Unit-2

Greedy Technique

1. Introduction:

Greedy is the most straight forward design technique. Most of the problems have n inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a feasible solution. We need to find a feasible solution that either maximizes or minimizes the objective function. A feasible solution that does this is called an optimal solution.

The greedy method is a simple strategy of progressively building up a solution, one element at a time, by choosing the best possible element at each stage. At each stage, a decision is made regarding whether or not a particular input is in an optimal solution. This is done by considering the inputs in an order determined by some selection procedure. If the inclusion of the next input, into the partially constructed optimal solution will result in an infeasible solution then this input is not added to the partial solution. The selection procedure itself is based on some optimization measure.

1.1 Elements of greedy algorithm

- Greedy choice property
  - A globally optimal solution is derived from a locally optimal (greedy) choice.
  - When choices are considered, the choice that looks best in the current problem is chosen, without considering results from sub problems.
- Optimal substructures
  - A problem has ***optimal substructure*** if an optimal solution to the problem is composed of optimal solutions to sub problems.
  - This property is important for both greedy algorithms and dynamic programming.

1.2 Problems based on Greedy Strategy:-

- Fractional Knapsack
- Optimal Merge Pattern
- Job sequencing with Deadlines
- Minimum Spanning Tree
- Single Source Shortest path

1.3 Advantages & Disadvantages

Disadvantages:

- They do not always work.
- Short term choices may be disastrous on the long term.
- Correctness is hard to prove

Advantages:

- When they work, they work fast
- Simple and easy to implement

Note:

- *Concepts*
  - *Choosing the best possible choice at each step.*
  - *This decision leads to the best overall solution.*
- *Greedy algorithms do not always yield optimal solutions.*

2. General Algorithm of Greedy Strategy:-

2.1 Steps in Design Greedy Algorithm:

- Determine the optimal substructure of the problem.
- Develop a recursive solution.
- Prove that at any stage of the recursion, one of the optimal choices is the greedy choice. Thus, it is always safe to make the greedy choice.
- Show that all but one of the sub problems induced by having made the greedy choice is empty.
- Develop a recursive algorithm that implements the greedy strategy.
- Convert the recursive algorithm to an iterative algorithm.

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.

Algorithm Greedy (a, n)

Let a[ ] be an array of elements that may contribute to a solution. Let S be a solution,

```
Greedy (a [ ], n)
{
    S = empty;
    for each element (i) from a[ ], i = 1:n
    {
        x = Select (a,i);
        if (Feasible(S,x))   S = Union(S,x);
    }
    return S;
}
```

