Design and Analysis of Algorithms (CSC-314)

Department of B.Sc. CSIT

Godawari College

- Unit-4:Greedy Algorithms

- **Introduction**:
- In the algorithms w e have studied so far correctness tended to b e easier than efficiency. In optimization problems we are interested in finding a thing which maximizes or minimizes some function.
- In computer science an optimization problem is the problem of finding the best solution from all feasible solutions.
- Optimization problems can be divided into two categories, depending on whether the variables are continuous or discrete:
- An optimization problem with discrete variables is known as a discrete optimization, in which an object such as an

Unit-4:Greedy Algorithms

- **Introduction**:
- In the algorithms w e have studied so far correctness tended to b e easier than efficiency. In optimization problems we are interested in finding a thing which maximizes or minimizes some function.
- In designing algorithms for optimization problem we must prove that the algorithm in fact gives the best possible solution.
- Greedy algorithms which makes the b est lo cal decision at each step occasionally produce a global optimum but you need a proof !

Unit-4:Greedy Algorithms

- **Introduction**:
- A solution (set of values for the decision variables) for which all of the constraints in the Solver model are satisfied is called a feasible solution. In some problems, a feasible solution is already known; in others, finding a feasible solution may be the hardest part of the problem.
- An optimal solution is a feasible solution where the objective function reaches its maximum (or minimum) value – for example, the most profit or the least cost.
- A globally optimal solution is one where there are no other feasible solutions with better objective function values.
- A locally optimal solution is one where there are no other

Unit-4:Greedy Algorithms

- **Introduction**:
- Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit.
- In other words A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage.
- In many problems, a greedy strategy does not produce an optimal solution, but a greedy heuristic can yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.

Unit-4:Greedy Algorithms

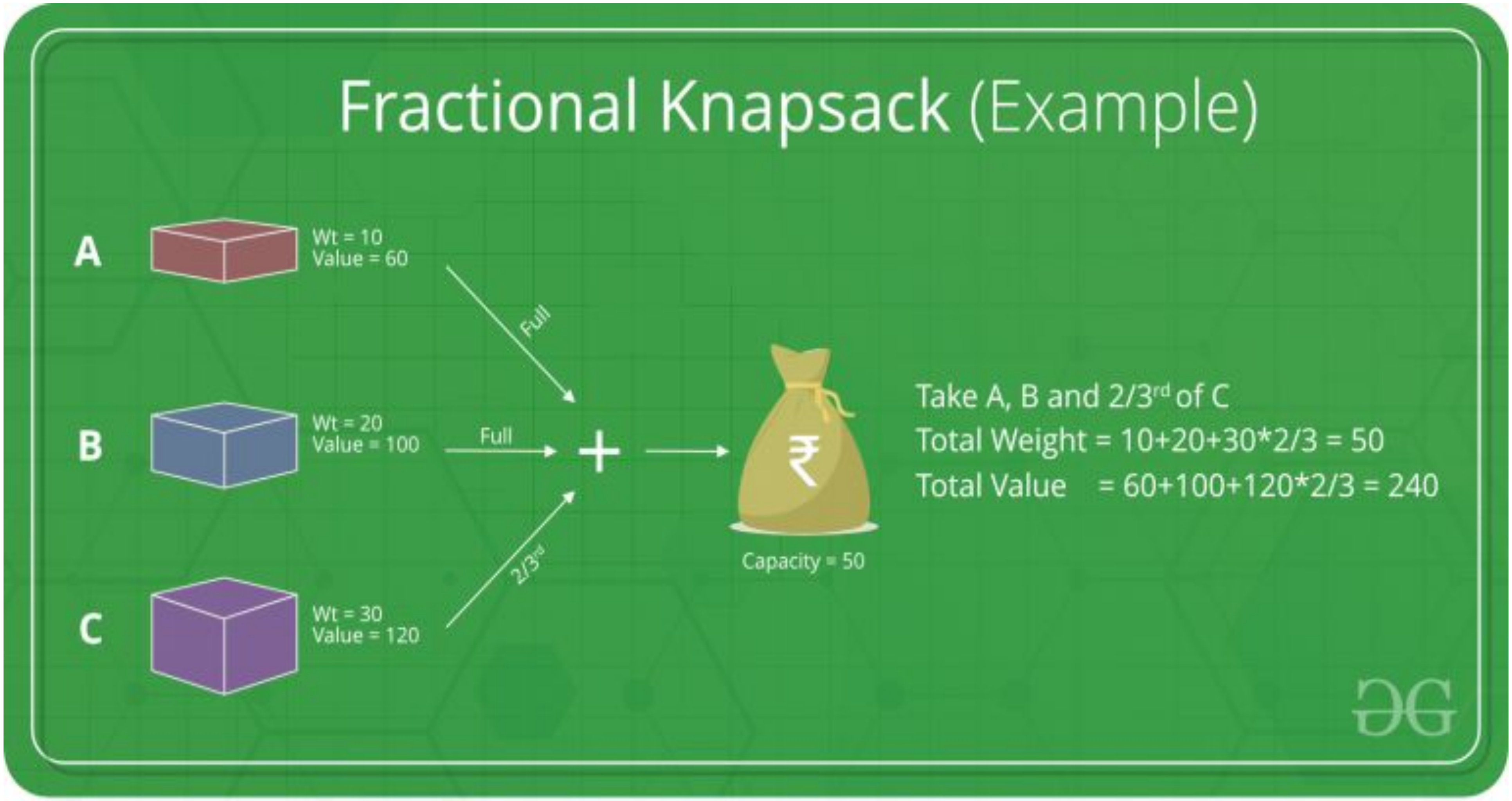
- **Introduction**:
- To prove that a greedy algorithm is optimal we must show the following two characteristics are exhibited.
  - **Greedy Choice Property**
    - We can make whatever choice seems best at the moment and then solve the subproblems that arise later.
    - It iteratively makes one greedy choice after another, reducing each given problem into a smaller one. In other words, a greedy algorithm never reconsiders its choices.
    - Thus, optimal solutions can be obtained by creating greedy choices

