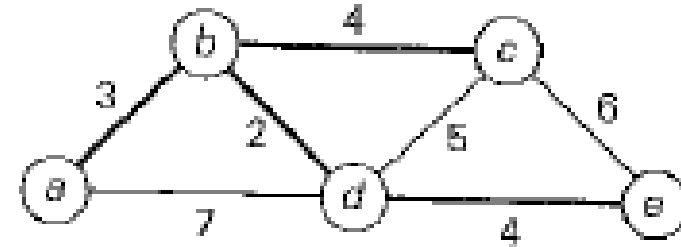| Tree vertices	Remaining vertices	Illustration
a(−, 0)	**b(a, 3)** c(−, ∞) d(a, 7) e(−, ∞)	
b(a, 3)	c(b, 3 + 4) **d(b, 3 + 2)** e(−, ∞)	

DIJKSTRA'S ALGORITHM - EXAMPLE

d(b, 5)   **c(b, 7)** e(d, 5 + 4)

c(b, 7)   **e(d, 9)**

e(d, 9)

DIJKSTRA'S ALGORITHM – EFFICIENCY

- If a graph is represented by its weight matrix and the priority queue is implemented as an unordered array, the algorithm's running time will be in $\Theta(|V|^2)$.

- If a graph is represented by its adjacency lists and the priority queue is implemented as a min-heap, the running time of the algorithm is in $\Theta(|E| \log |V|)$.

PROBLEM

Encode a text that comprises characters from some n-character alphabet by assigning to each of the text's characters some sequence of bits called the *codeword*.

SOLUTION

Fixed-length Encoding

- Assigns to each character a bit string of the same length m (m <= log2 n). This is exactly what the standard ASCII code does.
- Problem: Consumes more space

Variable-length Encoding

- Assigns codewords of different lengths to different characters.

PROBLEM WITH VARIABLE LENGTH ENCODING

Suppose Variable – Length Encoding assigns the following codes:

A – 0, B – 1, C – 01, D – 00, E – 001

So if the encoded string is 00101, is the actual string EC or AABC or DBC or … ?

PROBLEM WITH VARIABLE LENGTH ENCODING

- To avoid this complication, *prefix-free* (or simply *prefix) codes are used*.

- In a prefix code, no codeword is a prefix of a codeword of another character.

- Hence, with such an encoding, we can simply scan a bit string until we get the first group of bits that is a codeword for some character, replace these bits by this character, and repeat this operation until the bit string's end is reached.

HUFFMAN'S ALGORITHM

Huffman's Algorithm

Step 1 Initialize n one-node trees and label them with the characters of the alphabet. Record the frequency of each character in its tree's root to indicate the tree's *weight*. (More generally, the weight of a tree will be equal to the sum of the frequencies in the tree's leaves.)