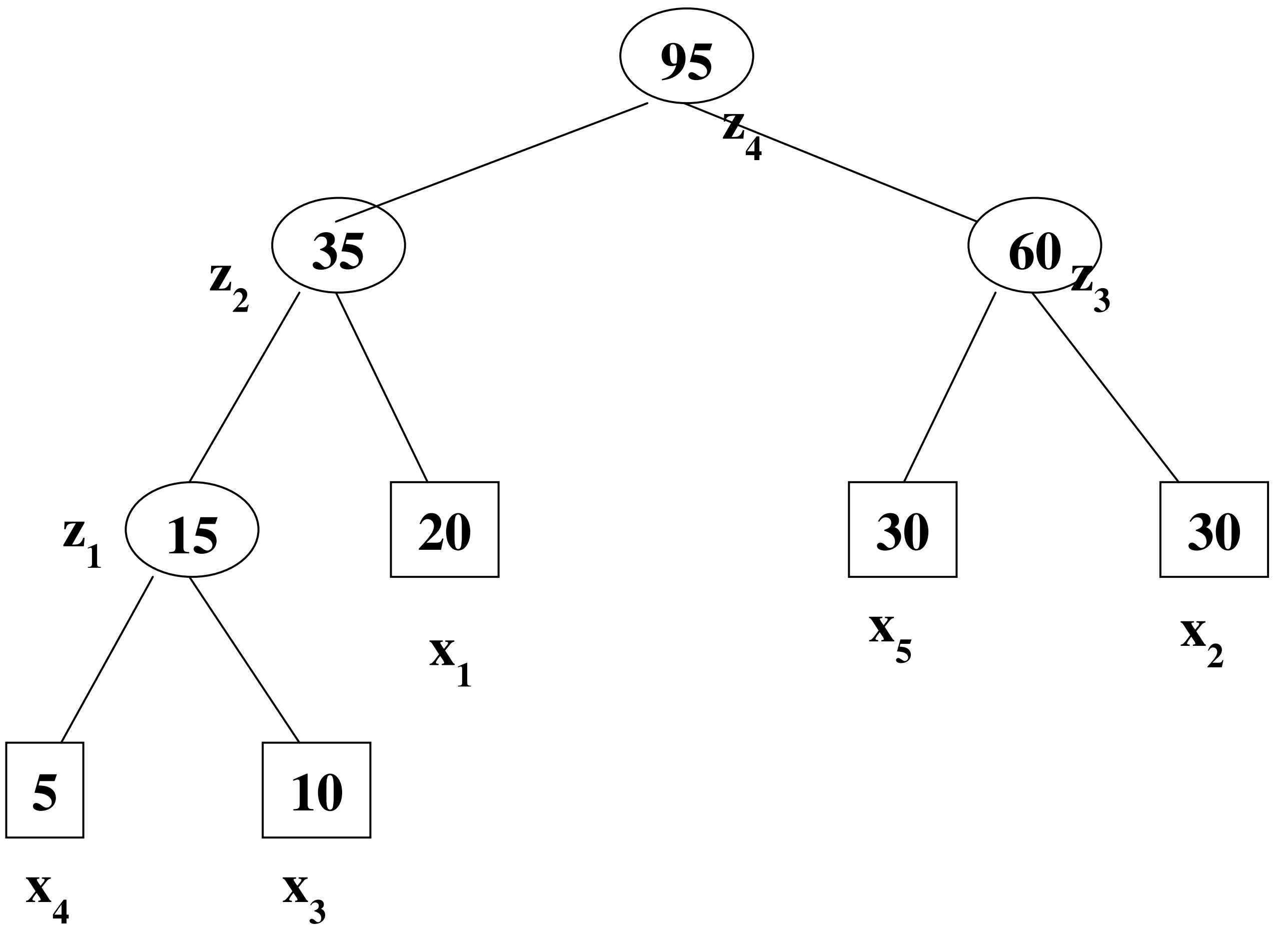
- ***Select:*** *Selects an element from a[ ] and removes it.*
  "Selection is optimized to satisfy an objective function."
- ***Feasible:*** *True if selected value can be included in the solution vector, False otherwise.*
- ***Union:*** *Combines value with solution and updates objective function.*

3. OPTIMAL MERGE PATERNS

Given ‘n’ sorted files, there are many ways to pair wise merging them into a single sorted file. As, different pairings require different amounts of computing time, we want to determine an optimal (i.e., one requiring the fewest comparisons) way to pair wise merge ‘n’ sorted files together.

This type of merging is called as 2-way merge patterns. To merge an n-record file and an m-record file requires possibly n + m record moves, the obvious choice choice is, at each step merge the two smallest files together. The two-way merge patterns can be represented by binary merge trees.

For Example: Five files with lengths (20,30,10,5,30)



Time Complexity:

- If list is kept in nondecreasing order: O(n^2)
- If list is represented as a minheap: O(n log n)

4. **Huffman Codes**

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.

