Algorithm Analysis

Chapter 9.1, 9.2, 9.3, 9.4

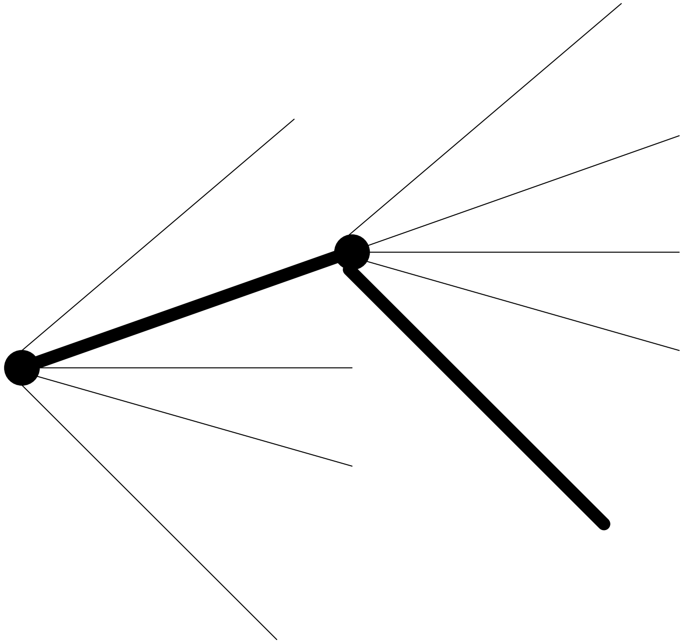
ROAD MAP

- **Greedy Technique**
  - **Knapsack Problem**
  - Minimum Spanning Tree Problem
    - Prim’s Algorithm
    - Kruskal’s Algorithm
  - Single Source Shortest Paths
    - Dijkstra’s Algorithm
  - Huffman Trees

Greedy Technique

- Used for solving *optimization problems*
  - such as engineering problems
- Construct a solution through a sequence of *decision steps*
  - Each expanding a partially constructed solution
  - Until a complete solution is reached
- Similar to dynamic programming
  - but, not all possible solutions are explored

Greedy Technique



On each decision step the choice should be

- **Feasible**
  - has to satisfy the problem’s constraints
- **Locally optimal**
  - has to be the best local choice
- **Irrevocable**
  - once made, it can not be changed

Greedy Technique

```
Greedy Algorithm ( a [ 1 .. N ] )
{
  solution = Ø
  for i = 1 to n
     x = select (a)
     if feasible ( solution, x )
          solution = solution U {x}
  return solution
}
```

Greedy Technique

- In each step, greedy technique suggests a *greedy* selection of the best alternative avaliable
  - Feasible decision
  - Locally optimal decision
  - Hope to yield a globally optimal solution
- Greedy technique *does not* give the optimal solution for all problems

Applications of the Greedy Strategy

- Optimal solutions:
  - change making for “normal” coin denominations
  - minimum spanning tree (MST)
  - single-source shortest paths
  - simple scheduling problems
  - Huffman codes

- Approximations:
  - traveling salesman problem (TSP)
  - knapsack problem
  - other combinatorial optimization problems

Change-Making Problem

Given unlimited amounts of coins of denominations $d_1 > \ldots > d_m$, give change for amount n with the least number of coins

Example: $d_1 = 25c$, $d_2 =10c$, $d_3 = 5c$, $d_4 = 1c$ and $n = 48c$

Greedy solution:

Greedy solution is

- optimal for any amount and “normal’’ set of denominations
- may not be optimal for arbitrary coin denominations

Fractional Knapsack Problem

- *Given :*

  w_i : weight of object *i*

  m : capacity of knapsack

  p_i : profit of all of *i* is taken

- *Find:*

  x_i : fraction of *i* taken

- *Feasibility:*

$$\sum_{i=1}^{n} x_i w_i \leq m$$

- *Optimality:*

$$\text{maximize} \quad \sum_{i=1}^{n} x_i p_i$$

Greedy Technique

```
Greedy Algorithm ( a [ 1 .. N ] )
{
  solution = Ø
  for i = 1 to n
     x = select (a)
     if feasible ( solution, x )
          solution = solution U {x}
  return solution
}
```

Knapsack Problem

```
Algorithm Knapsack (m,n)
   for i = 1 to n
        x(i) = 0
   for i = 1 to n
        select the object (j) with largest unit value
        if (w[j] < m)
              x[j] = 1.0
              m = m - w[j]
        else
              x[j] = m/w[j]
              break
```

