Dynamic Programming

UNIT 4

Prof. Rajesh R M
Dept. of AIML
RV College of Engineering
Bengaluru

Dynamic Programming

Dynamic Programming

- ➢ Is the most used design technique
- ➢ It is similar to the Divide and conquer, problem is divided into sub problems.
- ➢ The overlapping sub problems solved only once in Dynamic Programming.
- ➢ The solution of the sub problems are stored in the form of table.(Arrays)

To solve the problems

1. Optimal sub structure
2. Recurrence
3. The approach
   a. Top Down approach
   b. Bottom Up approach

Dynamic Programming

Binomial Coefficient

The objective of the Binomial coefficient is to find the number of combinations of 'k' elements in a set of 'n' elements.

It is denoted as c(n,k) or ${}^{n}C_{k}$

The recurrence relation to find Binomial Coefficient

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

$$C(n,k) = \begin{cases} 1 & k=0 \text{ or } k=n \\ 0 & k>n \\ C(n-1,\ k)+c(n-1,\ k-1) & k<n \ \&\ k!=0 \end{cases}$$

Dynamic Programming

Binomial Coefficient

The objective of the Binomial coefficient is to find the number of combinations of ‘k’ elements in a set of ‘n’ elements.

It is denoted as `c(n,k)` or $^{n}C_{k}$

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

```
//Program to demonstrate Binomial Coefficient nCr using recursion
#include <stdio.h>
#include <conio.h>
void main()
{
     int n=5,r=3;
     printf("%%d",nCr(n, r));
}
```

```
int factorial(int n)
{
     if (n == 0)
          return 1;
     return n * factorial(n - 1);
}
int nCr(int n, int r)
{
     return factorial(n)/(factorial(r)*factorial(n-r));
}
```

Dynamic Programming

Example Table Construction

$$C(n,k) = \begin{cases} 1 & k=0 \text{ or } k=n \\ 0 & k>n \\ C(n-1,\ k)+c(n-1,\ k-1) & k<n \ \&\ k!=0 \end{cases}$$

Example ${}^4C_3$ The table is constructed as follows

Here n denotes the no of rows (n+1)

K denotes the number of columns(k+1)

${}^4C_3$ value is 4

k →

n ↓

	0	**1**	**2**	**3**
0	1			
1	1	1		
2	1	2	1	
3	1	3	3	1
4	1	4	6	4

Dynamic Programming

Algorithm Binomial Coefficient (n,k)
// Computes c(n,k) using dynamic programming
// Input: Two non negative integer n and k
// Value of c(n,k)

```
for i <- 0 to n do
      for j <- 0 to min(i,k)
            if j=0 or j=i
                  c[i,j]=1
            else
                  c[i,j]=c[i-1,j]+c[i-1,j-1]
return c[n,k]
```

The time complexity is

$$\sum_{i=0}^{n}\sum_{j=0}^{i\,or\,k} 1 = \sum_{i=0}^{n} k = O(nk)$$

Recurrence Relation

$$A(n,k) = \begin{cases} 0 \text{ if } k=0 \text{ and } k=n \\ A(n-1,k)+A(n-1,k-1)+1 \end{cases}$$

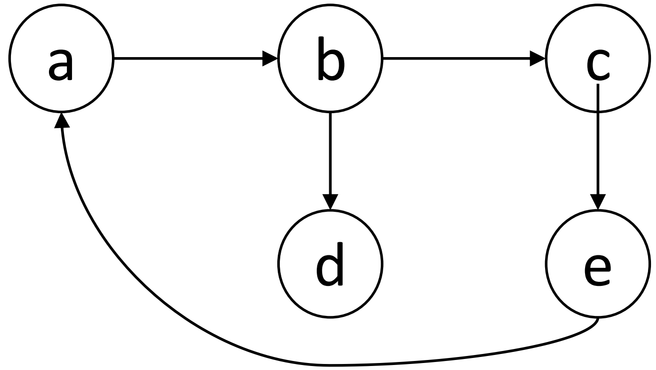
	0	1	2	3
0	1			
1	1	1		
2	1	2	1	
3	1	3	3	1
4	1	4	6	4

Dynamic Programming

Warshal Algorithm

Used to compute the transitive closeness of directed graph.