Steps to build Huffman code

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)

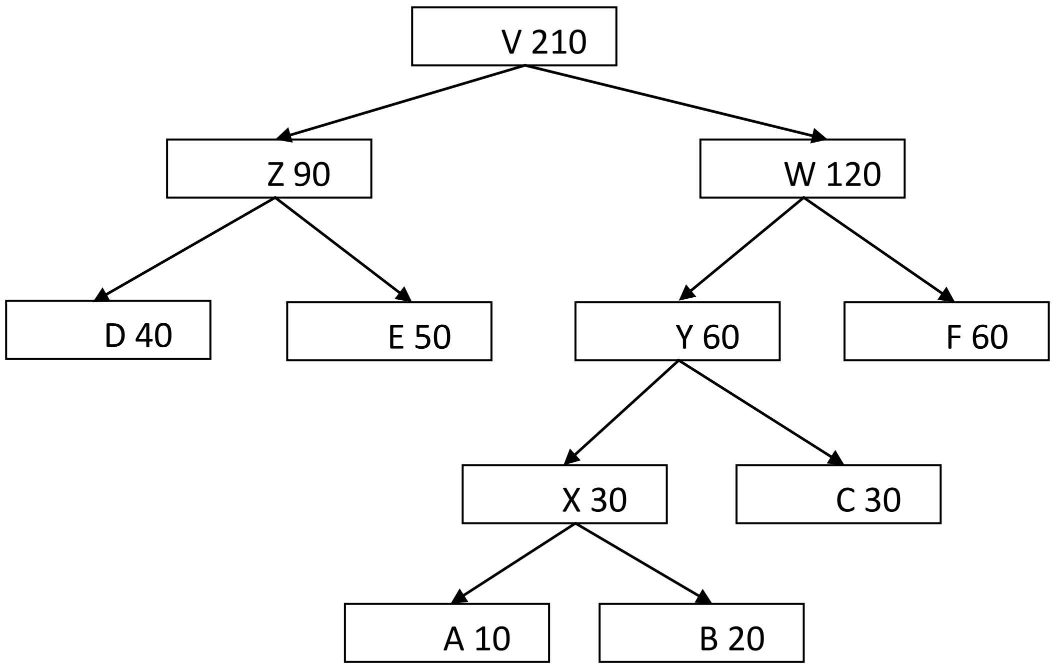
2. Extract two nodes with the minimum frequency from the min heap.

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.

Example:

Letter	**A**	**B**	**C**	**D**	**E**	**F**
Frequency	**10**	**20**	**30**	**40**	**50**	**60**



Example of Huffman code

Output => A:1000, B: 10001, C: 101, D: 00, E: 01, F:11

Analysis of algorithm

Each priority queue operation (e.g. heap): O(log n)

In each iteration: one less subtree.

Initially: n subtrees.

Total: O(n log n) time.

5. Minimum Spanning Tree

A given graph can have many spanning trees. From these many spanning trees, we have to select a cheapest one. This tree is called as minimal cost spanning tree.

Minimal cost spanning tree is a connected undirected graph G in which each edge is labeled with a number (edge labels may signify lengths, weights other than costs). Minimal cost spanning tree is a spanning tree for which the sum of the edge labels is as small as possible

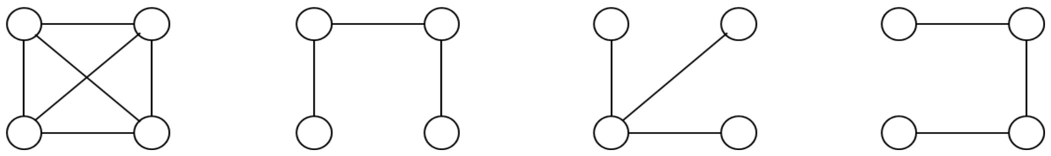
In the spanning tree algorithm, any vertex not in the tree but connected to it by an edge can be added. To find a Minimal cost spanning tree, we must be selective - we must always add a new vertex for which the cost of the new edge is as small as possible.

If each edge of E has a weight, G is called a weighted graph.

Let G=(V,E) be an undirected connected graph.

T is a ***minimum spanning tree*** of *G* if *T* ⊆ *E* is an acyclic subset that connects all of the vertices and whose total weight $w(T) = \sum_{(u,v) \in T} w(u, v)$ is minimized.