- <u>Example :</u>
  M = 20 n = 3
  p = (25, 24, 15) w = (18, 15, 10)

ROAD MAP

- **Greedy Technique**
  - Knapsack Problem
  - **Minimum Spanning Tree Problem**
    - Prim’s Algorithm
    - Kruskal’s Algorithm
  - Single Source Shortest Paths
    - Dijkstra’s Algorithm
  - Huffman Trees

Minimum Spanning Tree (MST)

- Problem Instance:
  - *A weighted, connected, undirected graph G (V, E)*

- Definition:
  - *A spanning tree* of a connected graph is its connected acyclic subgraph
  - A *minimum spanning tree* of a weighted connected graph is its spanning tree of the smallest weight
    - *weight* of a tree is defined as the sum of the weights on all its edges

- Feasible Solution:
  - *A spanning tree G' of G*

$$G' = (V, E') \qquad E' \subseteq E$$

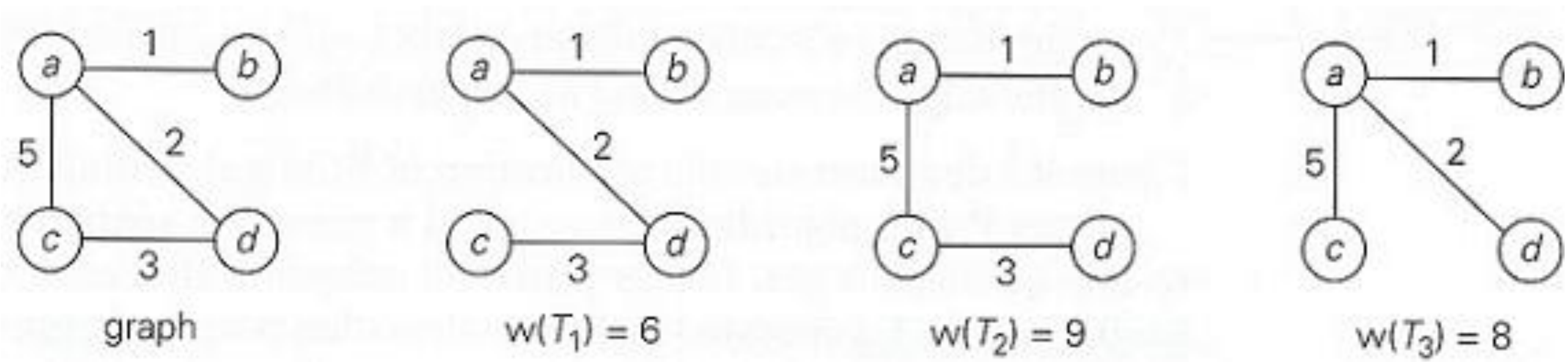
Minimum Spanning Tree

- Objective function :
  - Sum of all edge costs in G'

$$C(G') = \sum_{e \in G'} C(e)$$

- Optimum Solution :
  - Minimum cost spanning tree

Minimum Spanning Tree



T_1 is the minimum spanning tree

Greedy Technique

```
Greedy Algorithm ( a [ 1 .. N ] )
{
  solution = Ø
  for i = 1 to n
     x = select (a)
     if feasible ( solution, x )
          solution = solution U {x}
  return solution
}
```

Prim’s Algorithm

- Prim’s algorithm constructs a MST through a sequence of expanding subtrees

- Greedy choice :
  - Choose minimum cost edge add it to the subgraph

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

Approach :

1. Each vertex *j* keeps near[j] Є T (current tree)

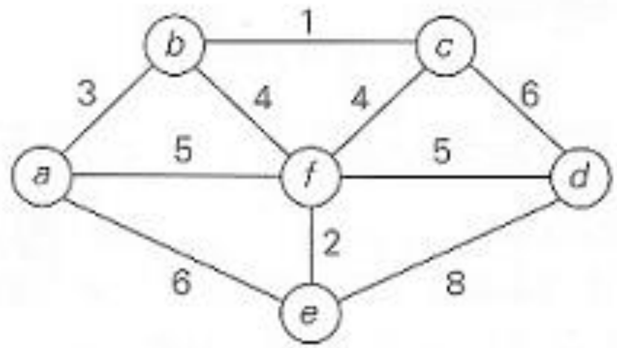
   where **`cost(j,near[j])`** is minimum