Unit-4:Greedy Algorithms

- **Introduction**:
- Elements of greedy strategy:
  - A candidate set
    - A optimal solution is created from this set.
  - A selection function
    - Used to select the best candidate to be added to solution.
  - A feasibility function
    - Used to determined whether a candidate can be used to contribute to the solution.
  - An objective function
    - Used to assign the value to solution or partial

Unit-4:Greedy Algorithms

- **Fractional Knapsack:**



Unit-4:Greedy Algorithms

- **Fractional Knapsack:**
- **Statement:**
  - A thief has a bag or knapsack that can contain maximum weight W of his loot.
  - There are n items and the weight of i^{th} item is w_i and it worth v_i .
  - Any amount of item can be put into the bag i.e. x i fraction of item can be collected, where $0<=x_i<=1$.
  - Here, the objective is to collect the items that maximize the total profit earned.
  - Here we arrange the items by ratio v_i/w_i.

Unit-4:Greedy Algorithms

- **Fractional Knapsack:**
- Take as much of the item with the highest value per weight (v_i / w_i) as you can.
- If the item is finished then move on to next item that has highest (v_i / w_i), continue this until the knapsack is full.
- v[1 … n] and w[1 … n] contain the values and weights respectively of the n objects sorted in non increasing ordered of v[i]/w[i] .
- W is the capacity of the knapsack, x[1 … n] is the solution vector that includes fractional amount of items and n is the number of items.

Unit-4:Greedy Algorithms

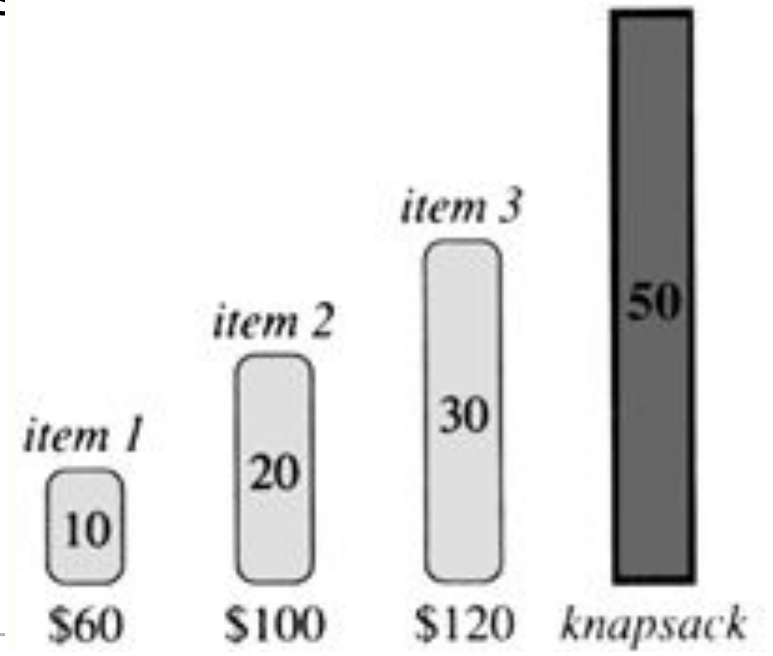
- **Fractional Knapsack: Pseudo Code**

```
GreedyFracKnapsack(W,n) {
    for(i=1; i<=n; i++)
        x[i] = 0.0;
    tw = W;
    for(i=1; i<=n; i++) {
        if(w[i] > tw)
        break;
    else
    x[i] = 1.0;
```

- Here is a single loop involved so running time is O(n).
- But the main requirements is sorting the v[1...n] and w[1...n], so we can use sorting algorithms to sort it in O(nlogn).
- Thus its running time complexity is O(nlogn)

Unit-4:Greedy Algorithms

- Examples: consider 3 items along with their weights and values respectively.
  - I1 w1 = 10 v1 = 60
  - I2 w2 = 20 v2 = 100
  - I3 w3 = 30 v3 = 120
  - The knapsack has capacity W= 50, then find optimal profit earned by using fractional knapsack
  -



Unit-4:Greedy Algorithms

- Examples: consider 3 items along with their weights and values respectively.
  - I1 w1 = 10 v1 = 60
  - I2 w2 = 20 v2 = 100
  - I3 w3 = 30 v3 = 120
  - The knapsack has capacity W= 50, then find optimal profit earned by using fractional knapsack.
  - **Solution**: **Step 1**