T=(V,E') is a spanning tree iff T is a tree.



5.1 How to solve this problem

Using greedy method there are two algorithms:

- **Prim's algorithm.**
- **Kruskal's algorithm.**

General Approach:

- The tree is built edge by edge.
- Let T be the set of edges selected so far.
- Each time a decision is made:
- Include an edge e to T s.t. :
- Cost (T)+w (e) is minimized, and
- T U {e} does not create a cycle.

5.1.1 Kruskal’s Algorithm

This is a greedy algorithm. A greedy algorithm chooses some local optimum (i.e. picking an edge with the least weight in a MST).

Kruskal's algorithm works as follows: Take a graph with 'n' vertices, keep on adding the shortest (least cost) edge, while avoiding the creation of cycles, until (n - 1) edges have been added. Sometimes two or more edges may have the same cost. The order in which the edges are chosen, in this case, does not matter. Different MSTs may result, but they will all have the same total cost, which will always be the minimum cost.

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;
i := 0; mincost := 0.0;
// Each vertex is in a different set.
while ((i < n -1) and (heap not empty)) do
{
Delete a minimum cost edge (u, v) from the heap and re-heapify using Adjust;
j := Find (u); k := Find (v);
if (j < k) then
{
i := i + 1;
t [i, 1] := u; t [i, 2] := v; mincost :=mincost + cost [u, v]; Union (j, k);
}
}
if (i >n-1) then write ("no spanning tree");
else return mincost;
}
```

Running time:

• The number of finds is at most 2e, and the number of unions at most n-1. Including the initialization time for the trees, this part of the algorithm has a complexity that is just slightly more than O (n + e).

• We can add at most n-1 edges to tree T. So, the total time for operations on T is O(n).

Summing up the various components of the computing times, we get O (n + e log e) as asymptotic complexity.

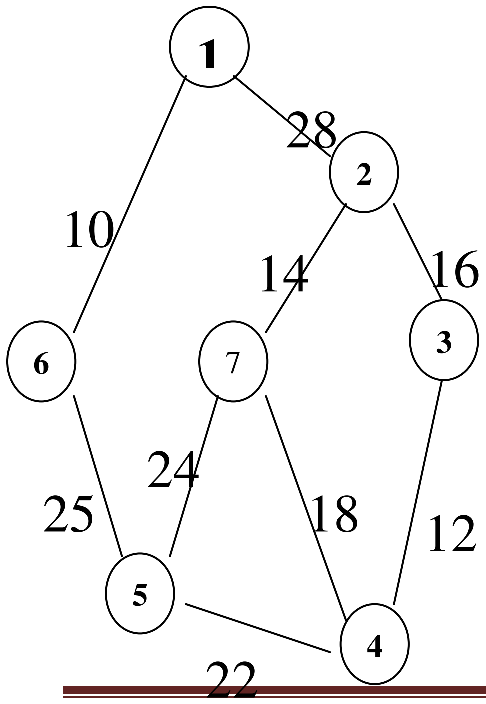
5.1.2 Prim's Algorithm

- Prim's algorithm has the property that the edges in the set *A* always form a single tree.
- The tree starts from an arbitrary root vertex *r.*
- Grow the tree until it spans all the vertices in *V*.
- At each step, a light edge is added to the tree *A* that connects *A* to an isolated vertex of $G_A = (V, A)$.

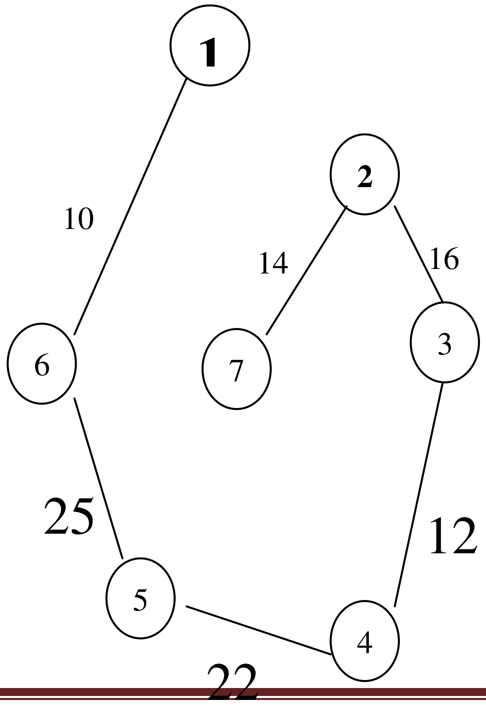
Algorithm Prim (E, cost, n, t)