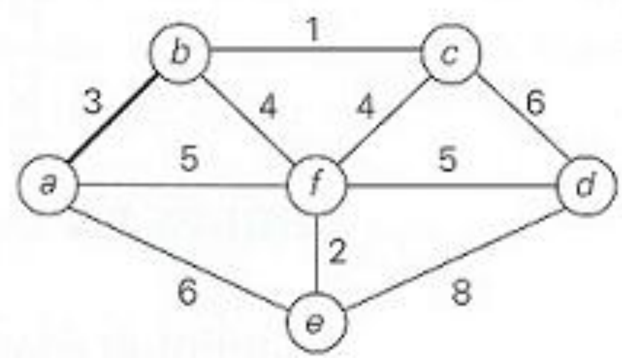
2. **`near[j] = 0 if j Є T`**

   = ∞ if there is no egde between *j* and *T*

3. Use a heap to select minimum of all edges

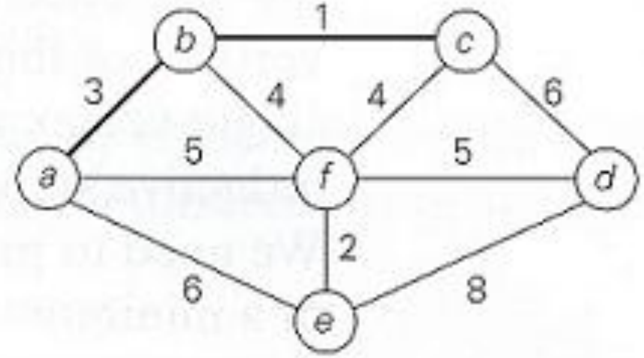
Prim's Algorithm Example



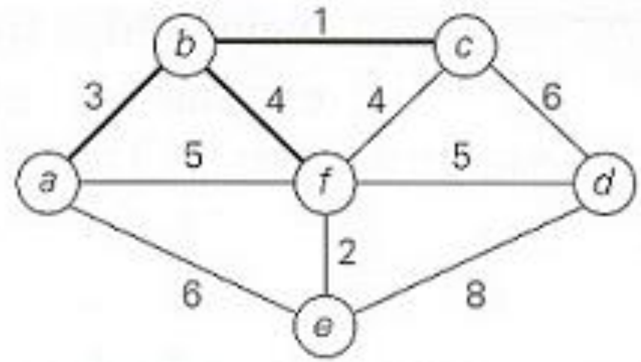
Tree vertices	Remaining vertices	Illustration
a(−, −)	**b(a, 3)** c(−, ∞) d(−, ∞) e(a, 6) f(a, 5)	

Prim's Algorithm Example

b(a, 3) **c(b, 1)** d(−, ∞) e(a, 6) f(b, 4)

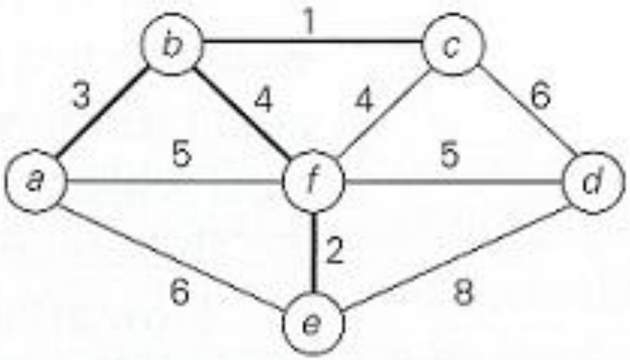


c(b, 1) d(c, 6) e(a, 6) **f(b, 4)**

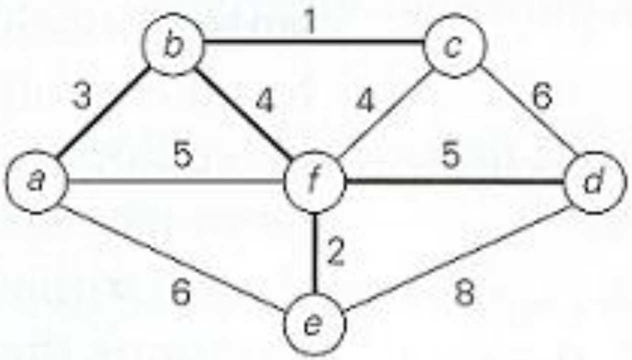


Prim's Algorithm Example

f(b, 4)   d(f, 5) **e(f, 2)**



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



d(f, 5)