Items	$\mathbf{W_i}$	$\mathbf{V_i}$
I1	10	60
I2	20	100
I3	30	120

Unit-4:Greedy Algorithms

- **Step 2: Calculate $P_i = V_i / W_i$**

Items	W_i	V_i	$P_i = V_i / W_i$
I1	10	60	6
I2	20	100	5
I3	30	120	4

- **Step 3: Arranging the items in non increasing order of Pi**

Items	W_i	V_i	$P_i = V_i / W_i$
I1	10	60	6
I2	20	100	5
I3	30	120	4

Unit-4:Greedy Algorithms

- **Step 3: Arranging the items in non increasing order of Pi**

Items	$\mathbf{W_i}$	$\mathbf{V_i}$	$\mathbf{P_i = V_i / W_i}$
I1	10	60	6
I2	20	100	5
I3	30	120	4

- Now Filling knapsack according to decreasing order of Pi
- Since, 30 W = 120 v
- i.e. 1 w =120/30 v = 4 v
- 20 w= 20 *4 v =80 v
- Thus, maximum value = v1+v2+ new (v3) = 60+100+80 = 240 v

Unit-4:Greedy Algorithms

- **Example (Practice):** Consider five items along with their respective weights and values.
  - I ={I1, I2, I3, I4, I5}
  - W= {5,10,20,30,40}
  - V ={ 30, 20, 100, 90, 160}

  - The knapsack has capacity W=60, then find optimal profit earned by using fractional knapsack.

- Unit-4:Greedy Algorithms

- **Job Sequencing with Deadline**

- We are given a set of n jobs. Associated with each job I, di≥0 is an integer deadline and pi≥0 is profit.
- For any job i profit is earned iff job is completed by deadline. To complete a job one has to process a job for one unit of time.

Unit-4:Greedy Algorithms

- **Job Sequencing with Deadline**
- Here, let there are n numbers of job J={j1,j2,……jn}
- With corresponding deadline ={d1,d2,...dn}
- Profit if job is completed within their deadline ={p1,p2….pn}
- Only one machine is available for processing jobs.
- Only one job is processed at a time on the machine.
- So our aim is to find feasible subset of jobs such that profit is maximum.

Unit-4:Greedy Algorithms

- **Job Sequencing with Deadline**
- Example: From the set of given job we have to find the sequence of job, which will be completed within their deadlines and will give maximum profit. Each job is associated with a deadline and profit.
- Job {J1,j2,j3,j4,j5}
- Deadline {2,1,3,2,1}
- Profit {60,100,20,40,20}

Unit-4:Greedy Algorithms

- **Job Sequencing with Deadline**
- **Solution:Step 1**Sort the job according to their profit in non increasing order:
- Job {J2,j1,j4,j3,j5}
- Deadline {1,2,2,3,1}
- Profit {100,60,40,20,20}

Job	Feasible/non-feasible	Processing Sequence	Total Profit
J2	Feasible	{J2}	100
J1	Feasible	{J2,J1}	100+60=160
J4	Not Feasible	{J2,J1}	160
J3	Feasible	{J2,J1,J3}	160+20=180
J5	Not Feasible	{J2,J1,J3}	180

- Thus, the sequence of job {J2,J1,J3} are being executed within their deadline and gives maximum profit i.e. 100+60+20=180.

Unit-4:Greedy Algorithms

Job Sequencing with Deadline

- **Example: Find the optimal Schedule for the following seven task given values and deadline.**

Job	{j1,j2,j3, j4,j5,j6,j7}
Deadline	{4,2,3,4,1,4,6}
Values	{70,60,50,40,30,20,10}

Unit-4:Greedy Algorithms

- **Job Sequencing with Deadline :Pseudo Code**
- Let us consider, a set of n given jobs which are associated with deadlines and profit is earned, if a job is completed by its deadline. These jobs need to be ordered in such a way that there is maximum profit.
- It may happen that all of the given jobs may not be completed within their deadlines.
- Assume, deadline of ith job J_i is d_i and the profit received from this job is p_i. Hence, the optimal solution of this algorithm is a feasible solution with maximum profit.

Unit-4:Greedy Algorithms

- **Job Sequencing with Deadline :Pseudo Code**
- Job-Sequencing-With-Deadline (D, J, n, k)
- D(0) = J(0) = 0
- k = 1
- J(1) = 1 // means first job is selected
- for i = 2 … n do
-     r = k
-     while D(J(r)) > D(i) and D(J(r)) ≠ r do

- **Analysis:**
- In this algorithm, we are using two loops, one is within another. Hence, the complexity of this algorithm is O(n*n)

Unit-4:Greedy Algorithms

- **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.
- The most frequent character gets the smallest code and the least frequent character gets the largest code.
- The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character.

Unit-4:Greedy Algorithms

- **Huffman Coding:**
- Let us understand prefix codes with a counter example.
- Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1.
- This coding leads to ambiguity because code assigned to c is the prefix of codes assigned to a and b.
- If the compressed bit stream is 0001, the de-compressed output may be “cccd” or “ccb” or “acd” or “ab”.