```
// 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] iseither a positive real number or if no edge
(i, j) exists.A minimum spanning tree is computed and stored as a set of edges in the array t [1:n-1, 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 of minimum cost in E;
mincost := cost [k, l];
t [1, 1] := k; t [1, 2] := l;
for     i :=1 to n do    // Initialize near if (cost [i, l] < cost [i, k]) then near [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 [j] ? 0 and cost [j, near [j]] is minimum;
t [i, 1] := j; t [i, 2] := near [j];
mincost := mincost + 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 [k] := j;
}
return mincost;
}
```

Example: Given Graph

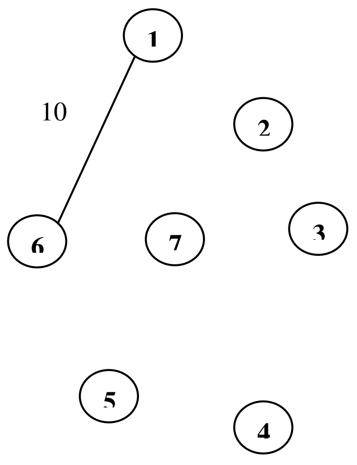


Minimum Spanning Tree

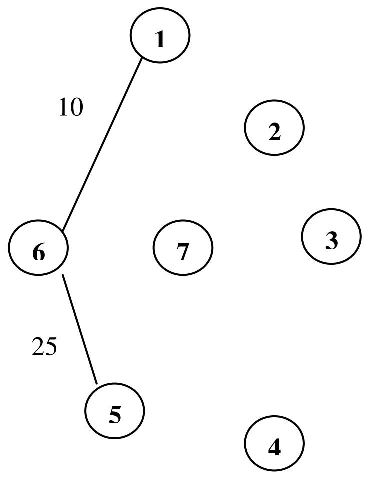


Solution of the above given graph:

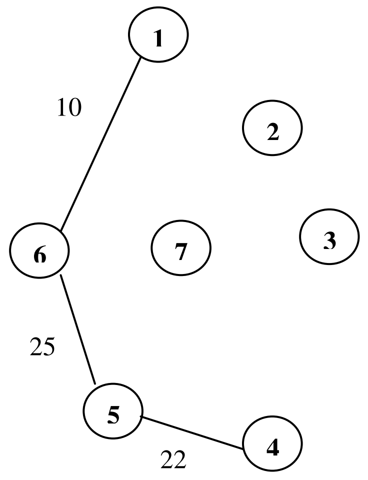
Prims Algorithm :



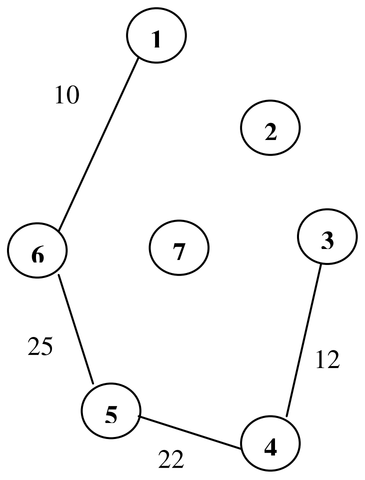
(a)



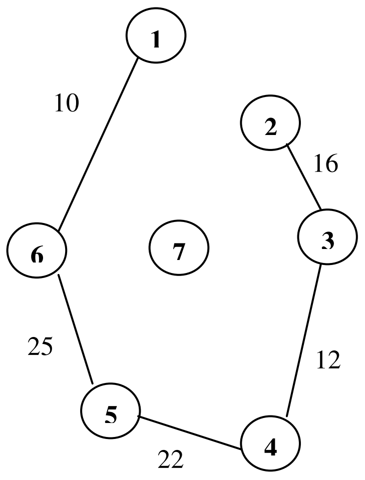
(b)



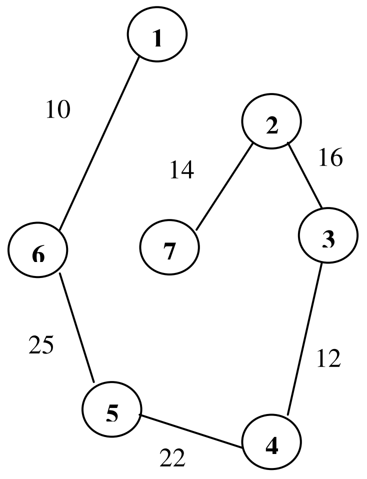
(c)



(d)

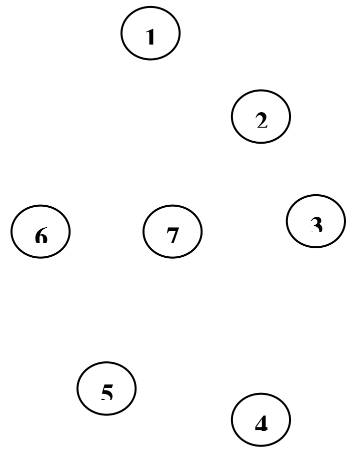


(e)

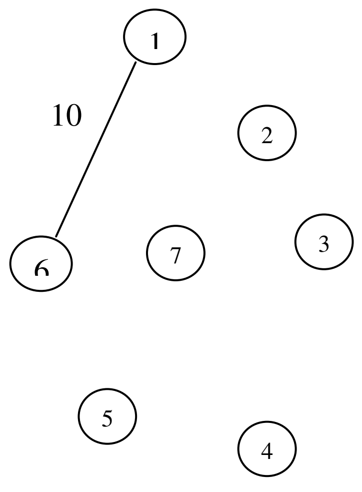


(f)

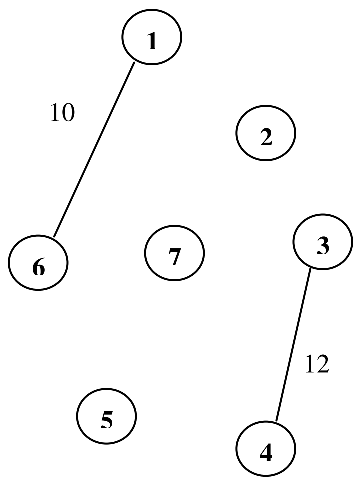
Kruskal Algorithm:



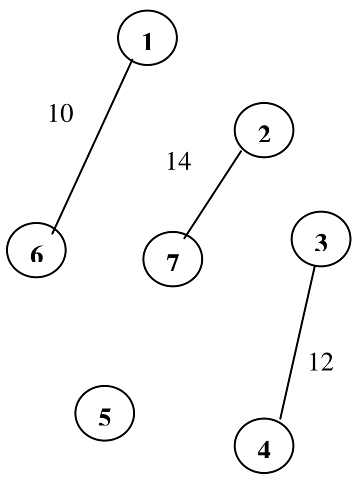
(a)



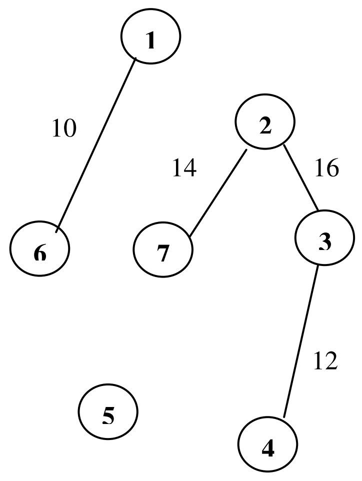
(b)



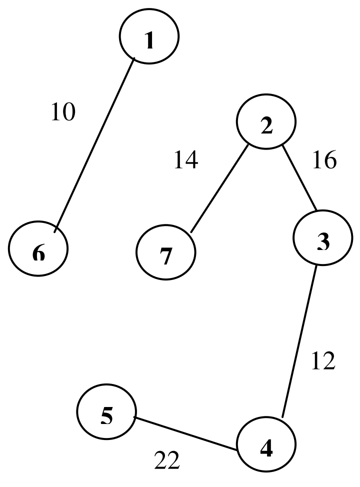