Prim's Algorithm

```
1.  Initialize S with the start vertex, s, and V–S with the remaining vertices
2.      for all v in V – S
3.              if there is an edge (s, v)
4.                      Set cost[v] to w(s, v)
5.                      Set next[v] to s
                else
6.                      Set cost[v] to ∞
7.                      Set next[v] to NULL
8.      while V – S is not empty
9.              for all u in V – S, find the smallest cost[u]
10.             Remove u from V – S and add it to S
11.             Insert the edge (u, next[u]) into the spanning tree.
12.             for all v adjacent to u in V – S
13.                     if w(u, v) < cost[v]
14.                             Set cost[v] to w(u, v)
15.                             Set next[v] to u.
```

Prim's Algorithm

Analysis :

- How efficient is Prim's algorithm ?
  - It depends on the data structure chosen
  - running time is $\Theta(|V|^2)$ If
    - graph is represented by its weight matrix
    - unordered array is used
  - running time of is $O(|E| \log |V|)$ If
    - graph is represented by adjacency list
    - priority queue such as a min-heap is used

Kruskal’s Algorithm

- Another algorithm to construct MST
- Expands a subgraph
  - initially contains all the vertices but no edges
- Generates a sequence of subgraphs
  - always acyclic
  - not necessarily connected
- Resulting graph is connected and acyclic (i.e., tree)

Greedy choice :

- Choose minimum cost edge
  - Connecting two disconnected subgraphs
- It always yields an optimal solution

Greedy Technique

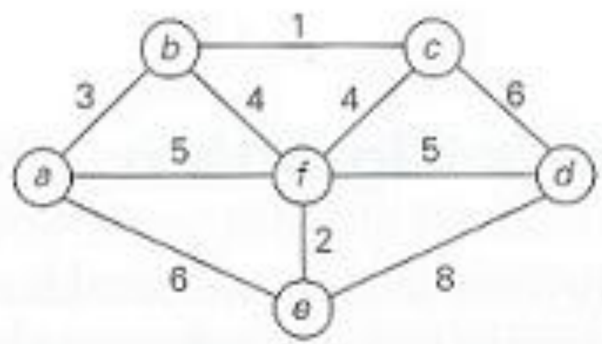
```
Greedy Algorithm ( a [ 1 .. N ] )
{
  solution = Ø
  for i = 1 to n
     x = select (a)
     if feasible ( solution, x )
          solution = solution U {x}
  return solution
}
```

Kruskal's Algorithm

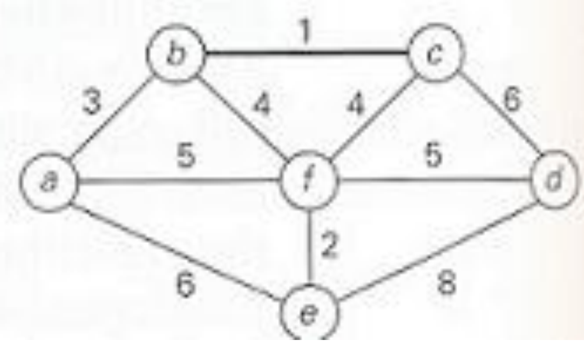
ALGORITHM *Kruskal*(G)

//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 $w(e_{i_1}) \leq \ldots \leq w(e_{i_{|E|}})$
$E_T \leftarrow \emptyset$; $\quad ecounter \leftarrow 0$ $\quad$ //initialize the set of tree edges and its size
$k \leftarrow 0$ $\quad$ //initialize the number of processed edges
while $ecounter < |V| - 1$ **do**
$\quad k \leftarrow k + 1$
$\quad$ **if** $E_T \cup \{e_{i_k}\}$ is acyclic
$\quad\quad E_T \leftarrow E_T \cup \{e_{i_k}\}$; $\quad ecounter \leftarrow ecounter + 1$
return E_T

Kruskal's Algorithm Example

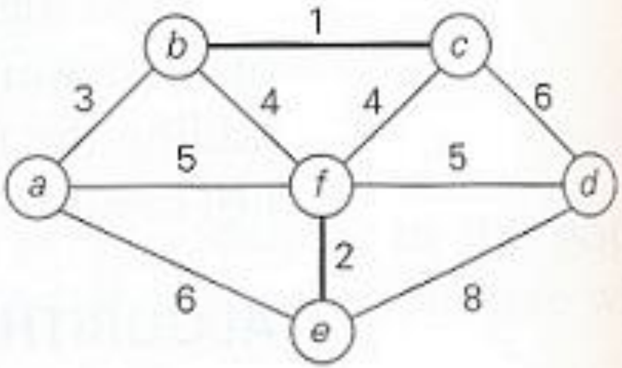


Tree edges	Sorted list of edges	Illustration
	bc 1, ef 2, ab 3, bf 4, cf 4, af 5, df 5, ae 6, cd 6, de 8	