The transitive closeness of a directed graph with ‘n’ verteces can be defined as nxn Boolean matrix $T=\{t_{ij}\}$ in which the element in the i^{th} row and the element in the j^{th} column is 1 if there exists an non trivial directed path from i^{th} vertex to j^{th} vertex; otherwise it is 0



Input Matrix

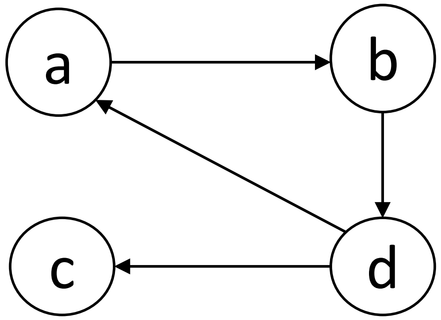
	a	b	c	d	e
a					
b					
c					
d					
e					

Output Matrix

	a	b	c	d	e
a					
b					
c					
d					
e					

Dynamic Programming

Warshal Algorithm



	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

0	1	0	0
0	0	0	1
0	0	0	0
1	0	1	0

R1	0	1	0	0
0	0	1	0	0
0	0	0	0	1
0	0	0	0	0
1	1	1	1	0

R2	0	0	0	1
1	0	1	0	1
0	0	0	0	1
0	0	0	0	0
1	1	1	1	1

R3	0	0	0	0
0	0	1	0	1
0	0	0	0	1
0	0	0	0	0
1	1	1	1	1

R4	1	1	1	1
0	1	1	1	1
0	1	1	1	1
0	0	0	0	0
1	1	1	1	1

Warshal Algorithm

ALGORITHM Warshall(A[1..n, 1..n])

//Implements Warshall’s algorithm for computing the transitive closure

//Input: The adjacency matrix A of a digraph with n vertices

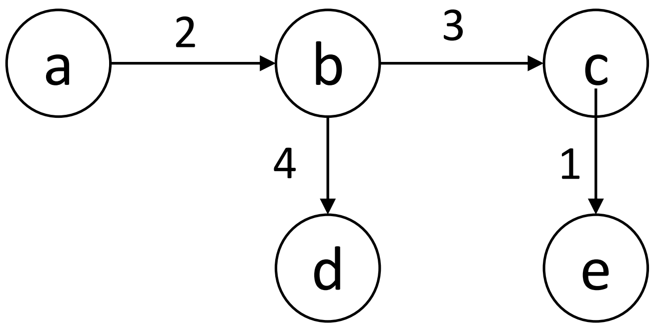
//Output: The transitive closure of the digraph

```
R(0) ← A
for k ← 1 to n do
   for i ← 1 to n do
      for j ← 1 to n do
         R(k)[i, j ] ← R(k−1) [i, j ] or (R(k−1) [i, k] and R(k−1) [k, j ])
return R(n)
```

Dynamic Programming

Warshal Floyd Algorithm

Used to compute the all pair of shortest path of weighted directed graph.



Here distance matrix is constructed to find the minimum path.

- Self nodes is having the value 0.
- If there is a direct path from one vertex to other then write the distance
- If there is no direct path from one vertex to other then write the infinity

Input Matrix D

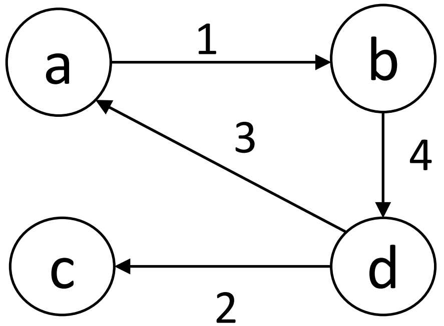
	a	b	c	d	e
a					
b					
c					
d					
e					

Output Matrix D

	a	b	c	d	e
a					
b					
c					
d					
e					

Dynamic Programming

Warshal Floyd Algorithm



D=

	a	b	c	d
a	0	1	∞	∞
b	∞	0	∞	4
c	∞	∞	0	∞
d	3	∞	2	0

D4	a	b	c	d
a	0	1	7	5
b	7	0	6	4
c	∞	∞	0	∞
d	3	4	2	0

D1	0	1	∞	∞
0	0	1	∞	∞
∞	∞	0	∞	4
∞	∞	∞	0	∞
3	3	4	2	0

D2	∞	0	∞	4
1	0	1	∞	5
0	∞	0	∞	4
∞	∞	∞	0	∞
4	3	4	2	0

D3	∞	∞	0	∞
∞	0	1	∞	5
∞	∞	0	∞	4
0	∞	∞	0	∞
2	3	4	2	0

D4	3	4	2	0
5	0	1	7	5
4	7	0	6	4
∞	∞	∞	0	∞
0	3	4	2	0

Dynamic Programming

Warshal Floyd Algorithm

Algorithm Floyds(w(1-n,1-n)

//Implements Floyd's algorithm using dynamic programming

//Input: A weighted graph represented n as the number of vertices in the matrix

//Output: A matrix which gives you the all pair of shortest path

```
d ← w
for i←1 to n do
      for j← 1 to n do
            for k←1 to n do
                  d[i,j]← min(d[i,j],d[i,k]+d[k,j])
```

The time complexity = $O(n^3)$

Dynamic Programming

0/1 Knapsack problem

- In knapsack problem there will be n number of objects known weights w1, . . . , wn.
- values are denoted as v1, . . . , vn.
- knapsack of capacity W.

find the most valuable subset of the items that fit into the knapsack.