(c)



(d)



(e)



(f)

6. KNAPSACK PROBLEM

In Knapsack problem there are n objects and each object i has a weight wi and a profit pi and Knapsack (Capacity) is M. The objective is to obtain a filling of knapsack to maximizing the total profit.

There are two types of Knapsack problem:

- Fractional Knapsack
- O/1 Knapsack

0/1 Knapsack is based on Dynamic Programming and Fractional knapsack is based on Greedy Strategy, Here we are discussing Fractional Knapsack.

Let us apply the greedy method to solve the knapsack problem. We are given 'n' objects and a knapsack. The object 'i' has a weight w_i and the knapsack has a capacity 'm'. If a fraction xi, 0 < xi < 1 of object i is placed into the knapsack then a profit of pi xi is earned. The objective is to fill the knapsack that maximizes the total profit earned.

Since the knapsack capacity is 'm', we require the total weight of all chosen objects to be at most 'm'.

Problem formulation

$$maximize \sum_{1\le i\le n} p_i x_i$$

$$subject\ to \sum_{1\le i\le n} w_i x_i \le m$$

$$and\ 0 \le x_i \le 1, \quad 1 \le i \le n$$

6.1 Algorithm

If the objects are already been sorted into non-increasing order of p[i] / w[i] then the algorithm given below obtains solutions corresponding to this strategy. Here p[i] / w[i] is our selection function.

