Program : **B.Tech**

Subject Name: **Analysis and Design of Algorithm**

Subject Code: **IT-403**

Semester: **4th**

Unit-2: Greedy Strategy:

Introduction:

The greedy method is perhaps the most straight forward design technique, used to determine a feasible solution that may or may not be optimal.

The method:

- Applicable to optimization problems ONLY
- Constructs a solution through a sequence of steps
- Each step expands a partially constructed solution so far, until a complete solution to the problem is reached.

On each step, 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

Feasible solution:- Most problems have n inputs and its solution contains a subset of inputs that satisfies a given constraint(condition). Any subset that satisfies the constraint is called feasible solution.

Optimal solution: To find a feasible solution that either maximizes or minimizes a given objective function. A feasible solution that does this is called optimal solution.

The greedy method suggests that an algorithm works in stages, considering one input at a time. At each stage, a decision is made regarding whether a particular input is in an optimal solution.

Greedy algorithms neither postpone nor revise the decisions (ie., nobacktracking).
Example: Kruskal's minimal spanning tree. Select an edge from a sorted list, check, decide, and never visit it again.

Examples of Greedy Method:

1) Job sequencing with deadline
2) 0/1 knapsack problem
3) Minimum cost spanning trees
4) Single source shortest path problem.

```
Algorithm for Greedy method
Algorithm Greedy(a,n)
//a[1:n] contains the n inputs.
{
Solution :=0;
For i=1 to n do
{
X:=select(a);
If Feasible(solution, x) then
Solution :=Union(solution,x);
}
Return solution;
}
```

- Selection Function, that selects an input from a[] and removes it. The selected input's value is assigned to x.
- Feasible Boolean-valued function that determines whether x can be included into the solution vector.
- Union function that combines x with solution and updates the objective function.

Optimal merge patterns

Merge a set of sorted files of different length into a single sorted file. We need to find an optimal solution, where the resultant file will be generated in minimum time.
If the number of sorted files are given, there are many ways to merge them into a single sorted file. This merge can be performed pair wise. Hence, this type of merging is called as **2-way merge patterns**.

As, different pairings require different amounts of time, in this strategy we want to determine an optimal way of merging many files together. At each step, two shortest sequences are merged.

Example
Let us consider the given files, f_1, f_2, f_3, f_4 and f_5 with 20, 30, 10, 5 and 30 number of elements respectively.

If merge operations are performed according to the provided sequence, then

M_1 = merge f_1 and f_2 => 20 + 30 = 50

M_2 = merge M_1 and f_3 => 50 + 10 = 60

M_3 = merge M_2 and f_4 => 60 + 5 = 65

M_4 = merge M_3 and f_5 => 65 + 30 = 95

Hence, the total number of operations is

50 + 60 + 65 + 95 = 270

Now, the question arises is there any better solution?

Sorting the numbers according to their size in an ascending order, we get the following sequence – **f_4, f_3, f_1, f_2, f_5**

Hence, merge operations can be performed on this sequence

M_1 = merge f_4 and f_3 => 5 + 10 = 15

M_2 = merge M_1 and f_1 => 15 + 20 = 35

M_3 = merge M_2 and f_2 => 35 + 30 = 65

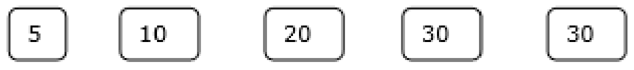
M_4 = merge M_3 and f_5 => 65 + 30 = 95

Therefore, the total number of operations is

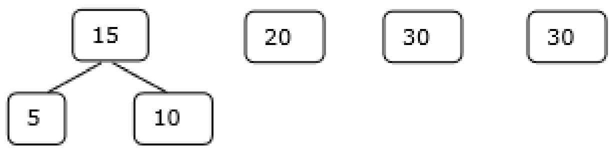
15 + 35 + 65 + 95 = 210

In this context, we are now going to solve the problem using this algorithm.