The main aim is to add more profitable objects in to the knapsack with a constraint to that weight of the added object should not exceed the knapsack capacity

Let xi denotes whether the object is included or not
Xi=1 (The object is included)
Xi=0(The object is not included)
The knapsack problem can be denoted as
Maximize$\sum_{i=0}^{n} xi\ wi \leq W$

Brute force Method to Solve Knapsack Problem

Let us consider

Objects (n)	1	2	3
Weight	2	3	2
Profit	5	10	8

The maximum Capacity of the Knapsack W = 5

Size W = 5	Profit
1	5
2	10
3	8
1,2	15
1,3	13
2,3	18
~~1,2,3~~ (2+3+2)>5	

Time Complexity :

Dynamic Programming

Dynamic Programming Method to Solve Knapsack Problem

Let us consider

Objects (n)	1	2	3
Weight	2	3	2
Profit	5	10	8

The maximum Capacity of the Knapsack W = 5

$$V[i,j]=\begin{cases} V[i-1, j] , \text{ if } j-wi<0 \\ \text{Max} \{ V[i-1, j], V[i-1, j-wi] + vi \} \text{ if } j-wi >=0 \end{cases}$$

Capacity of the Knapsack →

Objects (n=3)

j

Pi	Wi	↓	0	1	2	3	4	5
		0	0	0	0	0	0	0
5	2	1	0	0	5	5	5	5
10	3	2	0	0	5	10	10	15
8	2	3	0	0	8	10	13	18

Object To be included:

V[3,5] !=V[2,5] means 3rd object should be included

V[2,5-2] -> V[2,3] !=V[1,3] 2nd object should be included

V[1,3-3]-> V[1,0] Capacity of the knapsack is 0 no more space to include object

Dynamic Programming

Exercise

$w_1 = 2,\ v_1 = 12$
$w_2 = 1,\ v_2 = 10$
$w_3 = 3,\ v_3 = 20$
$w_4 = 2,\ v_4 = 15$

$$V[i,j] = \begin{cases} V[i-1, j]\ , \text{ if } j\text{-}wi<0 \\ \text{Max}\ \{ V[i-1, j], V[i-1, j\text{-}wi] + vi \} \end{cases}$$

	capacity j					
i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

V[4,5]>V[3,5] -> 4 is included

Knapsack Using Memory Function

ALGORITHM *MFKnapsack*(i, j)

//Implements the memory function method for the knapsack problem
//Input: A nonnegative integer i indicating the number of the first
// items being considered and a nonnegative integer j indicating
// the knapsack capacity
//Output: The value of an optimal feasible subset of the first i items
//Note: Uses as global variables input arrays $Weights[1..n]$, $Values[1..n]$,
//and table $F[0..n, 0..W]$ whose entries are initialized with -1's except for
//row 0 and column 0 initialized with 0's

```
if F[i, j] < 0
    if j < Weights[i]
        value ← MFKnapsack(i − 1, j)
    else
        value ← max(MFKnapsack(i − 1, j),
                    Values[i] + MFKnapsack(i − 1, j − Weights[i]))
    F[i, j] ← value
return F[i, j]
```

Dynamic Programming

```
value←max(MFKnapsack(i − 1, j), Values[i]+ MFKnapsack(i − 1, j −Weights[i]))
F[i, j ]←value
```

Wt 2 1 3 2
V 8 6 16 11

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	-1	-1	-1	-1	-1
2	0	-1	-1	-1	-1	-1
3	0	-1	-1	-1	-1	-1
4	0	-1	-1	-1	-1	-1

i=4 and j=5
V[4,5]=max{v[3,5], v[3,3]+11}

V[3,5]=max{v[2,5], v[2,2]+16}

V[3,3]=max{v[2,3], v[2,0]+16}

V[2,5]=max{v[1,5], v[1,3]+6}

V[2,2]=max{v[1,2], v[1,1]+6}

V[2,0]=v[1,0]=0

V[1,5]=max{v[0,5], v[0,3]+8}

V[1,4]=max{v[0,4], v[0,2]+8}

V[1,3]=max{v[0,3], v[0,1]+8}

V[1,2]=max{v[0,2], v[0,0]+8}

V[1,1]= V[0,1]=0

Dynamic Programming

```
value←max(MFKnapsack(i − 1, j), Values[i]+ MFKnapsack(i − 1, j −Weights[i]))
F[i, j ]←value
```

Wt 2 1 3 2
V 8 6 16 11

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	8	8	8	8
2	0	-1	8	14	-1	14
3	0	-1	-1	16	-1	24
4	0	-1	-1	-1	-1	27

i=4 and j=5
V[4,5]=max{v[3,5], v[3,3]+11}
=max(24,16+11) = 27
V[3,5]=max{v[2,5], v[2,2]+16}
=max(14,8+16) = 24
V[3,3]=max{v[2,3], v[2,0]+16}
=max(14,0+16) = 16

V[2,5]=max{v[1,5], v[1,3]+6}
=max(8,8+6) = 14
V[2,2]=max{v[1,2], v[1,1]+6}
=max(8,0+6) = 8
V[2,3]=max{v[1,3], v[1,2]+6}
=max(8,8+6) = 14
V[2,0]=v[1,0]=0

