UNIT-4

Greedy Method : General Method
-Applications
- Job sequencing with deadlines
- Knapsack problem
- Minimum cost spanning tree
- Single source shortest path problem.

Greedy Method.

General Method :

The Greedy method is a most powerful and straight forward technique in designing an algorithm. A wide variety of algorithms or problems can be solved using this technique.

The drawback of divide and conquer technique, which is rectified in greedy method. Divide and conquer technique is applicable only for problems which can be divisible. There exists some problems which can't be divisible.

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 a given objective function. A feasible solution that does this is called

an optimal solution.

Every feasible solution need not be an optimal solution, but every optimal solution is a feasible solution. Every problem will have a unique optimal solution.

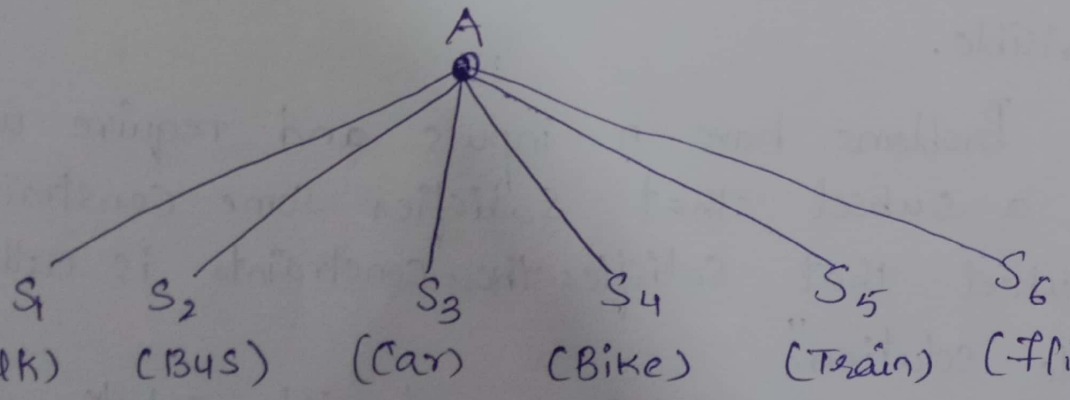
The greedy method operates in stages by considering single input at a time. It makes sequences of choices which is selected that appear best at that moment.

If the inclusion of next input into the partially constructed optimal solution will result in an infeasible solution. Then this input is not added to the partial solution. Otherwise, it is added.

The selection procedure itself is based on some optimization measure. This version of the greedy method is Subset paradigm.

Example:

Suppose there is a problem P and that problem is Person A wants to travel location X to Y. They more solutions. Let us say.



But there is a constraint in the problem, that it covers the journey within 12 hours.

With the constraint, by walk, bus, car, bike is not possible to travel within the time, only possible solutions are by train or by flight.

For any problem there are many solutions. but the solution which are satisfying the conditions for the given problem are called feasible solutions.

If the person covers the journey with the minimum cost (train) is called an optimal solution.

For any problem there can be only one optimal solution.

If the problem requires either minimum or maximum result then we call that type of problem as optimization problem.

Control Abstraction Algorithm for Greedy method :-

```
Algorithm Greedy (a, n)
// a[1:n] contains the inputs.
{
    solution = ϕ; // Initialize the solution.
    for i=1 to n do.
    {
        x = Select(a);
        if feasible(solution, x) then
            Solution = Union(solution, x);
    }
    return solution;
}
```

The function "select" selects an input from a[] removes from it. The selected inputs value is assigned to x.

Feasible is a Boolean valued function that determine whether x can be included in the solution vector.

The function 'Union' combines x with the solution x updates the objective function.

For problems that don't call for the selection of an optimal subset, in the greedy method, we make decision by considering the inputs in some order.

Each decision is made using an optimization criterion that can be computed using decisions already made.

The version of greedy method is called "ordering paradigm".