Step 2 Repeat the following operation until a single tree is obtained. Find two trees with the smallest weight (ties can be broken arbitrarily, but see Problem 2 in the exercises). Make them the left and right subtree of a new tree and record the sum of their weights in the root of the new tree as its weight.
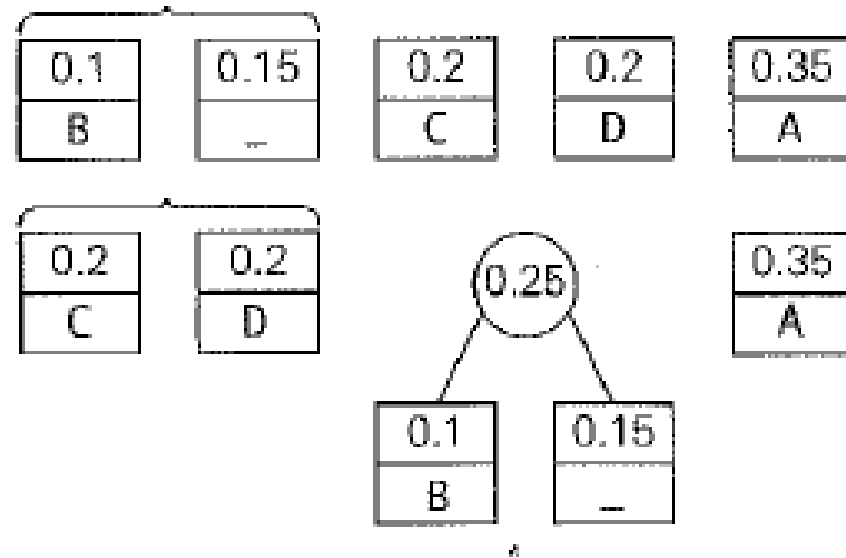
A tree constructed by the above algorithm is called a *Huffman tree*. It defines—in the manner described—a *Huffman code*.
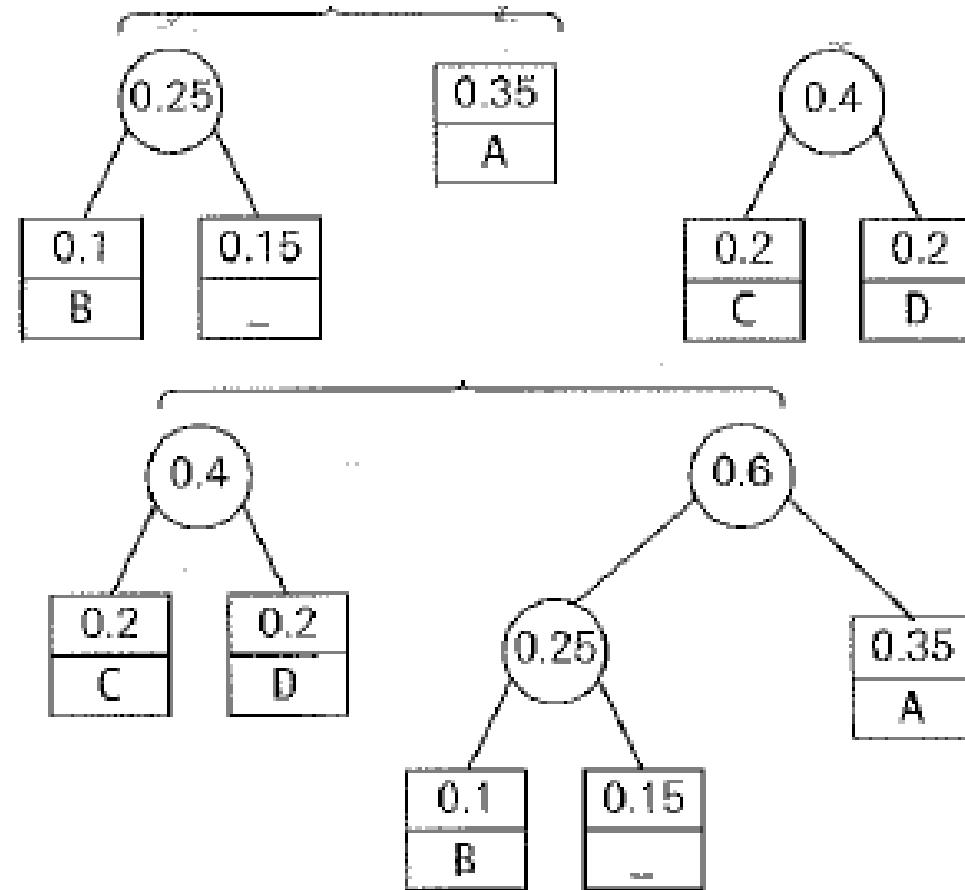
EXAMPLE