Unit-4:Greedy Algorithms

- Huffman Coding:
- Here are mainly two major parts in Huffman Coding
  - Build a Huffman Tree from input characters.
  - Traverse the Huffman Tree and assign codes to characters.

Unit-4:Greedy Algorithms

- **Huffman Coding: Compression Technique:**
- The technique works by creating a binary tree of nodes.
- These can stored in a regular array, the size of which depends on the number of symbols, n. A node can either be a leaf node or an internal node.
- Initially all nodes are leaf nodes, which contain the symbol itself, its frequency and optionally, a link to its child nodes.
- As a convention, bit '0' represents left child and bit '1' represents right child. Priority queue is used to store the nodes, which provides the node with lowest frequency when popped. The process is described below:

Unit-4:Greedy Algorithms

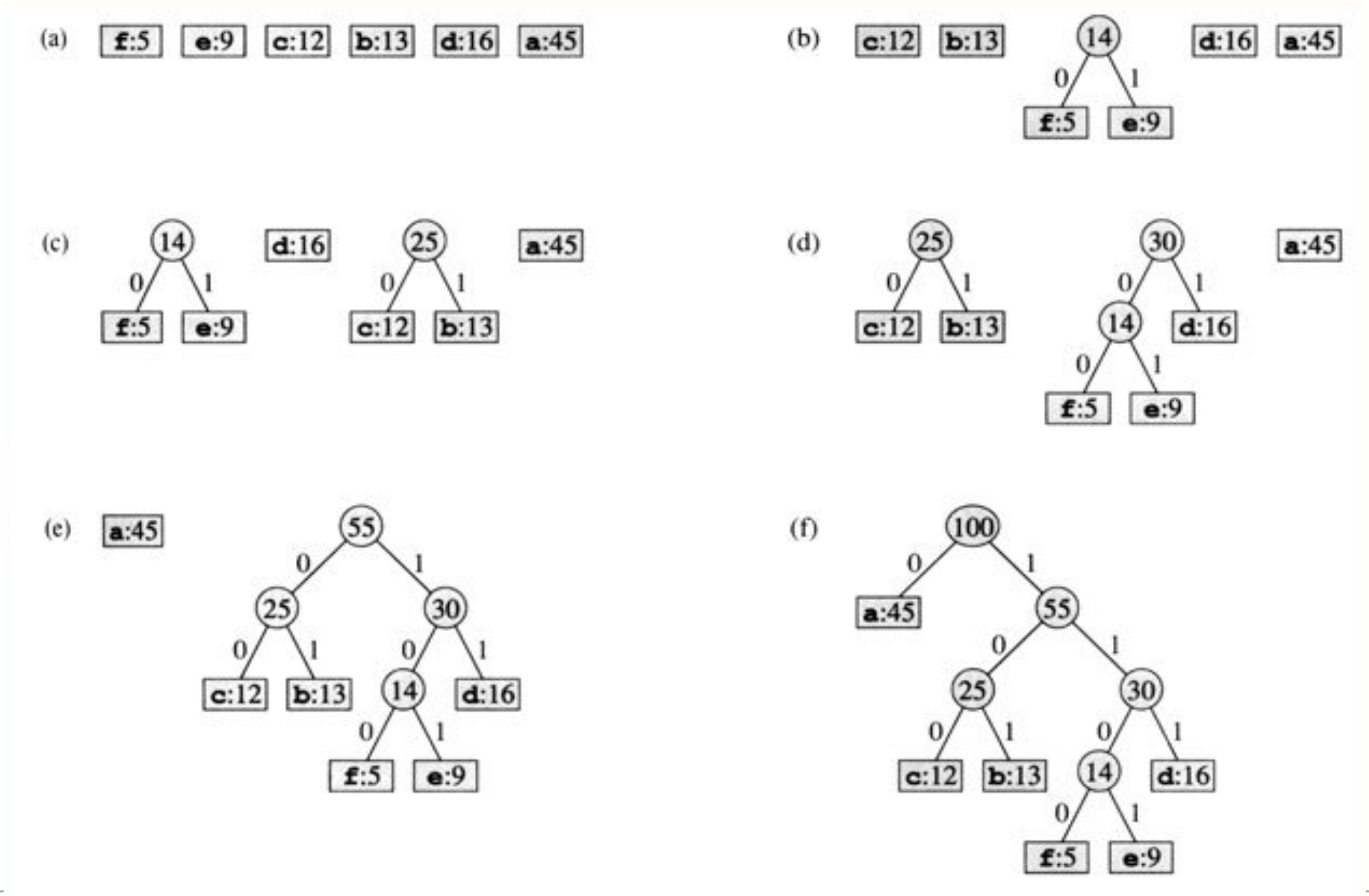
- Huffman Coding: Example
- Consider following character with frequencies:
  - Character ={a,b,c,d,e,f}
  - Frequencies ={45,13,12,16,9,5}
- **Solution**: Sorting according to frequencies in non decreasing order.