Greedy Algorithm is defined as a method for solving optimati optimization problems by taking the decisions that result in the most evident and immediate benefit irrespective of the final outcome. It works for cases where minimization or maximization leads to the required solution.

However its important to note that not all problems are suitable for greedy algorithms. They work best when the problem exhibits the following properties:

1. Greedy choice property: The optimal solution can

be constructed by making the best local choice at each step.

2. Optimal Substructure: The optimal solution to the problem contains the optimal solution to its subproblems.

Examples of Greedy algorithms:

- Dijkstra's Algorithm.
- Kruskal's Algorithm
- Fractional knapsack Algorithm
- Scheduling and resource allocation.
- Coin change problem.
- Huffman coding.

Applications of Greedy Method :-

1. Job sequencing with deadlines.
2. 0/1 knapsack problem.
3. Minimum cost spanning tree
4. Single source shortest path problem.

Advantages of Greedy method:

- Simple and easy to understand.
- Efficient for certain problems.
- fast execution time
- Intuitive and easy to explain
- Can be used as building block for more complex algorithms.

Disadvantages :

- Not always optimal.
- Difficult to prove optimality.
- Sensitive to input order.
- Limited applicability.

Greedy Method Time complexity :

The algorithm may be implemented to have the time complexity of $O(n \log n)$.

Job Sequencing with Deadlines :-

The prime objective of the job sequencing with deadlines algorithm is to complete the given order of jobs within respective deadlines, resulting in the highest possible profit. To acheive this, we are given a number of jobs, each associated with a specific deadline, and completing a job before it deadlines earns us a profit. The challenge is to arrange these jobs in a way that maximizes our total profit.

It is not possible to complete all of the assigned jobs within the deadlines. For each job, denoted as j_i, we have a deadline d_i and a profit p_i associated with completing it on time.

Our objective is to find the best solution that maximizes profit while still ensuring that the jobs are completed within their deadlines.

Steps for job sequencing with deadlines :

1. problem setup.
2. Sort the jobs by profit.
3. Initialize the schedule and available time slots.
4. Assign jobs to time slots.
5. Calculate total profit and scheduled jobs.
6. output the results.

Job sequencing with deadlines is often solved by using a Greedy algorithm approach, where jobs are selected based on their profitability and deadline constraints. The goal is to maximize the total profit by scheduling jobs in the most optimal manner.

We are given a set of n jobs, associated with job 'i' is an integer deadline $d_i \geq 0$ and a profit $P_i > 0$. For any job 'i' the profit P_i is earned iff the job is completed by its deadline.

To complete a job, one has to process a job on a machine for one unit of time. only one machine is available for processing jobs.

A feasible solution for the 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, or $\sum_{i \in j} P_i$.

An optimal solution is a feasible solution with maximum value.

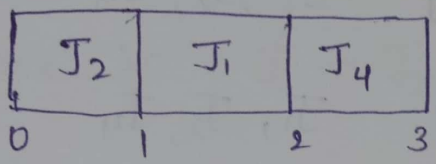
Example: Let $n = 5$

	J_1	J_2	J_3	J_4	J_5
Profits :	20	15	10	5	1
Deadlines :	2	2	1	3	3

Solution :

In the problem maximum deadline is '3'. only 3 jobs are completed within the deadline. We will select the jobs, in the order of their maximum profit. If the profits are not in the descending order, arrange the profits in the descending order.

- J_1 having the highest profit and deadline is 2, arrange J_1 in the 1 to 2 slot.



- J_2 having the next highest profit and deadline is 2. In 1 to 2 time slot, already J_1 is placed. So, J_2 is placed in 0-1 slot.
- Next J_3 deadline is 1. But J_3 not having the time slot and it is not feasible solution. Hence, discard it.
- Now J_4 deadline is 3, arrange J_4 in timeslot 2-3.
- There is not slot available for other jobs.

Job Sequence is $J_1 - J_2 - J_4$

or

$J_2 - J_1 - J_4$

Profit = $20+15+5 = 40$.