```
Greedy Fractional-Knapsack (P[1..n], W[1..n], X [1..n], M)
/* P[1..n] and W[1..n] contains the profit and weight of the n-objects ordered such that X[1..n] is
a solution set and M is the capacity of Knapsack*/
{
For i ← 1 to n do
X[i] ← 0
profit ← 0                    //Total profit of item filled in Knapsack
weight ← 0                    // Total weight of items packed in Knapsack
i←1
While (Weight < M) // M is the Knapsack Capacity
          {
if (weight + W[i] ≤ M)
X[i] = 1
weight = weight + W[i]
else
X[i] = (M-weight)/w[i]
weight = M
Profit = profit + p [i]*X[i]
i++;
}//end of while
}//end of Algorithm
```

Running time:

The objects are to be sorted into non-decreasing order of pi / wi ratio. But if we disregard the time to initially sort the objects, the algorithm requires O(nlogn) time.

Example:

n = 3 objects, m = 20

P = (25, 24, 15), W = (18, 15, 10),

V =P/W

V= (1.39, 1.6, 1.5)

Objects in decreasing order of V are {2 , 3 , 1}

Set X = {0, 0, 0} and Rem = m = 20

K = 1, Choose object i = 2:

w_2 < Rem, Set x_2 = 1, $w_2 x_2$ = 15 , Rem = 5

K = 2, Choose object i = 3:

w_3 > Rem, break;

K < n, x_3 = Rem / w_3 = 0.5

Optimal solution is X = (0 , 1.0 , 0.5) ,

Total profit is $\Sigma_{1\le i\le n} p_i x_i = 31.5$

Total weight is $\Sigma_{1\le i\le n} w_i x_i = m = 20$

7. JOB SEQUENCING WITH DEADLINES

The problem is stated as below.

- There are n jobs to be processed on a machine.
- Each job i has a deadline $d_i \geq 0$ and profit $p_i \geq 0$.
- Pi is earned if the job is completed by its deadline.
- The job is completed if it is processed on a machine for unit time.
- Only one machine is available for processing jobs.
- Only one job is processed at a time on the machine.
- A feasible solution is a subset of jobs J such that each job is completed by its deadline.
- An optimal solution is a feasible solution with maximum profit value.

7.1 How to solve the problem:

Sort the jobs in 'j' ordered by their deadlines. The array d [1 : n] is used to store the deadlines of the order of their p-values. The set of jobs j [1 : k] such that j [r], 1 ≤ r ≤ k are the jobs in 'j' and d (j [1]) ≤ d (j[2]) ≤ . . . ≤ d (j[k]). To test whether J U {i} is feasible, we have just to insert i into J preserving the deadline ordering and then verify that d [J[r]] ≤ r, 1 ≤ r ≤ k+1.

Algorithm GreedyJob (d, J, n)