Character	Frequencies
f	5
e	9
c	12
b	13
d	16
a	45

- Here before using Huffman algorithm at least we need 3 bit to represent 5 character. so total number bits are= 5*3+9*3+12*3+13*3+16*3+45*3 =300 bits.

Unit-4:Greedy Algorithms

- Huffman Coding: Now using Huffman algorithm
-



Unit-4:Greedy Algorithms

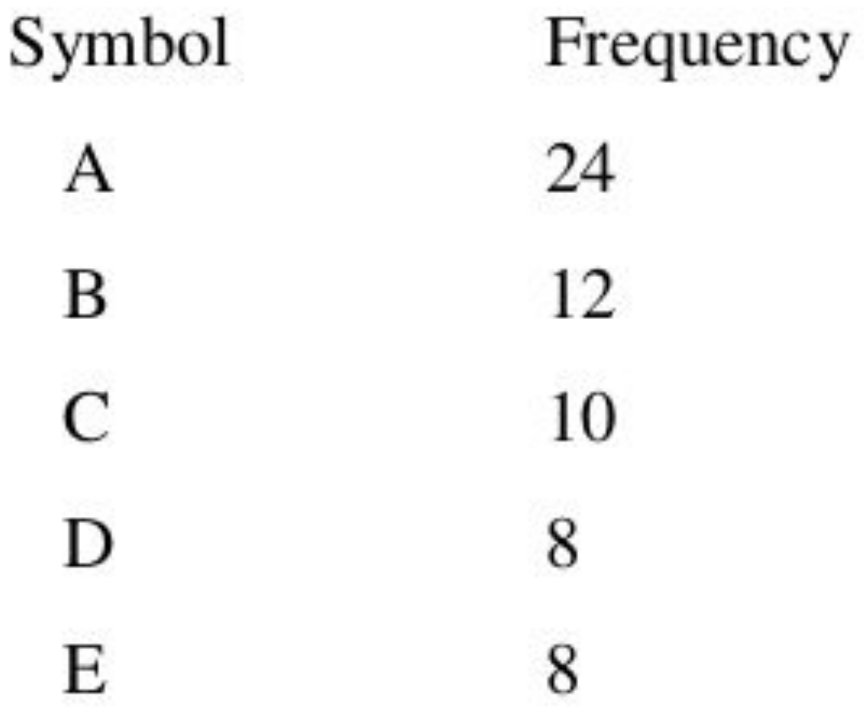
- Huffman Coding:
- Now from variable length code we get following code sequence;

Character	Frequencies	Code
f	5	1100
e	9	1101
c	12	100
b	13	111
d	16	101
a	45	0

- Number of bits required : 5*4+9*4+12*3+13*3+16*3+45*1 = 185 bits
  - (300-185)/300*100 % = 38.33%
  - i.e. we can save 38.33% space by using huffman coding.

Unit-4:Greedy Algorithms

- Huffman Coding: Example
- Trace the huffman algorithms for the given data.



Symbol	Frequency
A	24
B	12
C	10
D	8
E	8

Unit-4:Greedy Algorithms

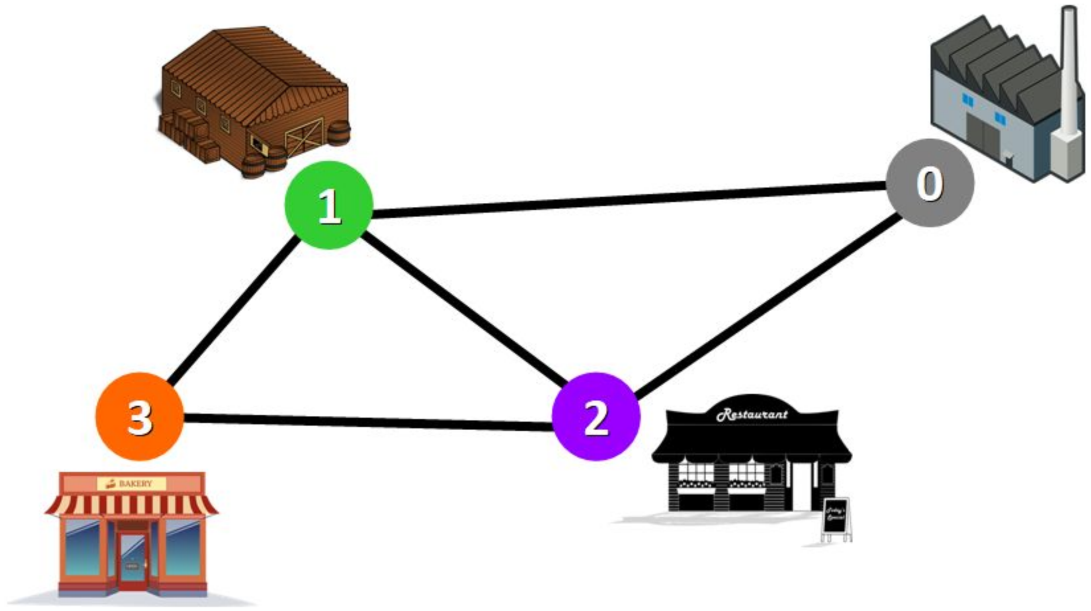
- **Huffman Code:**
- **The pseudo-code looks like:**

```
Procedure Huffman(C):     // C is the set of n characters and related information
n = C.size
Q = priority_queue()
for i = 1 to n
    n = node(C[i])
    Q.push(n)
end for
while Q.size() is not equal to 1
    Z = new node()
```