Job	Feasible solution (or) slots	Processing sequence	Profit $\sum_{i \in j} P_i$
–	–	ϕ	0
J_1	[1, 2]	J_1	20
J_2	[0, 1] [1, 2]	J_1, J_2	20+15
J_3 (rejected)	[0, 1] [1, 2]	J_1, J_2,	35
J_4	[0, 1] [1, 2] [2, 3]	J_1, J_2, J_4	20+15+5
J_5 (slot not available)	[0, 1] [1, 2] [2, 3]	J_1, J_2, J_4	40.

Example 2 :-

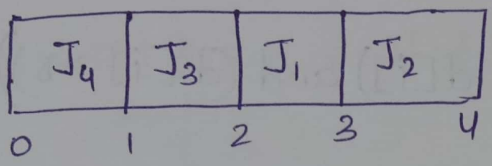
No. of jobs (n) = 7

Jobs :	J_1	J_2	J_3	J_4	J_5	J_6	J_7
Profits :	35	30	25	20	15	12	5
Deadlines :	3	4	4	2	3	1	2

Job	Slot or feasible solution	Processing sequence	Profit
J_1	[2,3]	J_1	35
J_2	[2,3] [3,4]	J_1, J_2	35+30
J_3	[1,2] [2,3] [3,4]	J_1, J_2, J_3	35+30+25
J_4	[0,1] [1,2] [2,3] [3,4]	J_1, J_2 J_3, J_4	35+30+25+20
J_5	[0,1] [1,2] [2,3] [3,4]	J_1, J_2, J_3, J_4	35+30,+25+20

J_6 [0,1] [1,2] [2,3] [3,4] J_1, J_2, J_3, J_4 35+30+25+20

J_4 [0,1] [1,2] [2,3] [3,4] J_1, J_2, J_3, J_4 110



Job sequence : is $J_1 - J_2 - J_3 - J_4$.

Algorithm for Job Sequencing with deadlines :-

```
Algorithm JS (d, j, 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. At termination d[J[i]]≤d[
// J[i+1], 1≤i≤K.
{                  → deadlines array   → Job sequence array
   d[0] := J[0] =0 ; //Initialize.
   J[1]:=1 ;  // include job 1
   K:=1 ; //no.of jobs taken
  for i:= 2 to n do   // 1st job we have taken already.
  {  // Consider jobs in non-decreasing order of P[i].
```

```
// Find position for i and check feasibility of insertion.
   → highest deadline
r := k;
   → check the current job with existing job
while ((d[J[r]] > d[i]) and (d[J[r]] ≠ r)) do
   shift existing Job downwards.
   Whether shifting is required or not
    r := r - 1;
if ((d[J[r]] ≤ d[i]) and (d[i] > r)) then
   Check the feasibility and Job can be inserted or not
{
    // insert i into J[]
    for q = k to (r+1) step -1 do J[q+1] := J[q];
    Time slot
    J[r+1] := i;   → insert the job in sequence
    k := k+1;      → increase the job count.
    }
    return k;
  }
}
```

	J_1	J_2	J_3	J_4	J_5
P	20	15	10	5	1
d	2	2	1	3	3

Algorithm logic :

$d[0] = 0$, $J[0] = 0$

$J[1] = 1$

$K = 1$, $i = 2$

$r = K \Rightarrow r = 1$

$d[J[r]] > d[i]$ and $d(J[1] \neq 1)$

deadline of J_1 — deadline of J_2

$2 > 2$ False

loop will never execute

There is no chance to shift the deadline

if $(d[J[r]] \leq d[i]$ and $(d[i] > r)$

$2 \leq 2$ and $2 > 1$ True

2^{nd} job can be inserted . check whether existing job shifting up or down.

for $q = 1$ to $i+1$ step -1

$q = 1$ to 2

$J[i+1] = 2 \Rightarrow J[2] = 2$ insert the job 2 in sequence

→ $K = 2$, $i = 3$, $r = K \Rightarrow r = 2$

$d[J[2]] > d[3]$ and $d[J[2]] \neq 2]$

$2 > 1$ and $2 \neq 2$ false

loop will never execute

if $(d[J[2]] \leq d[3]$ and $[3] > 2)$ then

$2 \leq 1$ condition False

Insertion not possible . The job is not feasible.