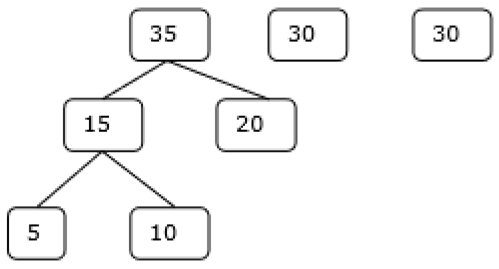
Initial Set



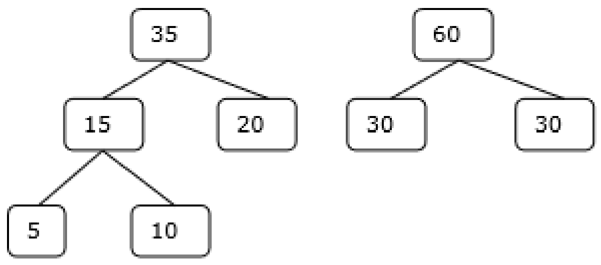
Step-1



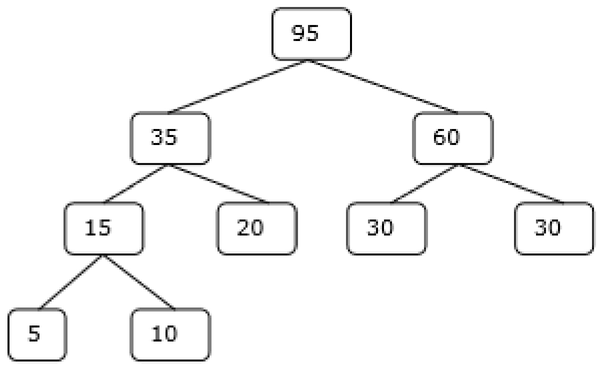
Step-2



Step-3



Step-4



Hence, the solution takes 15 + 35 + 60 + 95 = 205 number of comparisons.

Algorithm: TREE (n)

```
for i := 1 to n – 1 do
   declare new node
   node.leftchild := least (list)
   node.rightchild := least (list)
   node.weight) := ((node.leftchild).weight) + ((node.rightchild).weight)
   insert (list, node);
return least (list);
```

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 prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream.
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 prefix of codes assigned

to a and b. If the compressed bit stream is 0001, the de-compressed output may be "cccd" or "ccb"
There are mainly two major parts in Huffman Coding
1) Build a Huffman Tree from input characters.

2) Traverse the Huffman Tree and assign codes to characters.

Steps to build Huffman Tree:

Input is array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)

2. Extract two nodes with the minimum frequency from the min heap.

3. Create a new internal node with frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.

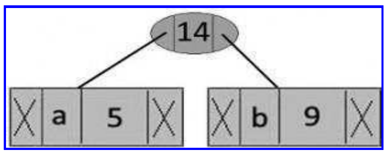
4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

Let us understand the algorithm with an example:

character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

Step 1. Build a min heap that contains 6 nodes where each node represents root of a tree with single node.

Step 2 Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5 + 9 = 14.

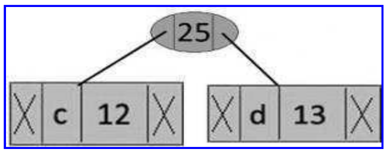


Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements

character	Frequency
c	12
d	13
Internal Node	14
e	16

f	45

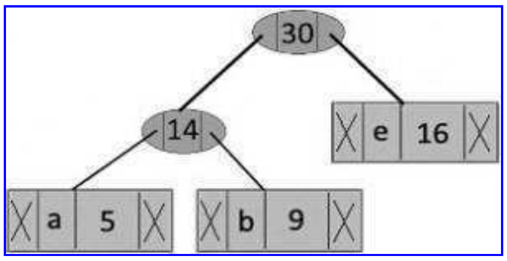
Step 3: Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25



Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes.

character	Frequency
Internal Node	14
e	16
Internal Node	25
f	45

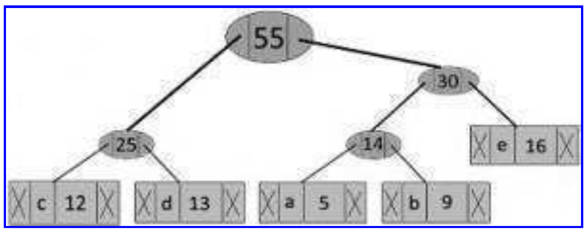
Step 4: Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30



Now min heap contains 3 nodes.

character	Frequency
Internal Node	25
Internal Node	30
f	45

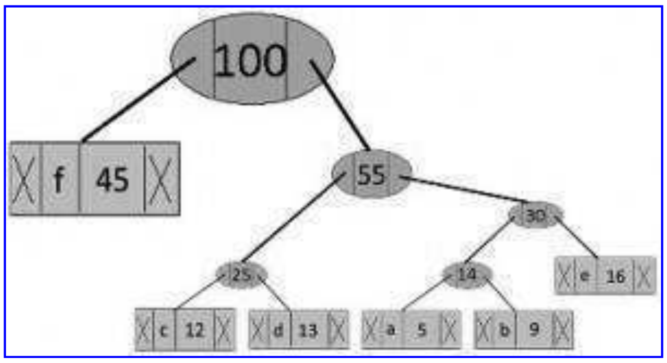
Step 5: Extract two minimum frequency nodes. Add a new internal node with frequency 25 + 30 = 55



Now min heap contains 2 nodes.

character	Frequency
f	45
Internal Node	55

Step 6: Extract two minimum frequency nodes. Add a new internal node with frequency 45 + 55 = 100

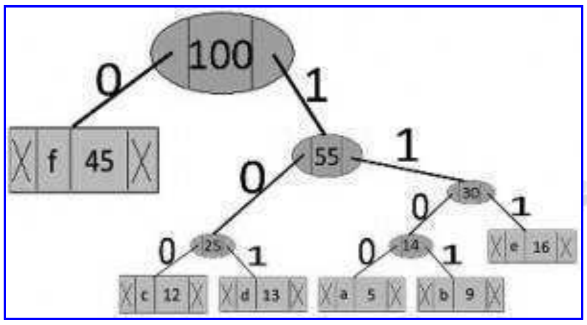


Now min heap contains only one node.

character	Frequency
Internal Node	100

Since the heap contains only one node, the algorithm stops here.