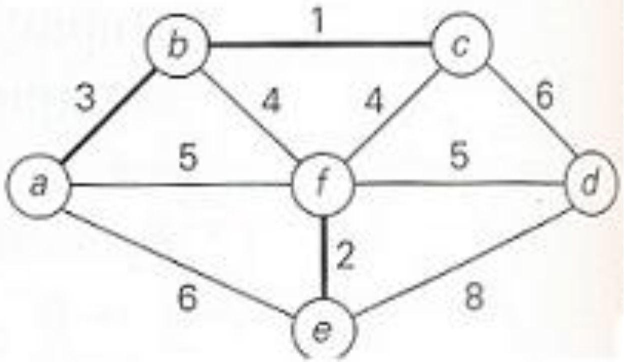


Kruskal's Algorithm Example

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



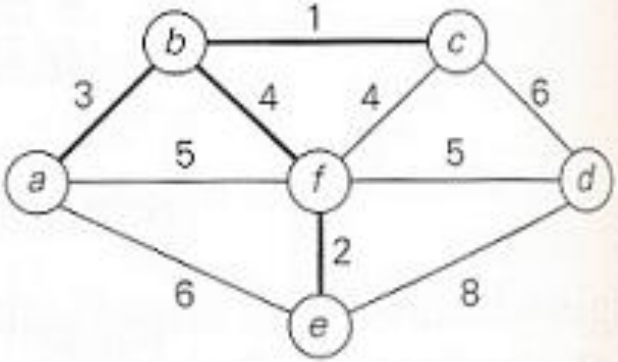
ef		bc	ef	**ab**	bf	cf	af	df	ae	cd	de
2		1	2	3	4	4	5	5	6	6	8



Kruskal's Algorithm Example

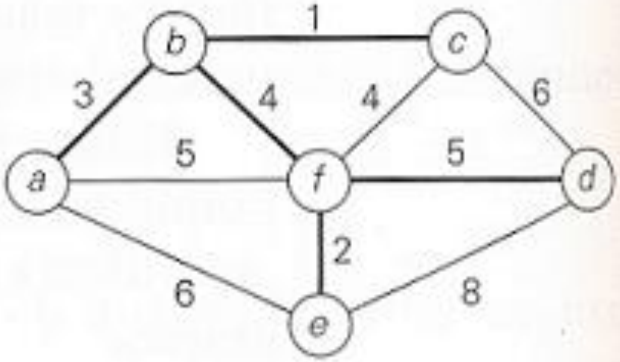
ab
3

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

ROAD MAP

- **Greedy Technique**
  - Knapsack Problem
  - Minimum Spanning Tree Problem
    - Prim's Algorithm
    - Kruskal's Algorithm
  - **Single Source Shortest Paths**
    - **Dijkstra's Algorithm**
  - Huffman Trees

Greedy Technique

- Construct a solution through a sequence of *decision steps*
  - Each expanding a partially constructed solution
  - Until a complete solution is reached
- On each decision step the choice should be
  - **Feasible :** has to satisfy the problem's constraints
  - **Locally optimal:** has to be the best local choice
  - **Irrevocable :** once made, can not be changed

Greedy Technique

```
Greedy Algorithm ( a [ 1 .. N ] )
{
  solution = Ø
  for i = 1 to n
     x = select (a)
     if feasible ( solution, x )
          solution = solution U {x}
  return solution
}
```

Single Source Shortest Paths

- Definition:
  - For a given vertex called ***source*** in a *weighted* connected graph, find shortest paths to all other vertices in the graph