Comes out of the loop.

→ i=4 , r=2, k=2

while (d[J[2]] > d[4] and d[J[2]] ≠ 2)

2>3 false

if (d[J[2]] ≤ d[4] and d[4] >2)

2≤3 and 3>2 True

q=2 to 3 step-1

loop cannot be executed. shifting is not required

J[2+1] =4 Job3 is inserted in the 4th slot.

k=3 , i=5, r=3

while (d[J[3]] >d[4] and d[J[3]] ≠ 3]

4>3 false

if (d[J[3]] ≤d[5] and d[5] ≥3)

3≤3 and 3>3 false

return k=3.

Knapsack problem:-

The knapsack problem is also used for solving real world problems. It is solved by using the greedy approach.

In this problem we can also divide the items means we can take a fractional part of the items that is why it is called the fractional knapsack problem.

In the knapsack problem we are given 'n' objects and a knapsack or bag. object i has a weight w_i and the knapsack has a capacity m.

If a fraction x_i, $0 \leq x_i \leq 1$ of object 'i' is placed into the knapsack, then a profit of $P_i x_i$ is earned. The resulting profit has to be maximum.

We require the total weight of all chosen objects to be at most m. The problem can be stated as,

$$\text{Maximize} \sum_{1 \leq i \leq n} P_i x_i$$

$$\text{Subject to} \sum_{1 \leq i \leq n} w_i x_i \leq m$$

The profits and weights are positive numbers.

Example I :- Consider the following instance in the knapsack problem, $n = 3$, $m = 20$ $(P_1, P_2, P_3) = (25, 24, 15)$ and $(w_1, w_2, w_3) = (18, 15, 10)$.

Solution : Knapsack problem is to find the most valuable subset of items that fit into the knapsack.

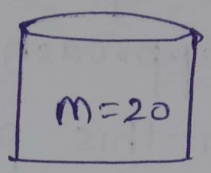
$n = 3$, $m = 20$.

$\frac{P_i}{w_i} = \frac{25}{18}, \frac{24}{15}, \frac{15}{10}$

$= 1.3, 1.6, 1.5$

$x_i = 0 \quad 1 \quad 1/2$

not placed $\quad$ $1/2$ placed



$20 - 15 = 5$

$5 - 5 = 0$

If 18 kg weight is placed into the bag then we will get only 25 profit.

So, Here we are calculating the profit per kg by dividing the objects into fractional parts to place it in a bag.

$\sum x_i w_i = 0 \times 18 + 1 \times 15 + 1/2 \times 10 = 20$

$\sum x_i p_i = 0 \times 25 + 1 \times 24 + 1/2 \times 15 = 31.5$

Optimal solution knapsack vector is $0 \leq x \leq 1$.

In this mel knapsack problem method, the list of items are divisible which means any fraction of the items are considered.

Example 2: Consider the following instance of the knapsack problem, n = 7, m = 15

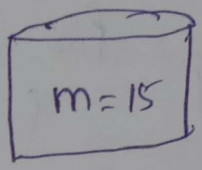
	1	2	3	4	5	6	7
Profit (P_i) =	10	5	15	7	6	18	3
weight (w_i) =	2	3	5	7	1	4	1

Sol:- $\frac{P_i}{w_i} = \frac{10}{2}, \frac{5}{3}, \frac{15}{5}, \frac{7}{7}, \frac{6}{1}, \frac{18}{4}, \frac{3}{1}$

5 1.3 3 1 6 4.5 3

Highest profit object is 5, place the object in bag.