Steps to print codes from Huffman Tree:
Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.



The codes are as follows:

character	code-word
f	0
c	100
d	101
a	1100
b	1101
e	111

<u>Minimum Cost Spanning Tree:</u>

<u>SPANNING TREE</u>: - A Sub graph ‘n’ of o graph ‘G’ is called as a spanning tree if

- ➢ It includes all the vertices of ‘G’
- ➢ It is a tree

<u>Minimum cost spanning tree:</u>
For a given graph ‘G’ there can be more than one spanning tree. If weights are assigned to the edges of ‘G’ then the spanning tree which has the minimum cost of edges is called as minimal spanning tree.

The greedy method suggests that a minimum cost spanning tree can be obtained by contacting the tree edge by edge. The next edge to be included in the tree is the edge that results in a minimum increase in the some of the costs of the edges included so far.

There are two basic algorithms for finding minimum-cost spanning trees, and both are greedy algorithms

Prim’s Algorithm
Kruskal’s Algorithm

Prim’s Algorithm: Start with any *one node* in the spanning tree, and repeatedly add the cheapest edge, and the node it leads to, for which the node is not already in the spanning tree.

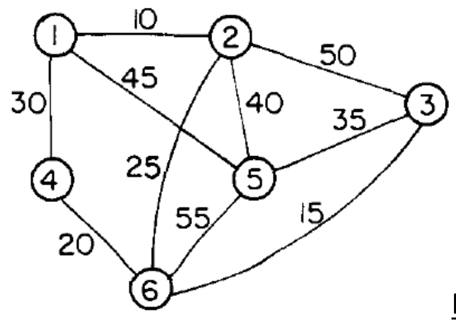
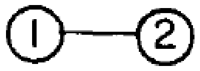
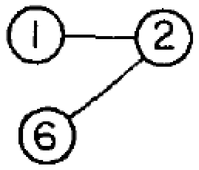
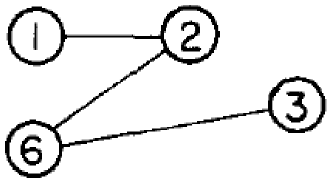
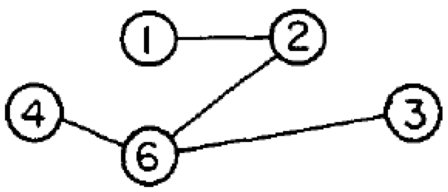
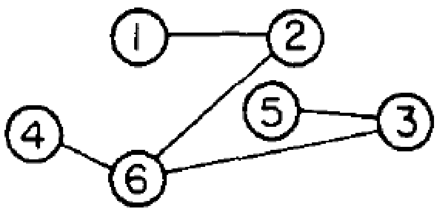


Fig1:Graph

Edge	Cost	Spanning tree
(1,2)	10	
(2,6)	25	
(3,6)	15	
(6,4)	20	
(1,4) (3,5)	reject 35	

Stages in Prim’s Algorithm

Fig2: Stages of Prim's Algorithm

PRIM'S ALGORITHM: -

- ***i)*** Select an edge with minimum cost and include in to the spanning tree.
- ***ii)*** Among all the edges which are adjacent with the selected edge, select the one with minimum cost.
- ***iii)*** Repeat step 2 until 'n' vertices and (n-1) edges are been included. And the sub graph obtained does not contain any cycles.

Notes: - At every state a decision is made about an edge of minimum cost to be included into the spanning tree. From the edges which are adjacent to the last edge included in the spanning tree i.e. at every stage the sub-graph obtained is a tree.