V[1,5]=max{v[0,5], v[0,3]+8}
=max(0,0+8) = 8
V[1,4]=max{v[0,4], v[0,2]+8}
=max(0,0+8) = 8
V[1,3]=max{v[0,3], v[0,1]+8}
=max(0,0+8) = 8
V[1,2]=max{v[0,2], v[0,0]+8}
=max(0,0+8) = 8
V[1,1]= V[0,1]=0

Greedy Technique

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- it has to satisfy the problem's constraints

Locally optimal- it has to be the best local choice among all feasible choices available on that step

Irrevocable- once made, it cannot be changed on subsequent steps of the algorithm

Greedy Technique

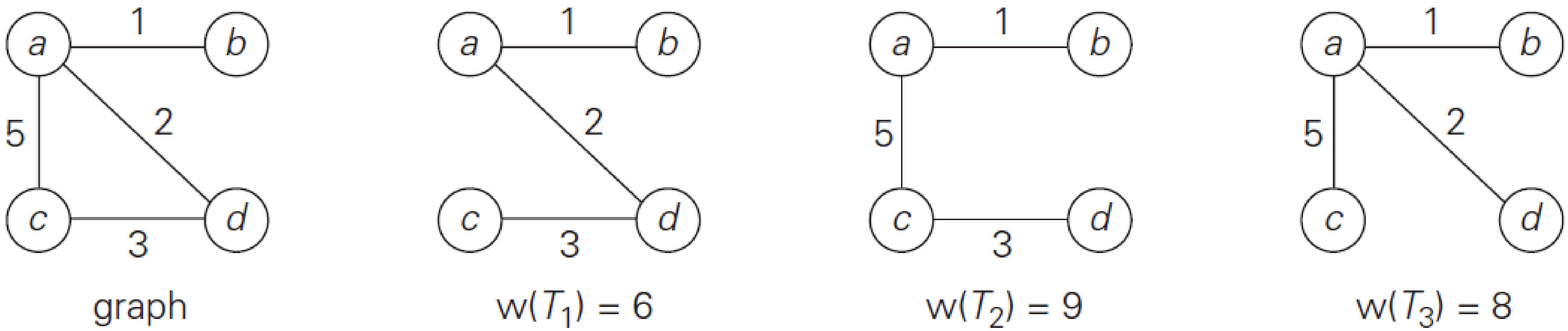
A General algorithm to for solving a problem using Greedy approach is

```
Algorithm Greedy(A,n)
//Solves a problem using greedy method
//A is the input in which n as size of elements
solution,← 0
for i← 1 to n
      x=choose(A)
      if(feaseable(x))
            solution← union(solution,x)
Return solution
```

Greedy Technique

Spanning Tree

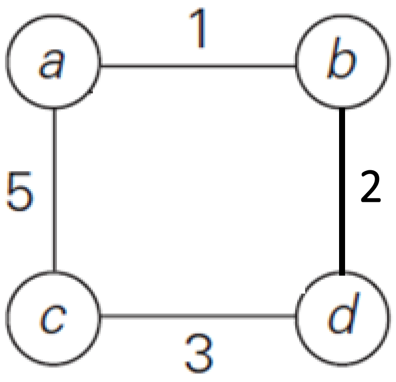
A ***spanning tree*** of an undirected connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph. If such a graph has weights assigned to its edges, a ***minimum spanning tree*** is its spanning tree of the smallest weight, where the ***weight*** of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



Greedy Technique

Prims 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, the algorithm expands the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree.

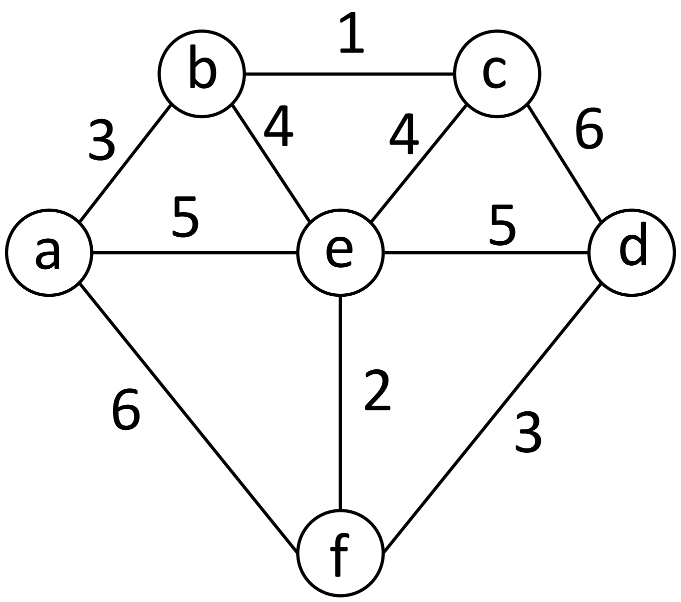


graph

```
V={a,b,c,d}
Et=0
```

Prims 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, the algorithm expands the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree.