- object 1 is placed, next
- object 6, object 3, object 7.
- object 2 placed
- object 4 is not placed in the bag.



15-1 = 14
14-2 = 12
12-4 = 8
8-5 = 3
3-1 = 2
2-2 = 0

	o1	o2	o3	o4	o5	o6	o7
$\frac{P_i}{w_i}$ =	5	1.3	3	1	6	4.5	3
x_i =	1	2/3 (only 2kg)	1	0	1	1	1

$\sum x_i w_i = 1\times2 + \frac{2}{3}\times3 + 1\times5 + 0\times7 + 1\times1 + 1\times4 + 1\times1$

$= 2 + 2 + 5 + 0 + 1 + 4 + 1 = 15$

$\sum x_i P_i = 1\times10 + \frac{2}{3}\times5 + 1\times15 + 0\times7 + 1\times6 + 1\times18 + 1\times3$

$= 10 + 2\times1.3 + 15 + 6 + 18 + 3$

$= 55.3.$

Algorithm GreedyKnapsack (m, 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 i = 1 to n do x[i] = 0.0 // initialize x.
  U = m;        // maximum weight
                // bag capacity
  for i = 1 to n do
  {
    if (w[i] > U) then break;
    x[i] = 1.0;
    U = U - w[i];
  }
  if (i ≤ n) then x[i] = U/w[i];
}
```

Time complexity : O(N(logN))

Space complexity : O(N)

Minimum Cost Spanning tree :-

A Spanning tree can be defined as a tree like subgraph of a connected undirected graph that includes all the vertices of the graph.

The minimum spanning tree has all the properties of a spanning tree with an added constraint of having the minimum possible weights among all the spanning trees.

Like a spanning tree, there can also be many possible minimum spanning trees for a graph.

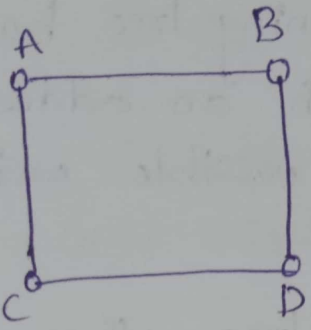
Properties of a spanning tree :

1. The no. of vertices (V) in the graph and the Spanning tree is same.
2. There is fixed no. of edges in the Spanning tree which is equal to one less than the total no. of vertices ($E = V-1$).
3. The spanning tree should not be directed as in there should only be a single source of component not more than that.
4. The spanning tree should be acyclic, which means there would not be any cycle in the tree.

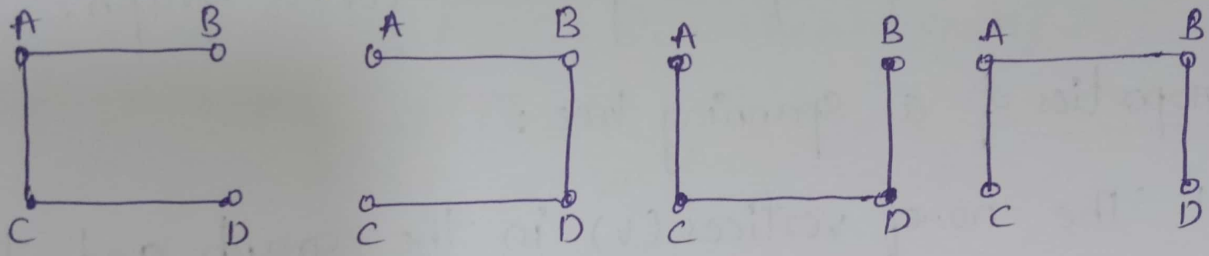
5. The total cost or weight of the spanning tree is defined as the sum of the edges weights of all the edges of the spanning tree.

6. There can be many possible spanning trees for a graph.

Example :



4 Spanning trees for above graph.



Conditions are satisfied.

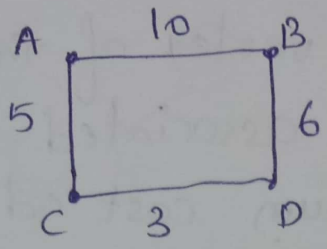
- Spanning tree contains all the vertices (A, B, C, D) of graph.
- Graph contains n vertices then spanning tree should contain (n-1) Edges. [4-1 = 3 edges]
- Spanning tree does not contain any cycle.

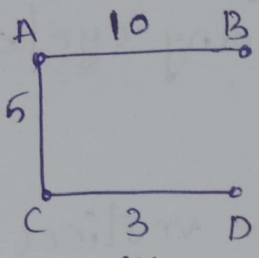