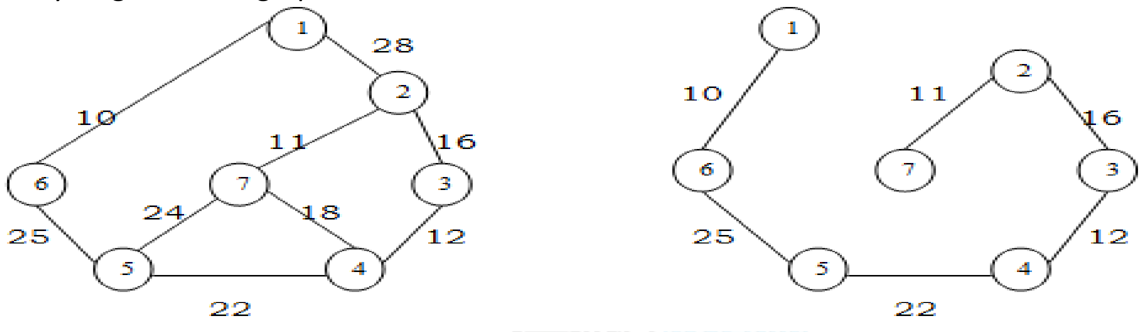


Fig3: Graph

Prim's minimum spanning tree algorithm
Algorithm Prim (E, cost, n,t)
// E is the set of edges in G. Cost (1:n, 1:n) is the
// Cost adjacency matrix of an n vertex graph such that
// Cost (i,j) is either a positive real no. or ∞ if no edge (i,j) exists.
//A minimum spanning tree is computed and
//Stored in the array T(1:n-1, 2).
//(t (i, 1), + t(i,2)) is an edge in the minimum cost spanning tree. The final cost is returned
{
Let (k, l) be an edge with min cost in E
Min cost: = Cost (x,l);
T(1,1):= k; + (1,2):= l;
for i:= 1 to n do//initialize near
if (cost (i,l)<cost (i,k) then n east (i): l;
else near (i): = k;
near (k): = near (l): = 0;
for i: = 2 to n-1 do
{//find n-2 additional edges for t
let j be an index such that near (i) 0 & cost (j, near (i)) is minimum;
t (i,1): = j + (i,2): = near (j);
min cost: = Min cost + cost (j, near (j));

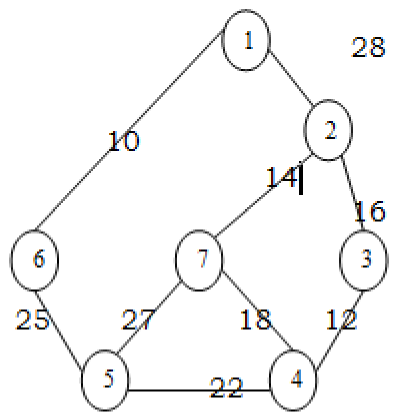
```
near (j): = 0;
for k:=1 to n do // update near ()
if ((near (k) 0) and (cost {k, near (k)) > cost (k,j)))
then near Z(k): = ji
}
return mincost;
}
```

The algorithm takes four arguments E: set of edges, cost is nxn adjacency matrix cost of (i,j)= +ve integer, if an edge exists between i&j otherwise infinity. 'n' is no/: of vertices. 't' is a (n-1):2matrix which consists of the edges of spanning tree.

E = { (1,2), (1,6), (2,3), (3,4), (4,5), (4,7), (5,6), (5,7), (2,7) }

n = {1,2,3,4,5,6,7}

$$\begin{array}{c|ccccccc} \text{Cost} & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 1 & \alpha & 28 & \alpha & \alpha & \alpha & 10 & \alpha \\ 2 & 28 & \alpha & 16 & \alpha & \alpha & \alpha & 14 \\ 3 & \alpha & 10 & \alpha & 12 & \alpha & \alpha & \alpha \\ 4 & \alpha & \alpha & 12 & \alpha & 22 & \alpha & 18 \\ 5 & \alpha & \alpha & \alpha & 22 & \alpha & 25 & 24 \\ 6 & 10 & \alpha & \alpha & \alpha & 25 & \alpha & \alpha \\ 7 & \alpha & 14 & \alpha & 18 & 24 & \alpha & \alpha \end{array}$$



$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 1 & 6 & 5 & 4 & 3 & 2 \\ 2 & 6 & 5 & 4 & 3 & 2 & 7 \end{array}$$

Start Vertex

Ending Vertex

Edges of spanning tree

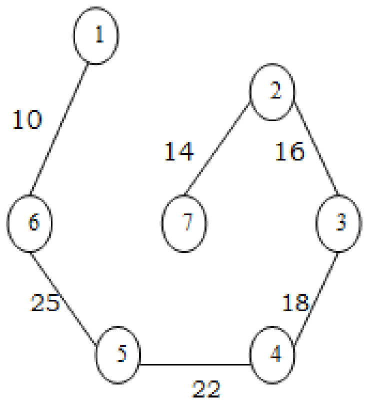


Fig4: Example

1. The algorithm will start with a tree that includes only minimum cost edge of G. Then edges are added to this tree one by one.
2. The next edge (i,j) to be added is such that i is a vertex which is already included in the treed and j is a vertex not yet included in the tree and cost of i,j is minimum among all edges adjacent to 'i'.
3. With each vertex 'j' next yet included in the tree, we assign a value near 'j'. The value near 'j' represents a vertex in the tree such that cost (j, near (j)) is minimum among all choices for near (j)
4. We define near (j):= 0 for all the vertices 'j' that are already in the tree.
5. The next edge to include is defined by the vertex 'j' such that (near (j)) 0 and cost of (j, near (j)) is minimum.