V={a,b,c,d,e,f}

Et=0 (Solution)

Iteration 1

Vt={a}

V-Vt={b,c,d,e,f}

{a,b},{a,e},{a,f}
3 5 6

Et={a,b} is minimum

Iteration 2

Vt={a,b}

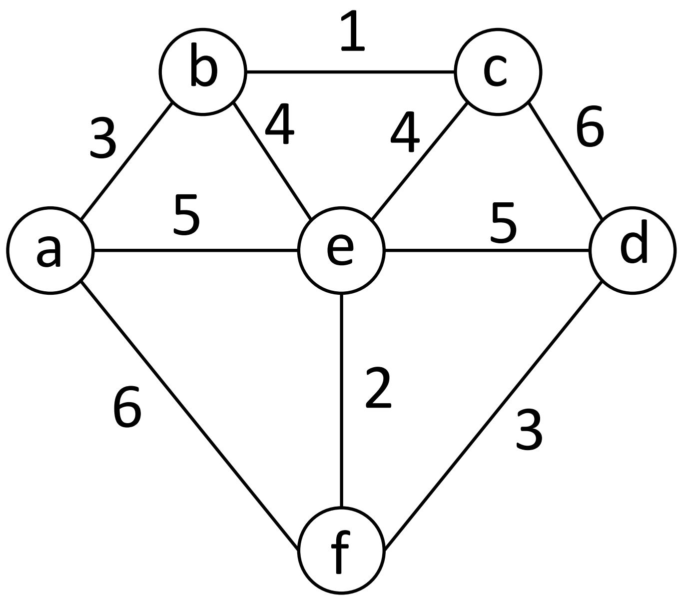
V-Vt={c,d,e,f}

{a,e},{a,f},{b,c},{b,e}
5 6 1 4

Et={{a,b},{b,c}}

Greedy Technique

Prims Algorithm



V={a,b,c,d,e,f}
Et=0 (Solution)

Iteration 1

Vt={a}
V-Vt={b,c,d,e,f}
(a,b),(a,e),(a,f)
3 5 6
Et={a,b} is minimum

Iteration 2

Vt={a,b}
V-Vt={c,d,e,f}
(a,e),(a,f),(b,c),(b,e)
5 6 1 4
Et={(a,b),(b,c)}

Iteration 3

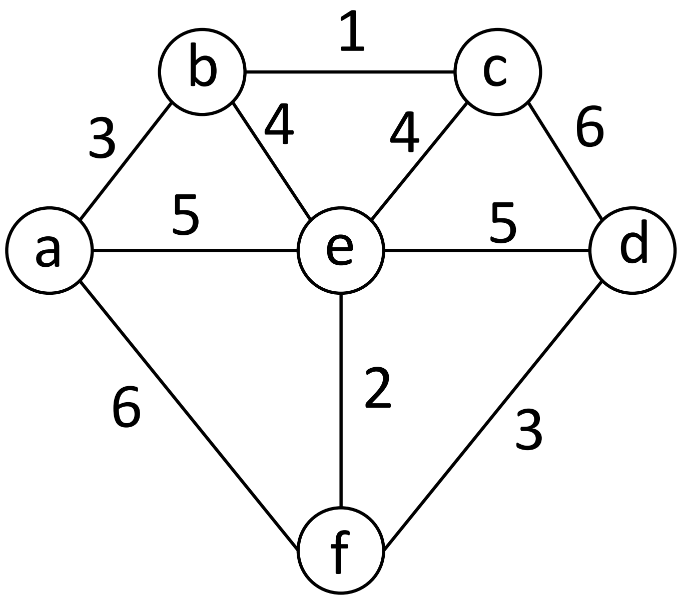
Vt={a,b,c}
V-Vt={d,e,f}
(a,e) (a,f) (b,e) (c,d),(c,e)
5 6 4 6 4
Et={={(a,b),(b,c),(c,e)}

Iteration 4

Vt={a,b,c,e}
V-Vt={d,f}
(a,f) (c,d) (e,d) (e,f)
6 6 5 2
Et={={(a,b),(b,c),(c,e),(e,f)}