- Analysis:
- Here is a two loops involved running time of each loop is O(n).
- we can use sorting algorithms to sort input symbols in ascending order of their frequencies in O(nlogn).
- Thus its running time complexity is O(nlogn)

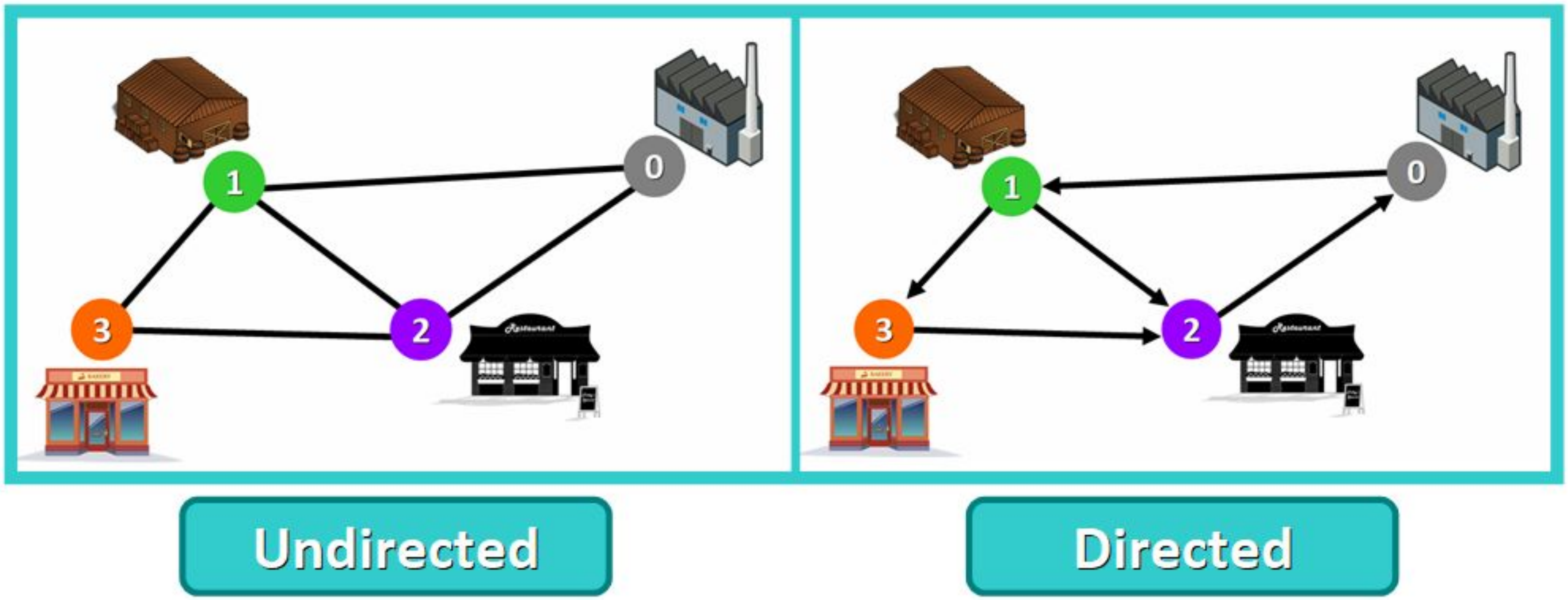
Unit-4:Greedy Algorithms

- **Graphs:**
- A graph is essentially an interrelationship of nodes/vertices connected by edges.
- Generally, graphs are suited to real-world applications, such as graphs can be used to illustrate a transportation system/network, where nodes represent facilities that transfer or obtain products and edges show routes or subways that connect nodes.