Analysis: -

The time required by the prince algorithm is directly proportional to the no/: of vertices. If a graph 'G' has ‘n’ vertices then the time required by prim’s algorithm is **O(n^2)**

Kruskal's Algorithm: Start with *no* nodes or edges in the spanning tree, and repeatedly add the cheapest edge that does not create a cycle.

In Kruskals algorithm for determining the spanning tree we arrange the edges in the increasing order of cost.

i) All the edges are considered one by one in that order and deleted from the graph and are included in to the spanning tree.

ii) At every stage an edge is included; the sub-graph at a stage need not be a tree. Infect it is a forest.

iii) At the end if we include 'n' vertices and n-1 edges without forming cycles then we get a single connected component without any cycles i.e. a tree with minimum cost.

At every stage, as we include an edge in to the spanning tree, we get disconnected trees represented by various sets. While including an edge in to the spanning tree we need to check it does not form cycle. Inclusion of an edge (i,j) will form a cycle if i,j both are in same set. Otherwise the edge can be included into the spanning tree.

Kruskal minimum spanning tree algorithm

```
Algorithm Kruskal (E, cost, n,t)
//E is the set of edges in G. 'G' has 'n' vertices
//Cost {u,v} is the cost of edge (u,v) t is the set
//of edges in the minimum cost spanning tree
//The final cost is returned
{ construct a heap out of the edge costs using heapify;
        for i:= 1 to n do parent (i):= -1 // place in different sets
//each vertex is in different set                {1} {1} {3}
        i: = 0; min cost: = 0.0;
        While (i<n-1) and (heap not empty))do
{
Delete a minimum cost edge (u,v) from the heaps; and reheapify using adjust;
j:= find (u); k:=find (v);
if (j?k) then
{  i: = 1+1;
   + (i,1)=u; + (i, 2)=v;
    mincost: = mincost+cost(u,v);
    Union (j,k);
    }
}
if (i?n-1) then write ("No spanning tree");
   else return mincost;
}
```

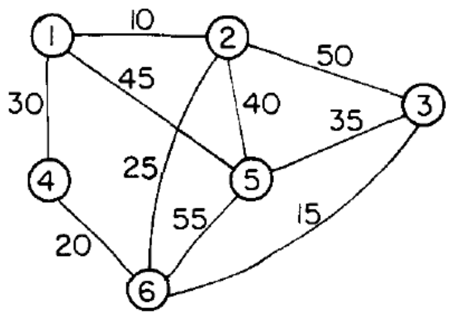
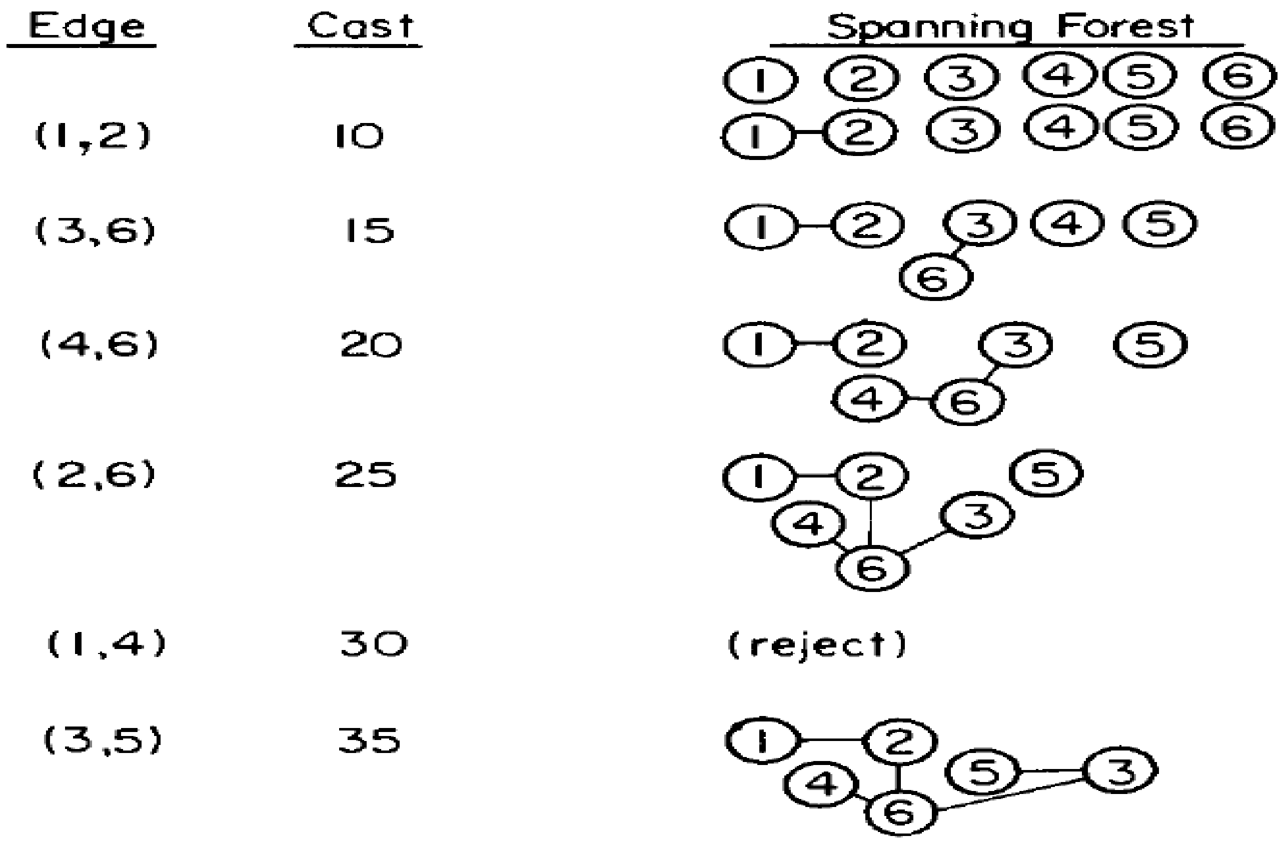