Greedy Technique

Prims Algorithm



Iteration 5

Vt={a,b,c,e,f}

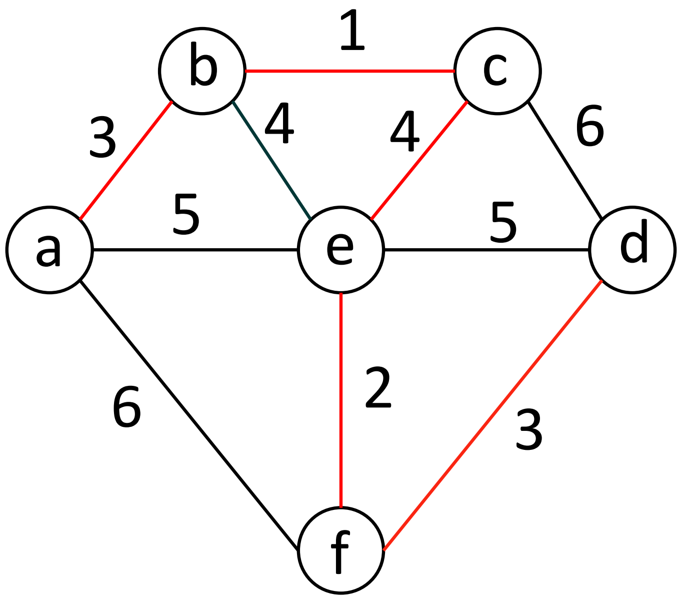
V-Vt={d}

(c,d) (e,d) (f,d)

6 5 3

Et={={(a,b),(b,c),(c,e),(e,f),(f,d)}

Weight=13



Greedy Technique

Prims Algorithm

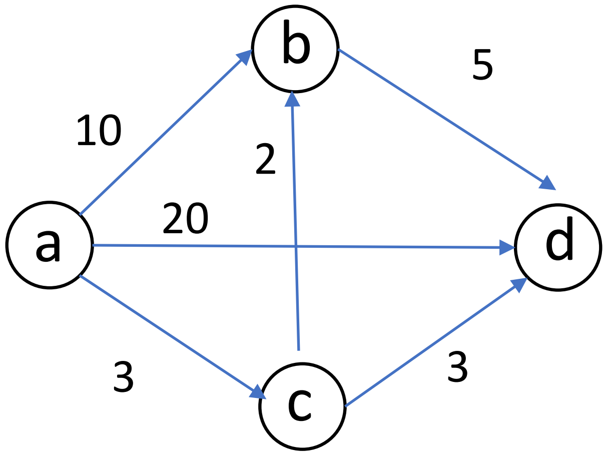
ALGORITHM $Prim(G)$

//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
$E_T \leftarrow \varnothing$
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

Greedy Technique

Dijkstra's Algorithm

This algorithm is Used to solve ***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.




```
V={a,b,c,d}
W={a,b,c,d}
Sol=0
```

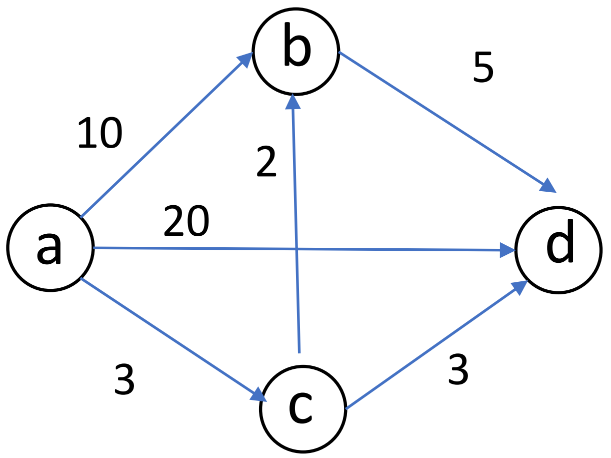
V	D[]	P[]
a	0	Nil
b	∞	Nil
c	∞	Nil
d	∞	Nil