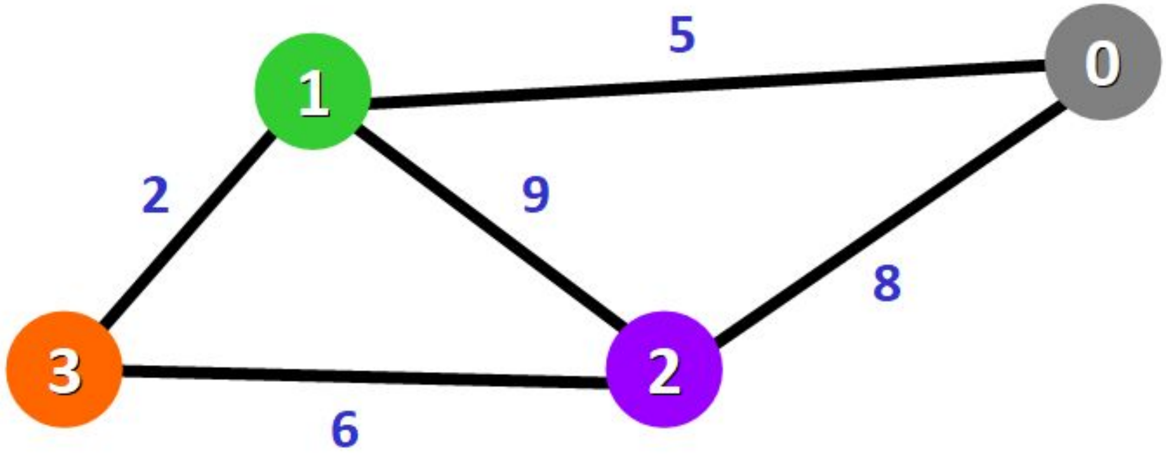
Unit-4:Greedy Algorithms

- **Graphs can be divided into two parts:**
  - **Undirected**: if for every pair of connected nodes, you can go from one node to the other in both directions.
  - **Directed**: if for every pair of connected nodes, you can only go from one node to another in a specific direction. We use arrows instead of simple lines to represent directed edges.



Unit-4:Greedy Algorithms

- **Graphs:**
- **Weighted Graphs**
  - The weight graphs are the graphs where edges of the graph have “a weight” or “cost” and also where weight could reflect distance, time, money or anything that displays the “association” of a couple of nodes it links. These weights are an essential element under Dijkstra's Algorithm.

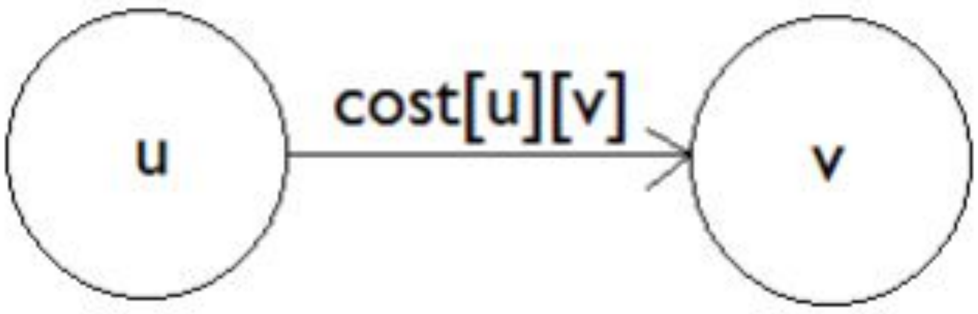


Unit-4:Greedy Algorithms

- **Dijkastra Shortest Path Algorithms:**
- This is an approach of getting single source shortest paths.
- In this algorithm it is assumed that there is no negative weight edge. Dijkstra’s algorithm works using greedy approach, as below:

Unit-4:Greedy Algorithms

- **Dijkastra Shortest Path Algorithms:**
- Let's say, the distance of each node from the source is kept in **d[ ]** array. As in, **d[3]** represents that **d[3]** time is taken to reach node 3 from source.
- If we don't know the distance, we will store infinity in d[3].
- Also, let **cost[u][v]** represent the cost of **u-v**. That means it takes **cost[u][v]** to go from **u** node to **v** node.

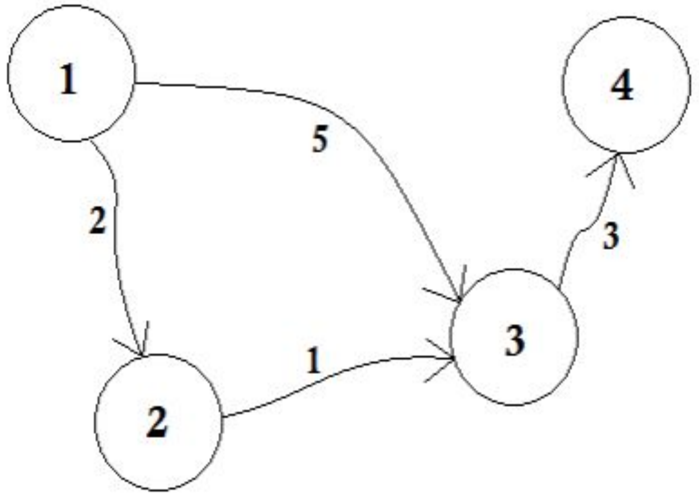


Unit-4:Greedy Algorithms

- **Dijkastra Shortest Path Algorithms:**
- We need to understand Edge Relaxation. Let's say, from your house, that is source, it takes 10 minutes to go to place A. And it takes 25 minutes to go to place B. We have,
  - d[A] = 10
  - d[B] = 25
- Now let's say it takes 7 minutes to go from place A to place B, that means: cost[A][B] = 7
- Then we can go to place **B** from source by going to place A from source and then from place **A**, going to place **B**, which will take 10 + 7 = 17 minutes, instead of 25 minutes. So,
    - **d[B] = d[A] + cost[A][B]**
- This is called relaxation. We will go from node **u** to node **v** and if
- **d[u] + cost[u][v] < d[v]** then we will update **d[v] = d[u] + cost[u][v].**