Fig5: Graph

Consider the above graph of , Using Kruskal's method the edges of this graph are considered for inclusion in the minimum cost spanning tree in the order (1, 2), (3, 6), (4, 6), (2, 6), (1, 4), (3, 5), (2, 5), (1, 5), (2, 3), and (5, 6). This corresponds to the cost sequence 10, 15, 20, 25, 30, 35, 40, 45, 50, 55. The first four edges are included in T. The next edge to be considered is (I, 4). This edge connects two vertices already connected in T and so it is rejected. Next, the edge (3, 5) is selected and that completes the spanning tree.



Edge	Cost	Spanning Forest
(1,2)	10	
(3,6)	15	
(4,6)	20	
(2,6)	25	
(1,4)	30	(reject)
(3,5)	35	

Stages in Kruskal's algorithm

Fig6: Stages of Kruskal's Algorithm

Analysis: - If the no/: of edges in the graph is given by /E/ then the time for Kruskals algorithm is given by 0 (|E| log |E|).

Knapsack problem

The **knapsack problem** or **rucksack (bag) problem** is a problem in combinatorial optimization: Given a set of items, each with a mass and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible

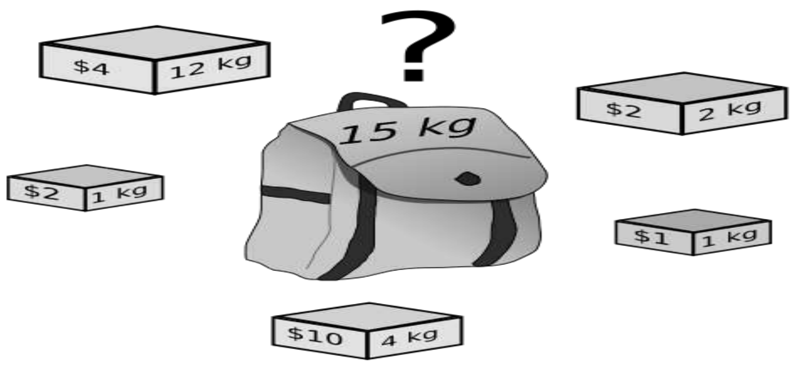


Fig7: Knapsack with items and weights

There are two versions of the problems

- 0/1 knapsack problem
- Fractional Knapsack problem

1. Bounded Knapsack problem.
2. Unbounded Knapsack problem.

Solutions to knapsack problems

- **Brute-force approach**:-Solve the problem with a straight farward algorithm
- **Greedy Algorithm**:- Keep taking most valuable items until maximum weight is reached or taking the largest value of eac item by calculating v_i=$value_i$/$Size_i$
- **Dynamic Programming**:- Solve each sub problem once and store their solutions in an array.

Fractional knapsack problem:

Fractional Knapsack

In this case, items can be broken into smaller pieces, hence the thief can select fractions of items.

According to the problem statement,

- There are **n** items in the store
- Weight of **ith** item $w_i > 0$
- Profit for **ith** item $p_i > 0$
- and
- Capacity of the Knapsack is **W**

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction ***x_i*** of **ith** item. $0 \leqslant x_i \leqslant 1$ The **ith** item contributes the weight $x_i.w_i$ to the total weight in the knapsack and profit $x_i.p_i$ to the total profit. Hence, the objective of this algorithm is to

maximize $\sum_{n=1}^{n}(x_i.p_i)$

subject to constraint, $\sum_{n=1}^{n}(x_i.w_i) \leqslant W$

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit. Thus, an optimal solution can be obtained by $\sum_{n=1}^{n}(x_i.w_i) = W$ In this context, first we need to sort those items according to the value of $p_i w_i$, so that $p_{i+1} w_{i+1} \leq p_i w_i$. Here, ***x*** is an array to store the fraction of items.

Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)

```
for i = 1 to n
  do x[i] = 0
weight = 0
for i = 1 to n
  if weight + w[i] ≤ W then
    x[i] = 1
    weight = weight + w[i]
  else
    x[i] = (W - weight) / w[i]
    weight = W
    break
return x
```

Analysis

If the provided items are already sorted into a decreasing order of **piwi**

, then the whileloop takes a time in ***O(n)***; Therefore, the total time including the sort is in ***O(n logn)***.

Example

Let us consider that the capacity of the knapsack ***W*** **= 60** and the list of provided items are shown in the following table −

Item	**A**	**B**	**C**	**D**
Profit	280	100	120	120
Weight	40	10	20	24
Ratio (*piwi*)	7	10	6	5

As the provided items are not sorted based on **piwi**

. After sorting, the items are as shown in the following table.

Item	**B**	**A**	**C**	**D**
Profit	100	280	120	120
Weight	10	40	20	24
Ratio (*piwi*)	10	7	6	5

Solution

After sorting all the items according to *piwi*

First all of ***B*** is chosen as weight of ***B*** is less than the capacity of the knapsack. Next, item ***A*** is chosen, as the available capacity of the knapsack is greater than the weight of ***A***. Now, ***C*** is chosen as the next item. However, the whole item cannot be chosen as the remaining capacity of the knapsack is less than the weight of ***C***.

Hence, fraction of ***C*** (i.e. (60 − 50)/20) is chosen.

Now, the capacity of the Knapsack is equal to the selected items. Hence, no more item can be selected.

The total weight of the selected items is **10 + 40 + 20 * (10/20) = 60**

And the total profit is **100 + 280 + 120 * (10/20) = 380 + 60 = 440**

This is the optimal solution. We cannot gain more profit selecting any different combination of items.

Ex: - Consider 3 objects whose profits and weights are defined as
$(P_1, P_2, P_3) = (25, 24, 15)$
$W_1, W_2, W_3) = (18, 15, 10)$
n=3?number of objects
m=20?Bag capacity

Consider a knapsack of capacity 20. Determine the optimum strategy for placing the objects in to the knapsack. The problem can be solved by the greedy approach where in the inputs are arranged according to selection process (greedy strategy) and solve the problem in stages. The various greedy strategies for the problem could be as follows.

(x_1, x_2, x_3)	**$\sum x_i w_i$**	**$\sum x_i p_i$**
(1, 2/15, 0)	$18x1+\frac{2}{15}x15 = 20$	$25x1+\frac{2}{15}x\ 24 = 28.2$

(0, 2/3, 1)	$\frac{2}{3}$ x15+10x1= 20	$\frac{2}{3}$ x 24 +15x1 = 31
(0, 1, ½)	1x15+ $\frac{1}{2}$ x10 = 20	1x24+ $\frac{1}{2}$ x15 = 31.5
(½, ⅓, ¼)	½ x 18+⅓ x15+ ¼ x10 = 16. 5	½ x 25+⅓ x24+ ¼ x15 = 12.5+8+3.75 = 24.25

Analysis: - If we do not consider the time considered for sorting the inputs then all of the three greedy strategies complexity will be O(n).

Job Sequence with Deadline:

In job sequencing problem, the objective is to find a sequence of jobs, which is completed within their deadlines and gives maximum profit.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 ***Ji*** is ***di*** and the profit received from this job is ***pi***. Hence, the optimal solution of this algorithm is a feasible solution with maximum profit.

Method:

There is set of n-jobs. For any job i, is a integer deadling di≥0 and profit Pi>0, the profit Pi is earned iff the job completed by its deadline.

To complete a job one had to process the job on a machine for one unit of time. Only one machine is available for processing jobs.

A feasible solution for this problem is a subset J of jobs such that each job in this subset can be completed by its deadline.

The value of a feasible solution J is the sum of the profits of the jobs in J, i.e., $\sum_{i \in J} P_i$

An optimal solution is a feasible solution with maximum value.

The problem involves identification of a subset of jobs which can be completed by its deadline. Therefore the problem suites the subset methodology and can be solved by the greedy method.

Example

Let us consider a set of given jobs as shown in the following table. We have to find a sequence of jobs, 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

Solution

To solve this problem, the given jobs are sorted according to their profit in a descending order. Hence, after sorting, the jobs are ordered as shown in the following table.

Job	**J2**	**J1**	**J4**	**J3**	**J5**
Deadline	1	2	2	3	1
Profit	100	60	40	20	20

From this set of jobs, first we select ***J2***, as it can be completed within its deadline and contributes maximum profit.

- Next, ***J1*** is selected as it gives more profit compared to ***J4***.
- In the next clock, ***J4*** cannot be selected as its deadline is over, hence ***J3*** is selected as it executes within its deadline.
- The job ***J5*** is discarded as it cannot be executed within its deadline.