- Consider the five-character alphabet {A, B, C, D, _} with the following occurrence probabilities:

character	A	B	C	D	_
probability	0.35	0.1	0.2	0.2	0.15

EXAMPLE

EXAMPLE

EXAMPLE
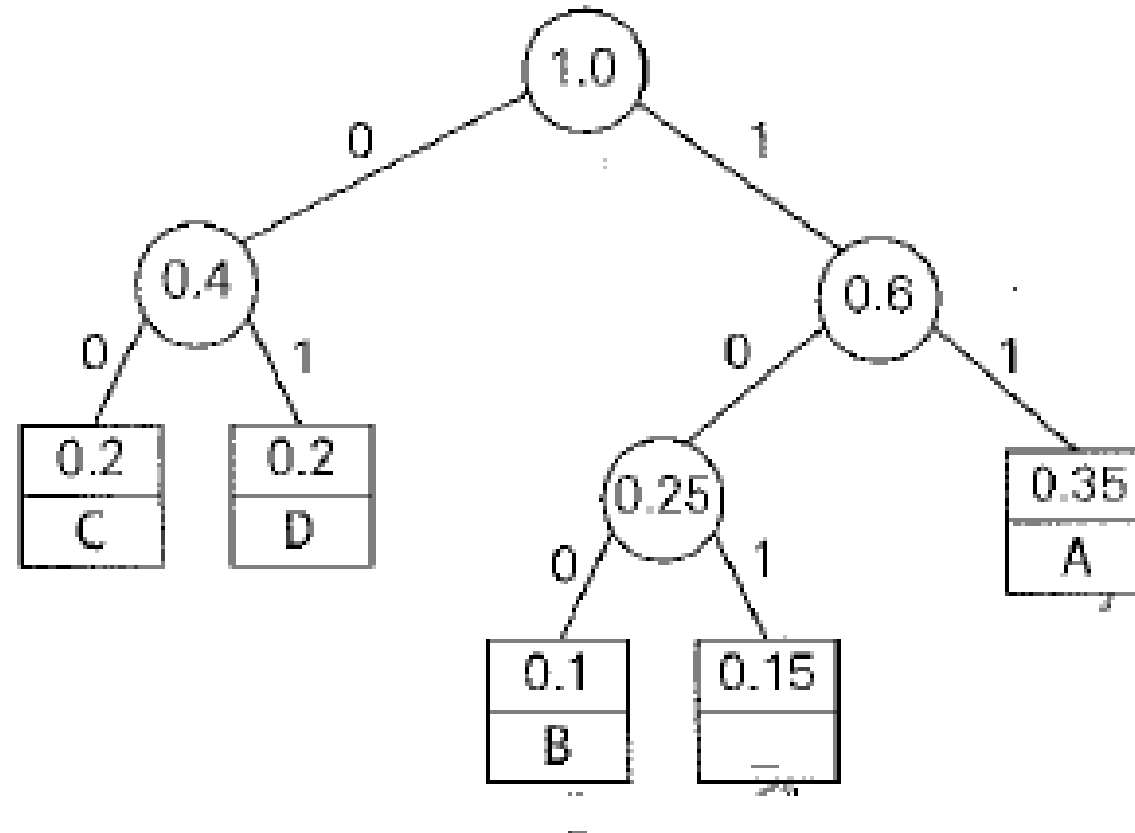
EXAMPLE

character	A	B	C	D	_
probability	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

EXAMPLE

- With the occurrence probabilities given and the codeword lengths obtained, the expected number of bits per character in this code is

  2. 0.35 + 3. 0.1 + 2. 0.2 + 2. 0.2 + 3. 0.15 = 2.25.

- Had we used a fixed-length encoding for the same alphabet, we would have to use at least three bits per each character.

- Thus, for this toy example, Huffman's code achieves the *compression ratio* – a standard measure of a compression algorithm's effectiveness-of (3- 2.25)/3 · 100% = 25%.

APPLICATIONS

- File Compression (ZIP, WINZIP, JPEG, PNG, etc.)
- Construction of a binary tree with minimum weighted path length