Unit-4:Greedy Algorithms

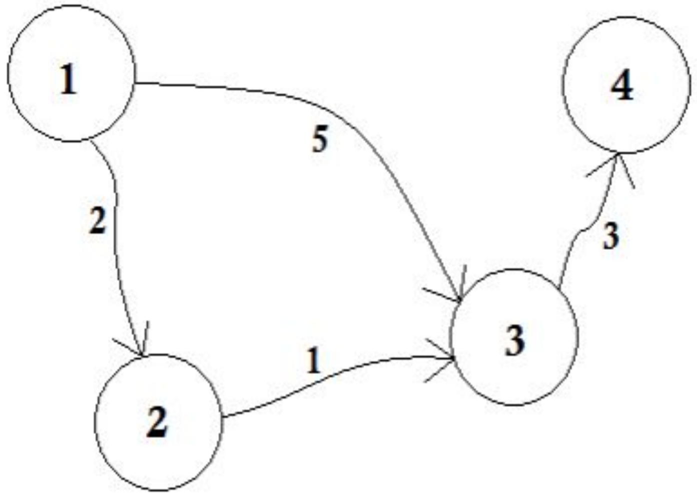
- **Dijkastra Shortest Path Algorithms: Consider a following problem:**
- Let's assume, Node 1 is the Source. Then,
  - d[1] = 0
  - d[2] = d[3] = d[4] = infinity
- We set, d[2], d[3] and d[4] to infinity because
- we don't know the distance yet.
- And the distance of source is of course 0.



- Now, we go to other nodes from source and if we can update them, then we'll push them in the queue.
- Say for example, we'll traverse edge **1-2.** As **d[1] + 2 < d[2]**
  - Which will make **d[2] = 2.**
- Similarly, we'll traverse edge **1-3** which makes **d[3] = 5**.
- We can clearly see that 5 is not the shortest distance we can cross to go to node 3.
- Then we go from node 2 to node 3 using edge **2-3**, we can update **d[3] = d[2] + 1 = 3**.
- Then we go from node 3 to node using edge **3-4**, we can update **d[4] = d[3] + 3 = 6**.

Unit-4:Greedy Algorithms

- **Dijkastra Shortest Path Algorithms: Consider a following problem:**
- So we can see that one node can be updated many tir
- How many times you ask?
- The maximum number of times a node can be updated
- is the number of in-degree of a node.
- Thus we get all the shortest path vertex as:
  - Weight form 1- 2= 2
  - Weight form 1- 3= 3
  - Weight form 1- 4= 6


$$\begin{tikzcd} 1 \arrow[d, "2"] \arrow[dr, bend left, "5"] & & 4 \\ 2 \arrow[r, "1"] & 3 \arrow[ur, "3"] & \end{tikzcd}$$

Unit-4:Greedy Algorithms

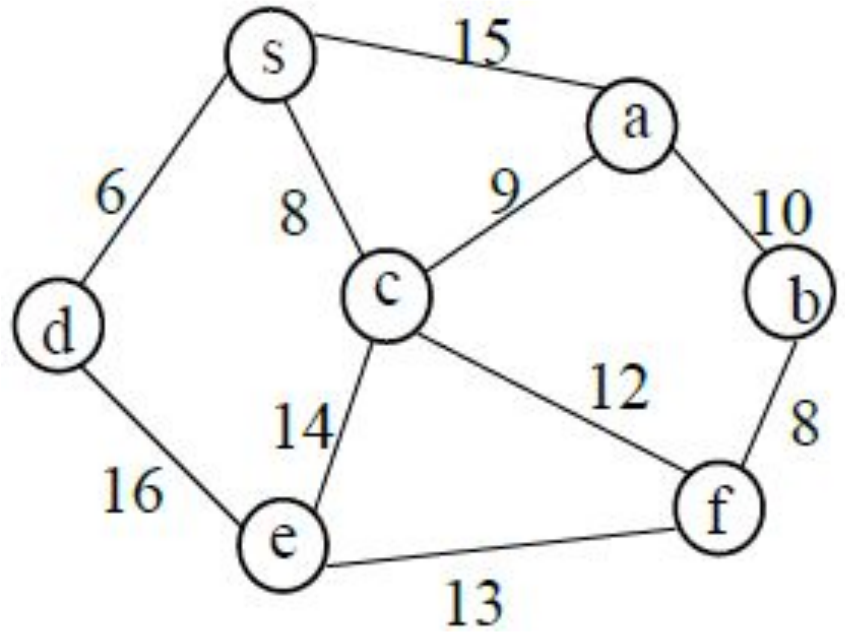
- **Dijkastra Shortest Path Algorithms: Pseudo code**

```
Dijkstra(G,w,s){
    for each vertex vÎ V
        do d[v] = ∞
    p[v] = Nil
    d[s] = 0
    S = Φ
    Q = V
    While(Q!= Φ){
        u = Take minimum from Q and delete.
        S = S □ {u}
```

- **Analysis**:
- In the above algorithm, the first for loop block takes O(V) time.
- Initialization of priority queue Q takes O(V) time.
- The while loop executes for O(V), where for each execution the block inside the loop takes O(V) times .
- Hence the total running time is
- $O(V^2)$.

Unit-4:Greedy Algorithms

- **Dijkastra Shortest Path Algorithms: Example**
- Find the shortest path from the node **S** to other nodes in the following graph.


s
a
b
c
d
e
f
15
6
8
9
10
12
14
16
8
13