Thus, the solution is the sequence of jobs (***J2, J1, J4***), which are being executed within their deadline and gives maximum profit.

Total profit of this sequence is **100 + 60 + 20 = 180**.

```
algorithm js(d, j, n)
//d: dead line, j:subset of jobs ,n: total number of jobs
// d[i]≥1 1 ≤ i ≤ n are the dead lines,
// the jobs are ordered such that p[1]≥p[2]≥---≥p[n]
//j[i] is the ith job in the optimal solution 1 ≤ i ≤ k, k subset range
{
d[0]=j[0]=0;
j[1]=1;
k=1;
for i=2 to n do{
r=k;
while((d[j[r]]>d[i]) and [d[j[r]]≠r)) do
r=r-1;
if((d[j[r]]≤d[i]) and (d[i]> r)) then
{
for q:=k to (r+1) setp-1 do j[q+1]= j[q];
j[r+1]=i;
k=k+1;
}
}
return k;
}
```

Single Source Shortest Path Algorithm:

- Graphs can be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway.
- The edges have assigned weights which may be either the distance between the 2 cities connected by the edge or the average time to drive along that section of highway.
- For example if A motorist wishing to drive from city A to B then we must answer the following questions
  - Is there a path from A to B
  - If there is more than one path from A to B which is the shortest path
- The length of a path is defined to be the sum of the weights of the edges on that path.

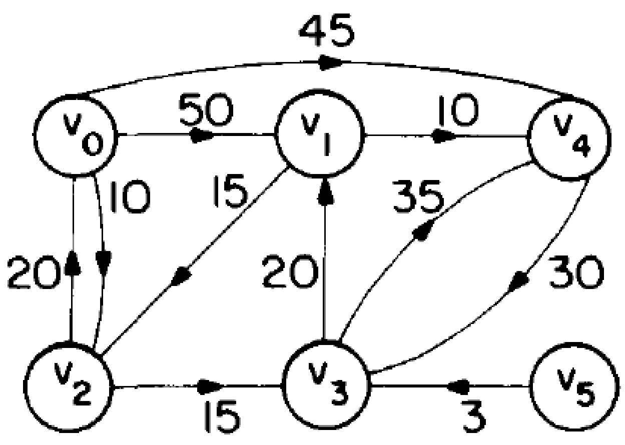
Given a directed graph G(V,E) with weight edge w(u,v). e have to find a shortest path from source vertex S∈v to every other vertex v1∈ v-s.

- To find SSSP for directed graphs G(V,E) there are two different algorithms.

- Bellman-Ford Algorithm
- Dijkstra's algorithm

1. Bellman-Ford Algorithm:- allow –ve weight edges in input graph. This algorithm either finds a shortest path form source vertex **S∈V** to other vertex **v∈V** or detect a –ve weight cycles in G, hence no solution. If there is no negative weight cycles are reachable form source vertex **S∈V** to every other vertex **v∈V**

2. Dijkstra's algorithm:- allows only +ve weight edges in the input graph and finds a shortest path from source vertex **S∈V** to every other vertex **v∈V**.



	Path	*Length*
1)	$v_0\ v_2$	10
2)	$v_0\ v_2\ v_3$	25
3)	$v_0\ v_2\ v_3\ v_1$	45
4)	$v_0\ v_4$	45

Graph and shortest paths from v_0 to all destinations

- Consider the above directed graph, if node 1 is the source vertex, then shortest path from 1 to 2 is 1,4,5,2. The length is 10+15+20=45.
- To formulate a greedy based algorithm to generate the shortest paths, we must conceive of a multistage solution to the problem and also of an optimization measure.
- This is possible by building the shortest paths one by one.
- As an optimization measure we can use the sum of the lengths of all paths so far generated.
- If we have already constructed 'i' shortest paths, then using this optimization measure, the next path to be constructed should be the next shortest minimum length path.
- The greedy way to generate the shortest paths from Vo to the remaining vertices is to generate these paths in non-decreasing order of path length.
- For this 1st, a shortest path of the nearest vertex is generated. Then a shortest path to the 2nd nearest vertex is generated and so on.

Algorithm for finding Shortest Path

```
Algorithm ShortestPath(v, cost, dist, n)
//dist[j], 1≤j≤n, is set to the length of the shortest path from vertex v to vertex j in graph g
with n-vertices.
// dist[v] is zero
{
for i=1 to n do{
s[i]=false;
dist[i]=cost[v,i];
}
s[v]=true;
dist[v]:=0.0; // put v in s
for num=2 to n do{
// determine n-1 paths from v
choose u form among those vertices not in s such that dist[u] is minimum.
s[u]=true; // put u in s
for (each w adjacent to u with s[w]=false) do
if(dist[w]>(dist[u]+cost[u, w])) then
dist[w]=dist[u]+cost[u, w];
}
}
```