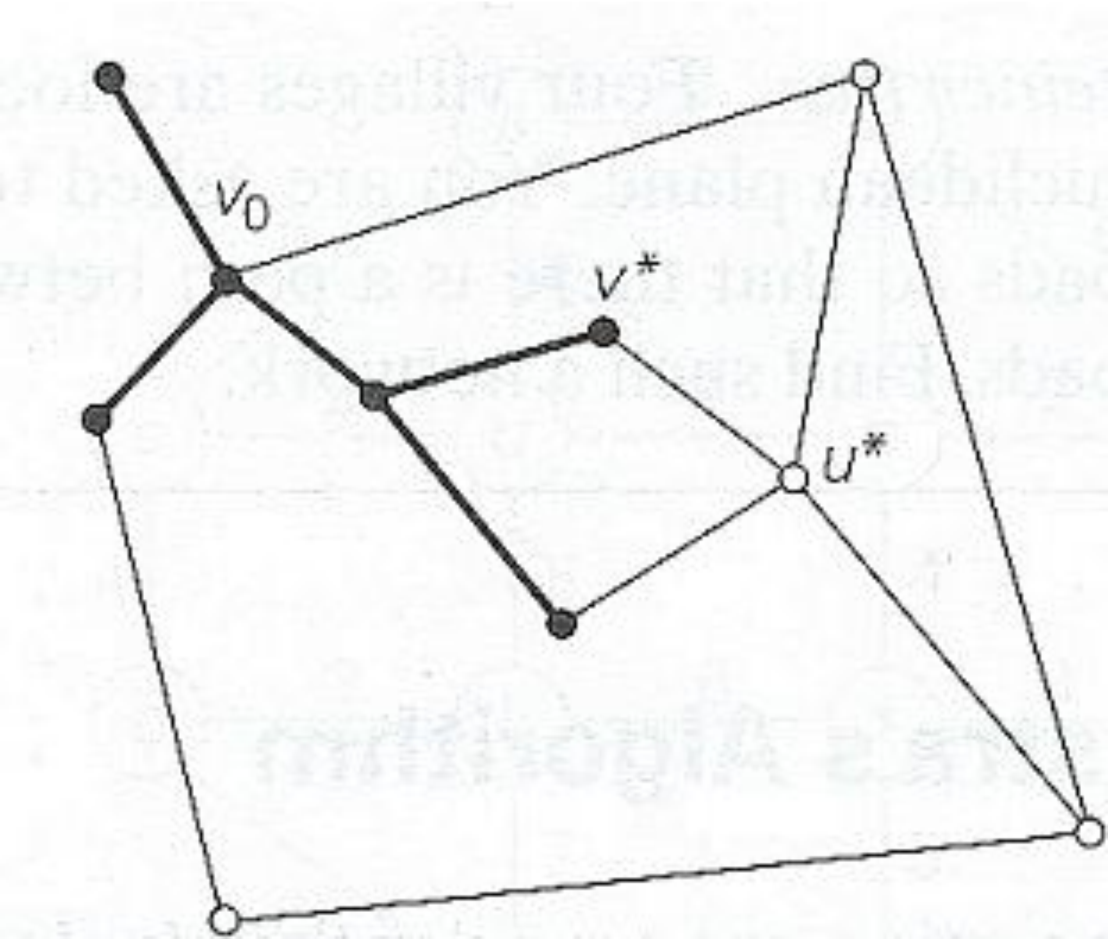
Dijkstra's Algorithm

- Idea :
  - Incrementally add nodes to an empty tree
  - Each time add a node that has the smallest path length

- Approach :

  1. S = { }
  2. Initialize *dist [v]* for all *v*
  3. Insert *v* with min *dist[v]* in *T*
  4. Update *dist[w]* for all *w* not in *S*

Dijkstra's Algorithm



Idea of Dijkstra's algorithm

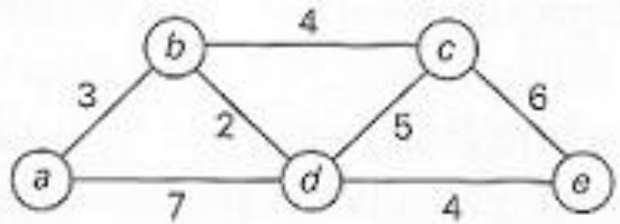
Greedy Technique

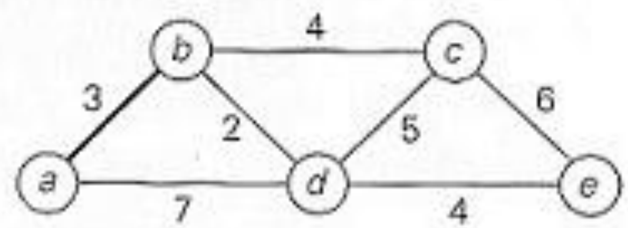
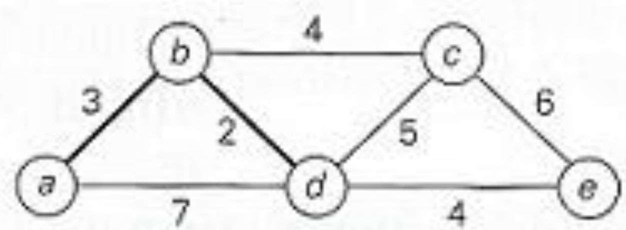
```
Greedy Algorithm ( a [ 1 .. N ] )
{
  solution = Ø
  for i = 1 to n
     x = select (a)
     if feasible ( solution, x )
          solution = solution U {x}
  return solution
}
```