Applications :

- It is used in constructing networks
- Used to find the airline routes.
- Social network analysis.
- Image processing.

A spanning tree in which the cost is minimum than the remaining spanning trees. It is said to be minimum cost spanning tree.

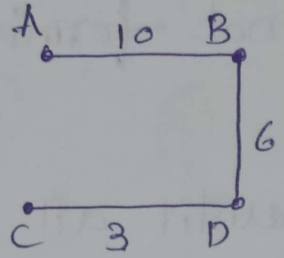
Ex:





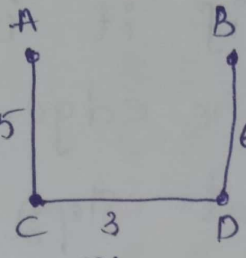
(1)

Cost of the spanning tree = (18)



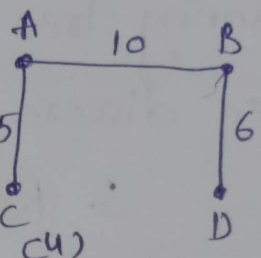
(2)

Cost - 19



(3)

Cost - 14



(4)

Cost - 21

∴ Minimum cost spanning tree is Cost = 14 fig(3)

Minimum cost spanning tree is calculated by two methods:

1. Prim's Algorithm
2. Kruskal's Algorithm.

Prim's Algorithm :-

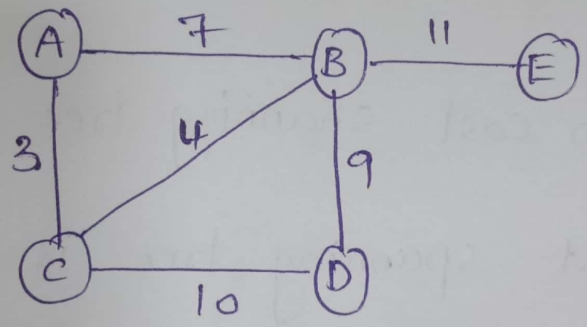
This algorithm mainly used to calculate minimum cost spanning tree.

Steps : 1. start with any vertex of the graph.

2. Find the edges associated with that vertex and add minimum cost edge to the spanning tree, if it is not forming any cycle then discard the edge.

3. Repeat step 2 until all the vertices are covered.

Example :



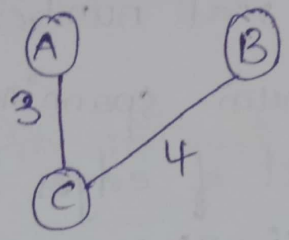
Solution :

Step 1 : Start with the first vertex A. The edge weights of this vertex are compared i.e. 3 is compared with 7. Since 3 is smaller than 7, the edge (A, C) is selected.

A - B - 7
A - C - 3 ✓

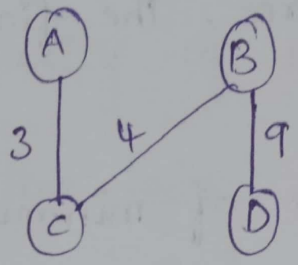
Step 2: Now Consider the vertex 'C'. The edge weights of this vertex are compared and the edge with the smallest weight is selected. ie. edge (C, B).

C-A -3 (Already taken)
C-B -4 ✓
C-D -10

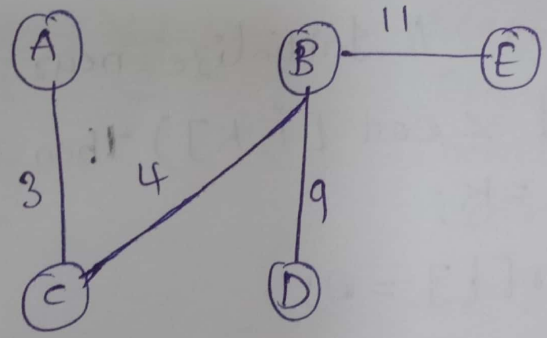


Step 3:

B-C - Already taken
B-D -9 ✓
B-E -11
B-A - 7 (cycle is forming)



Step 4:



Cost of the spanning tree : 3+4+9+11 = 27.

This spanning tree satisfies all the conditions of the Spanning tree.

The resulting graph obtained is a minimum Spanning tree in which the comparisons made by the tree algorithm are 4 in total. for n=5 vertices (n-1) = 5-1 comparisons are made.

↱ set of edge ⟶ tree

Algorithm Prim (E, cost, n, t)

array ⟵ no. of vertices

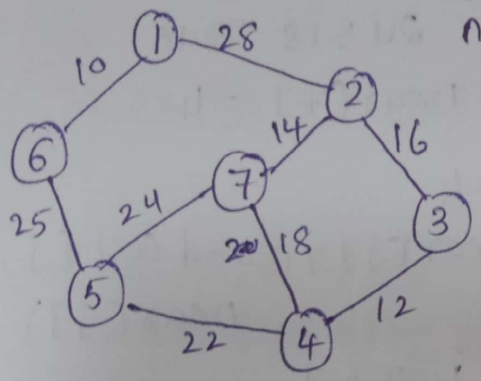
// E is the set of edges in G. cost [1:n, 1:n] is the cost adjacency matrix if an n vertex graph such that cost [i,j] is either a positive real number or '∞' if no edge [i,j] exists. A minimum spanning tree is computed and stored as a set of edges in the array t[1:n-1, 1:2] (t[i,1], t[i,2]) is an edge 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 min cost;
}
```