```
// J is a set of jobs that can be completed by their deadlines.
{
J := {1};
for i := 2 to n do
{
if (all jobs in J U {i} can be completed by their dead lines)
then J := J U {i};
}
}
```

Example : Let n = 4, $(p_1,p_2,p_3,p_4) = (100,10,15,27)$, $(d_1,d_2,d_3,d_4) = (2,1,2,1)$

S.No.	Feasible Solution	processing sequence	value
1.	(1, 2)	2, 1	110
2.	(1, 3)	1, 3 or 3, 1	115
3.	(1, 4)	4, 1	127
4.	(2, 3)	2, 3	25
5.	(3, 4)	4, 3	42
6.	(1)	1	100
7.	(2)	2	10
8.	(3)	3	15
9.	(4)	4	27

We still have to discuss the running time of the algorithm. The initial sorting can be done in time O(n log n), and the rest loop takes time O(n). It is not hard to implement each body of the second loop in time O(n), so the total loop takes time $O(n^2)$. So the total algorithm runs in time $O(n^2)$. Using a more sophisticated data structure one can reduce this running time to O(n log n), but in any case it is a polynomial-time algorithm.

8. The Single Source Shortest-Path Problem: DIJKSTRA'S ALGORITHMS

In the previously studied graphs, the edge labels are called as costs, but here we think them as lengths. In a labeled graph, the length of the path is defined to be the sum of the lengths of its edges.

In the single source, all destinations, shortest path problem, we must find a shortest path from a given source vertex to each of the vertices (called destinations) in the graph to which there is a path.

Dijkstra’s algorithm is similar to prim's algorithm for finding minimal spanning trees.

Dijkstra’s algorithm takes a labeled graph and a pair of vertices P and Q, and finds the shortest path between then (or one of the shortest paths) if there is more than one. The principle of optimality is the basis for Dijkstra’s algorithms.

Dijkstra's algorithm does not work for negative edges at all.

Algorithm Shortest-Paths (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 the digraph G with n vertices.
// dist [v] is set to zero. G is represented by its
// cost adjacency matrix cost [1:n, 1:n].
{
for i :=1 to n do
{
S [i] := false;          // Initialize S. dist [i] :=cost [v, i];
}
S[v] := true; dist[v]  := 0.0;            // Put v in S. for num := 2 to n – 1 do
{
Determine n - 1 paths from v.
Choose u from among those vertices not in S such that dist[u] is minimum; S[u] := true; // Put u is S.
for (each w adjacent to u with S [w] = false)
do
if (dist [w] > (dist [u] + cost [u,  w]) then  // Update distances dist [w] := dist [u] + cost [u, w];
}
}
```

Running time:

For heap A = O (n); B = O (log n); C = O (log n) which gives O (n + m log n) total.