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 ← ∞;   p_v ← null
    Insert(Q, v, d_v)   //initialize vertex priority in the priority queue
d_s ← 0;   Decrease(Q, s, d_s)   //update priority of s with d_s
V_T ← ∅
for i ← 0 to |V| − 1 do
    u* ← DeleteMin(Q)   //delete the minimum priority element
    V_T ← V_T ∪ {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 ← d_u* + w(u*, u);   p_u ← u*
            Decrease(Q, u, d_u)
```

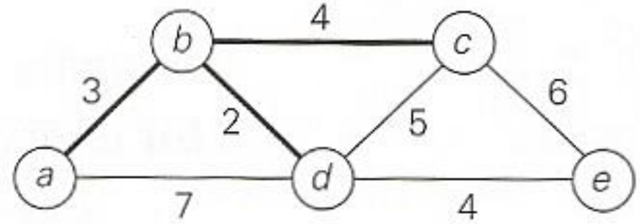
Dijkstra's Algorithm Example



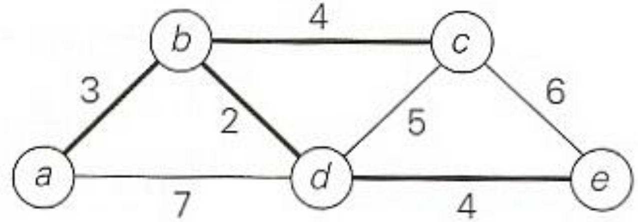
Tree vertices	Remaining vertices	Illustration
a(−, 0)	**b(a, 3)** c(−, ∞) d(a, 7) e(−, ∞)	
b(a, 3)	c(b, 3 + 4) **d(b, 3 + 2)** e(−, ∞)	

Dijkstra's Algorithm Example

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



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



e(d, 9)

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

- **Analysis :**
  - Time efficiency depends on the data structure used for priority queue and for representing an input graph itself
  - For graphs represented by their weight matrix and priority queue implemented as an unordered array, efficiency is in $\Theta(|V|^2)$
  - For graphs represented by their adjacency list and priority queue implemented as a min-heap efficiency is in $O(|E|\log|V|)$
  - A better upper bound for both Prim and Dijkstra's algorithm can be achieved, if *Fibonacci heap* is used

ROAD MAP

- **Greedy Technique**
  - Knapsack Problem
  - Minimum Spanning Tree Problem
    - Prim’s Algorithm
    - Kruskal’s Algorithm
  - Single Source Shortest Paths
    - Dijkstra’s Algorithm
  - **Huffman Trees**

Encoding Text

- Suppose we have to 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 ***codeword***
- We can use a fixed-encoding that assigns to each character
  - Good if each character has same frequency
  - What if some characters are more frequent than others

Encoding Text

- EX: The number of bits in the encoding of 100 characters long text

	a	b	c	d	e	f	
freq	45	13	12	16	9	5	
fixed word	000	...				101	= 300
variable word	0	101	100	111	1101	1100	= 224

Prefix Codes

- A codeword is not prefix of another codeword
  - Otherwise decoding is not easy and may not be possible
- Encoding
  - Change each character with its codeword
- Decoding
  - Start with the first bit
  - Find the codeword
    - A unique codeword can be found – prefix code
  - Continue with the bits following the codeword
- Codewords can be represented in a tree

Prefix Codes

- EX: Trees for the following codewords…

	a	b	c	d	e	f
fixed word	000	...				101
variable word	0	101	100	111	1101	1100

Huffman Codes

- <u>Given:</u> The characters and their frequencies
- <u>Find:</u> The coding tree
- <u>Cost :</u> Minimize the cost

$$Cost = \sum_{c \in C} f(c) \times d(c)$$

- *f(c) : frequency of c*
- *d(c) : depth of c*

Huffman Codes

- What is the greedy strategy?

Huffman Codes

- Approach :

1. Q = forest of one-node trees
   // initialize n one-node trees;
   // label the nodes with the characters
   // label the trees with the frequencies of the chars
2. for i=1 to n-1
3. x = select the least freq tree in Q & delete
4. y = select the least freq tree in Q & delete
5. z = new tree
6. z→left = x and z→right = y
7. f(z) = f(x) + f(y)
8. Insert z into Q

Greedy Technique