Greedy Technique

Dijkstra's Algorithm

V={a,b,c,d}
W={a,b,c,d}
Sol=0

V	D[]	P[]
a	0	Nil
b	∞	Nil
c	∞	Nil
d	∞	Nil



Iteration 1
W={a,b,c,d}
Extract min = a
Sol=a

V	D[]	P[]
a	0	Nil
b	10	a
c	3	a
d	20	a

Iteration 2
W={b,c,d}
Extract min = c
Sol={a,c}

V	D[]	P[]
a	0	Nil
b	5	c
c	3	a
d	6	c

Iteration 3
W={b,d}
Extract min = b
Sol={a,c,b}

V	D[]	P[]
a	0	Nil
b	5	c
c	3	a
d	6	c

Iteration 4
W={b,d}
Extract min = d
Sol={a,c,b,d}

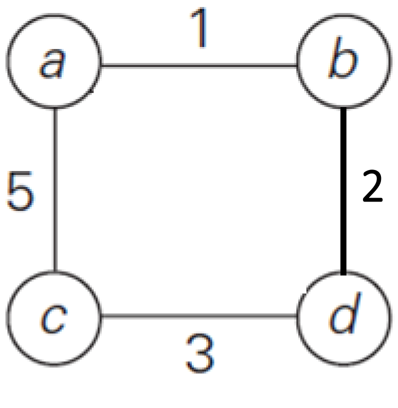
V	D[]	P[]
a	0	Nil
b	5	c
c	3	a
d	6	c

Greedy Technique

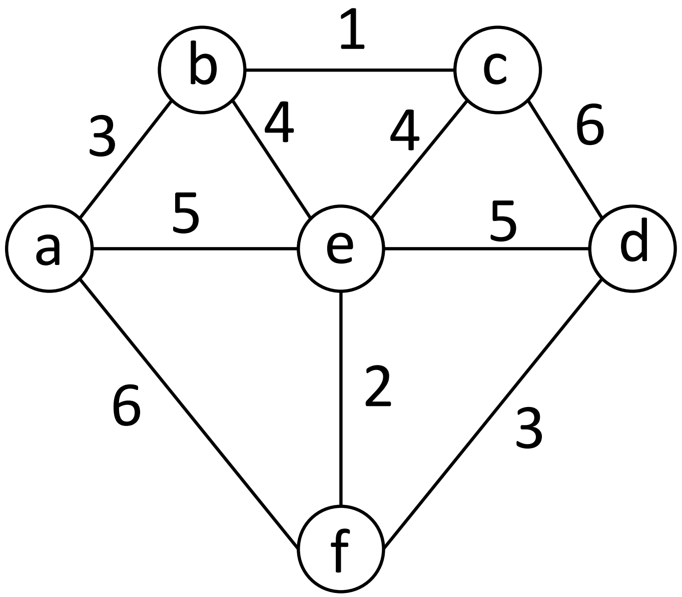
ALGORITHM *Dijkstra*(G, s)

//Dijkstra's algorithm for single-source shortest paths
//Input: A weighted connected graph $G = \langle V, E\rangle$ 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 priority queue to empty
for every vertex v in V
    $d_v \leftarrow \infty$; $p_v \leftarrow$ **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
$V_T \leftarrow \varnothing$
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 $d_{u^*} + w(u^*, u) < d_u$
            $d_u \leftarrow d_{u^*} + w(u^*, u)$; $p_u \leftarrow u^*$
            $Decrease(Q, u, d_u)$

Greedy Technique



graph



Huffman Coding

Huffman coding is a **lossless data compression** algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters.

Suppose we have to encode a text that comprises symbols from some *n*-symbol alphabet by assigning to each of the text's symbols some sequence of bits called the ***code word***.

Types of coding

(i) Fixed length Code: Each letter represented by an equal number of bits. With a fixed length code, at least 3 bits per character:

(ii) A variable-length code: It can do considerably better than a fixed-length code, by giving many characters short code words and infrequent character long code words.

Huffman Coding

Input:

A = 8 = 40%

B = 2 = 10%

C = 4 = 20%

D = 3 = 15%

_ = 3 = 15%

If Huffman Coding is employed in this case for data compression, the following information must be determined for decoding:

- For each character, the Huffman Code
- Huffman-encoded message length (in bits), average code length

