UNIT-III
Greedy Method

What is Greedy Approach?

- Suppose that a problem can be solved by a sequence of decisions. The greedy method has that each decision is locally optimal. These locally optimal solutions will finally add up to a globally optimal solution.
- Only a few optimization problems can be solved by the greedy method.

Control abstraction for Greedy Method

```
Algorithm GreedyMethod (a, n)
{
    // a is an array of n inputs
    Solution: =Ø;
    for i: =o to n do
    {
            s: = select (a);
             if (feasible (Solution, s)) then
             {
                     Solution: = union (Solution, s);
             }
             else
       reject (); // if solution is not feasible reject it.
    }
    return solution;
}
```

Three important activities

1. A selection of solution from the given input domain is performed, i.e. *s:= select(a)*.
2. The feasibility of the solution is performed, by using feasible '*(solution, s)*' and then all feasible solutions are obtained.
3. From the set of feasible solutions, the particular solution that *minimizes* or *maximizes* the given objection function is obtained. Such a solution is called optimal solution.

Differentiate *Greedy* and *Divide-and-Conquer*

GREEDY APPROACH	DIVIDE AND CONQUER
1.Many decisions and sequences areguaranteed and all the overlapping subinstancesare considered.	1.Divide the given problem into many subproblems.Find the individual solutions andcombine them to get the solution for themain problem
2. Follows Bottom-up technique	2. Follows top down technique
3.Split the input at every possible points rather than at a particular point	3.Split the input only at specific points (midpoint), each problem is independent.
4. Sub problems are dependent on the main Problem	4. Sub problems are independent on the main Problem
5. Time taken by this approach is not that much efficient when compared with DAC.	5. Time taken by this approach efficient when compared with GA.
6.Space requirement is less when compared DAC approach.	6.Space requirement is very much high when compared GA approach.

Application - JOB SEQUENCING WITH DEADLINES

Procedure

- In this problem we have n jobs j1, j2, … jn, each has an associated their deadlines d1, d2, … dn and their profits p1, p2, … pn.
- Profit will only be awarded or earned if the job is completed on or before the deadline.
- We assume that each job takes unit time to complete.
- The objective is to earn ***maximum*** profit when only one job can be scheduled or processed at any given time.

Contd …

Example:

index	1	2	3	4	5
JOB	j1	j2	j3	j4	j5
DEADLINE	2	1	3	2	1
PROFIT	60	100	20	40	20

index	1	2	3	4	5
JOB	j2	j1	j4	j3	j5
DEADLINE	1	2	2	3	1
PROFIT	100	60	40	20	20

Contd …

Initially

time slot	1	2	3
status	EMPTY	EMPTY	EMPTY

Optimal Solution

time slot	1	2	3
status	J2	J1	J3

Maximum Profit: 100+60+20 = 180

Algorithm

```
int JS(int d[], int j[], int n)
// d[i]>=1, 1<=i<=n are the deadlines, n>=1. 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.
// Also, at termination d[J[i]]<=d[J[i+1]], 1<=i<k.
{
   d[0] = J[0] = 0; // Initialize.
   J[1] = 1; // Include job 1.
   int k=1;
   for (int i=2; i<=n; i++) {
   //Consider jobs in nonincreasing
   // order of p[i]. Find position for
   // i and check feasibility of insertion.
      int r = k;
      while ((d[J[r]] > d[i]) && (d[J[r]] != r)) r--;
      if ((d[J[r]] <= d[i]) && (d[i] > r)) {
         // Insert i into J[].
         for (int q=k; q>=(r+1); q--) J[q+1] = J[q];
         J[r+1] = i; k++;
      }
   }
   return (k);
}
```

Application - KNAPSACK PROBLEM

- In this problem we have a Knapsack that has a weight limit M
- There are items i1, i2, ..., in each having weight w1, w2, ... wn and some benefit (value or profit) associated with it p1, p2, ..., pn
- Our objective is to maximise the benefit such that the total weight inside the knapsack is at most M, and we are also allowed to take an item in fractional part.

$$\max \sum_{0 \le i < n} p_i x_i$$

$$s.t.$$

$$\sum_{0 \le i < n} w_i x_i \le M$$

$$0 \le x_i <= 1$$

$$p_i \ge 0, w_i \ge 0, 0 \le i < n$$

Example

ITEM	WEIGHT	VALUE
i1	6	6
i2	10	2
i3	3	1
i4	5	8
i5	1	3
i6	3	5

M=16

- Maximum Profit (20)
- Minimum Weight (17)
- Maximum Profit – Weight ratio (22.333336)

Algorithm

```
void GreedyKnapsack(float m, int n)
// p[1:n] and w[1:n] contain the profits and weights
// respectively of the n objects ordered such that
// p[i]/w[i] >= p[i+1]/w[i+1]. m is the knapsack
// size and x[1:n] is the solution vector.
{
   for (int i=1; i<=n; i++) x[i] = 0.0; // Initialize x.
   float U = m;
   for (i=1; i<=n; i++) {
      if (w[i] > U) break;
      x[i] = 1.0;
      U -= w[i];
   }
   if (i <= n) x[i] = U/w[i];
}
```

Application – Minimum Spanning Tree

- A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.

Note 1: Every connected and undirected Graph G has at least one spanning tree.

Note 2: A disconnected graph does not have any spanning tree.