```
Greedy Algorithm ( a [ 1 .. N ] )
{
  solution = Ø
  for i = 1 to n
     x = select (a)
     is feasible ( solution, x )
          solution = solution U {x}
  return solution
}
```

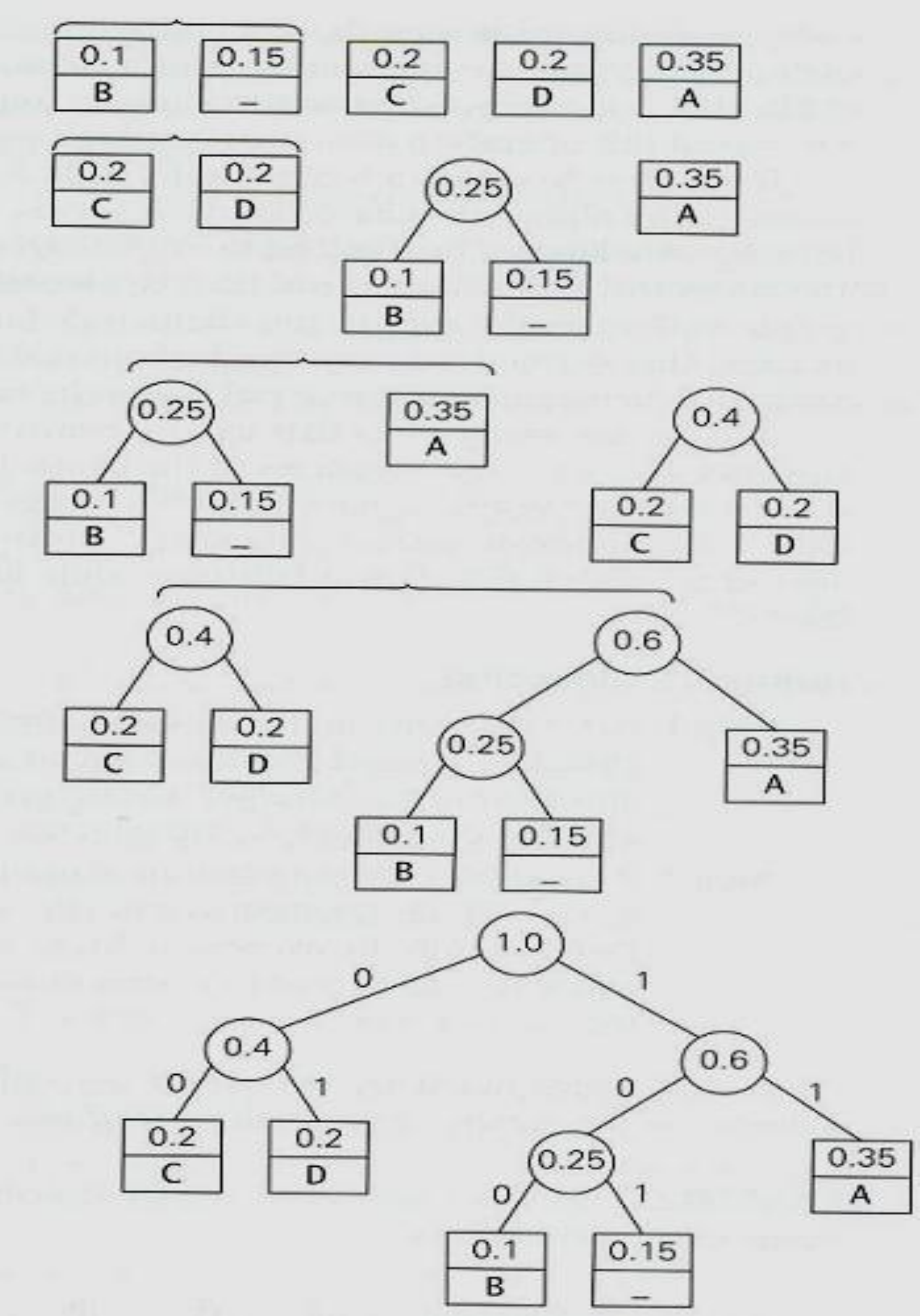
Huffman Codes Example

Consider five characters {A,B,C,D,-} with following occurrence probabilities

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

The Huffman tree construction for this input is as follows

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


0.1
B
0.15
_
0.2
C
0.2
D
0.35
A
0.25
0.4
0.6
1.0
0
1