Huffman Coding

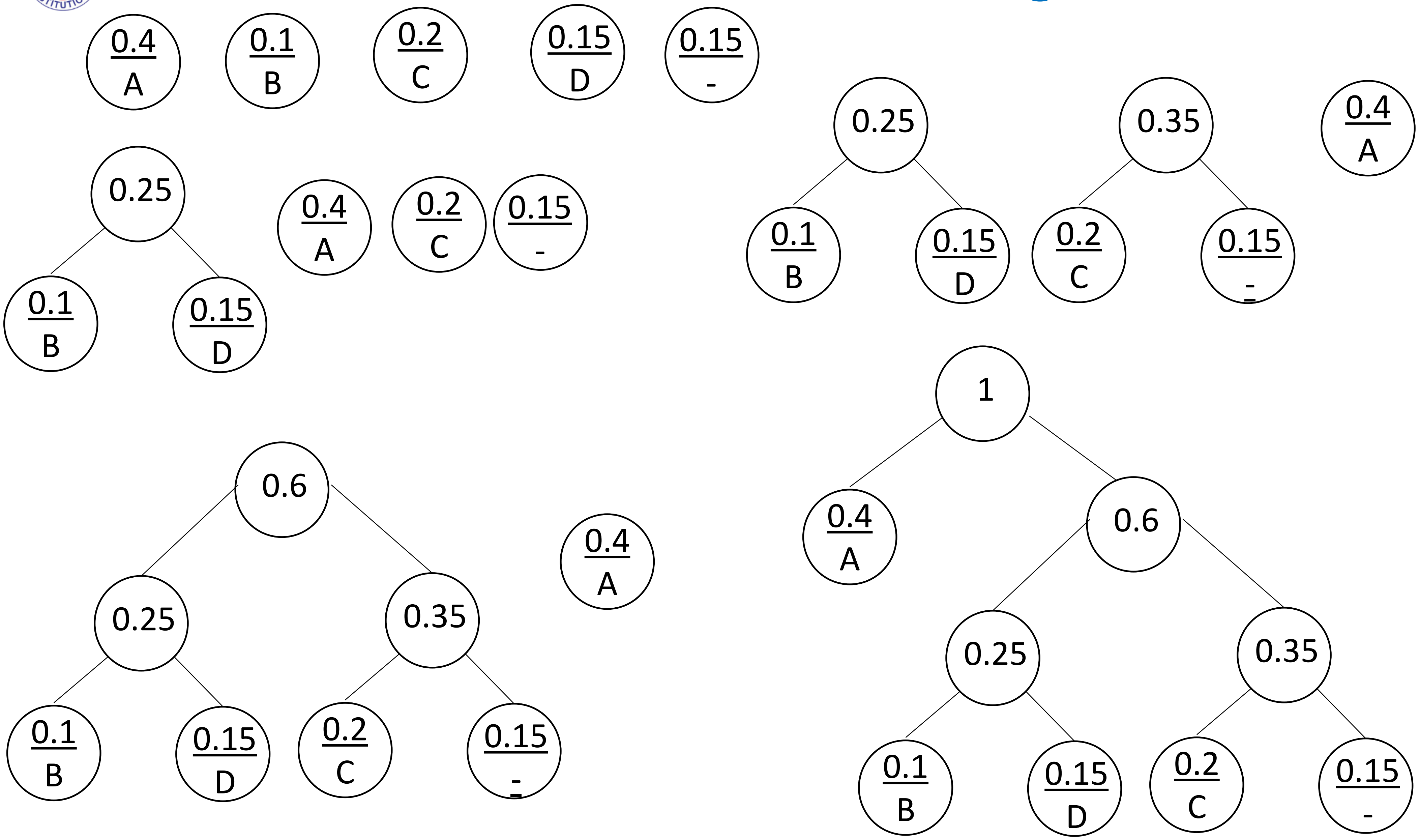
Huffman's algorithm

Step 1 Initialize n one-node trees and label them with the symbols of the alphabet given. Record the frequency of each symbol 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 this section's 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.

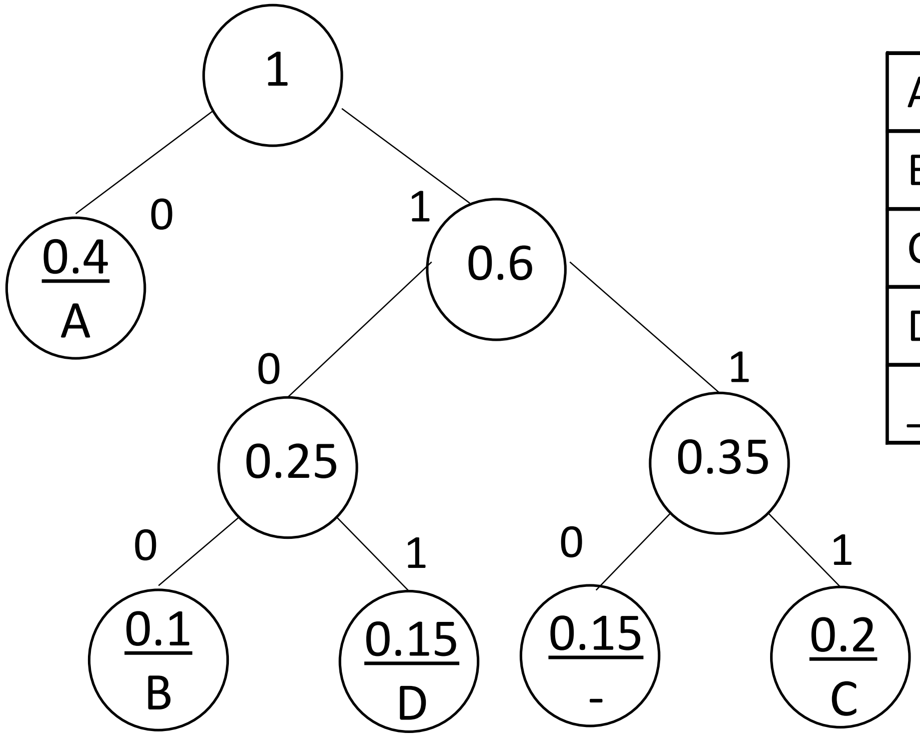
Huffman Coding

A = 8 = 40%
B = 2 = 10%
C = 4 = 20%
D = 3 = 15%
_ = 3 = 15%



Huffman Coding

A = 8 = 40%
B = 2 = 10%
C = 4 = 20%
D = 3 = 15%
_ = 3 = 15%



A	0
B	100
C	110
D	101
_	111

Without Huffman codes ASCII Representation is used
Size of ASCII value is 8 bits

8x8+8x2+8x4+8x3+8x3 = 160 bits
With Huffman Code = 13 bits

Average number of bits per character
= 1x0.4 + 3x0.1 + 3x0.2 + 3x0.15 + 3x0.15
= 0.4 + 0.3 + 0.6 + 0.45 +0.45
=2.2

Compression Ratio

$$= \frac{\text{(Total no of bits - Average no of bits)}}{\text{Total bits}} \times 100$$

Compression Ratio

$$= \frac{(3-2.2)}{3} \times 100$$

Compression Ratio
26%