- A complete undirected graph can have maximum n^{n-2} number of spanning trees, where n is the number of nodes.

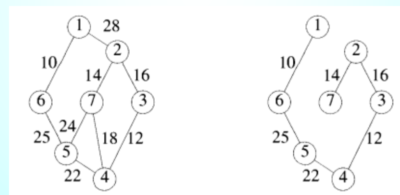
i. Kruskal's Algorithm

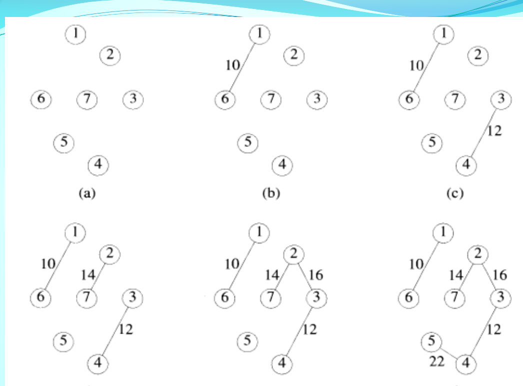
Step 1 - Remove all loops and Parallel Edges.

Step 2 - Arrange all edges in their increasing order of weight.

Step 3 - Add the edge which has the least weightage iff it does not form cycle.

Ex:



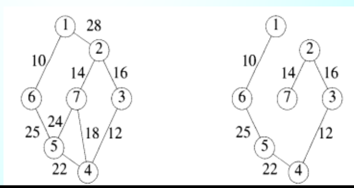


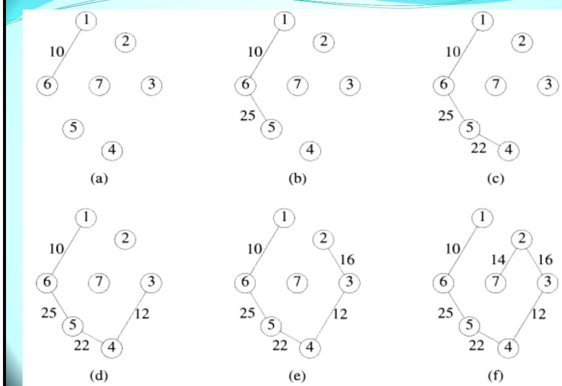
Algorithm

```
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;
    // Each vertex is in a different set.
    i := 0; mincost := 0.0;
    while ((i < n − 1) and (heap not empty)) do
    {
        Delete a minimum cost edge (u, v) from the heap
        and reheapify 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;
}
```

ii. Prim's Algorithm

- Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.
- Step 1 - Remove all loops and parallel edges.
- Step 2 - Choose any arbitrary node as root node.
- Step 3 - Check outgoing edges and select the one with less cost.





Algorithm

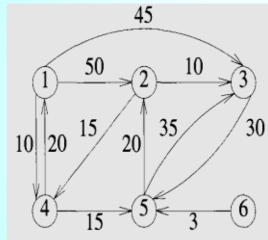
```
Algorithm Prim(E, cost, n, t)
// E is the set of edges in G. cost[1 : n, 1 : n] is the cost
// adjacency matrix of an n vertex graph such that cost[i, j] is
// either a positive real number or ∞ if no edge (i, j) exists.
// A minimum spanning tree is computed and stored as a set of
// edges in the 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;
}
```

Application – Single source shortest path

- For a given source node in the graph, the algorithm finds the shortest path between that node and every other. It also used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined.

Algorithm

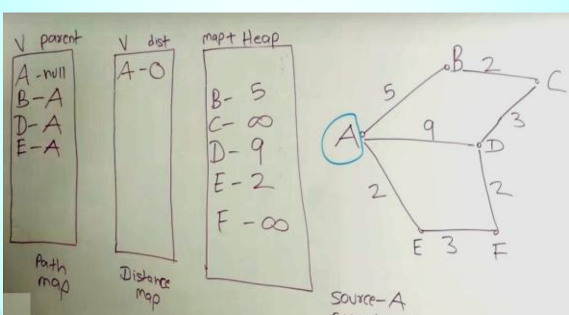
```
Algorithm ShortestPaths(v, cost, dist, n)
// dist[j], 1 ≤ j ≤ n, is set to the length of the shortest
// path from vertex v to vertex j in a digraph G with n
// vertices. dist[v] is set to zero. G is represented by its
// cost adjacency matrix cost[1 : n, 1 : n].
{
    for i := 1 to n do
    { // Initialize S.
        S[i] := false; 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 in S.
        for (each w adjacent to u with S[w] = false) do
            // Update distances.
            if (dist[w] > dist[u] + cost[u, w]) then
                dist[w] := dist[u] + cost[u, w];
    }
}
```



(a) Graph

	Path	*Length*
1)	1, 4	10
2)	1, 4, 5	25
3)	1, 4, 5, 2	45
4)	1, 3	45

(b) Shortest paths from 1



Kruskal's vs Prim's

- Prim's algorithm initializes with a node, whereas Kruskal's algorithm initiates with an edge.
- Prim's algorithms span from one node to another while Kruskal's algorithm select the edges in a way that the position of the edge is not based on the last step.
- In prim's algorithm, graph must be a connected graph while the Kruskal's can function on disconnected graphs too.
- Prim's algorithm has a time complexity of $O(V^2)$, and Kruskal's time complexity is $O(E\log V)$.