Time complexity of Prim's Algorithm is $O(n^2)$.

Prim's Algorithm logic:



n = 7

K = 1, l = 6

mincost = cost(1,6) = 10

T[1,1] = 1, T[1,2] = 6

i = 1 to n do

if (Cost[1,6) < cost(1,1)) then near[1] = 6 else n[6] = 1

10 < 8 (True) then near[1] = 6.

i = 2

Cost(2,6) < cost(2,1) then near[2] = 6

∞ < 28 False

else near[2] = 1.

i = 3

Cost(3,6) < cost(3,1) then near[3] = 6

∞ < ∞ False

near[3] = 1 near[6] = 1

near[4] = 1 near[7] = 1

near[5] = 6

near[1] = near[6] = 0

for i = 2 to n-1 (7-1) do

near[j] ≠ 0 and cost[j, near[j]) is minimum

j = 2, 3, 4, 5, 7

Cost(2,1) = 28

Cost(3,1) = ∞

Cost(4,1) = ∞

Cost(5,6) = 25

Cost(7,1) = ∞

minimum cost (5,6) = 25

i=2 , j=5

T[2,1] = 5 T[2,2] = near[5]

near[5] = 0

for K = 1 to n

if (near[K] ≠ 0 and (cost[K, near[K]] > cost(K,j));

K = 2,3,4,7.

Cost(2,1) > Cost(2,5)

28 > ∞ False

Cost(3,1) > Cost(3,5)

∞ > ∞ False

Cost(4,1) > Cost(5,4)

∞ > 22 true

near[4] = 5

Cost[7,1) > cost(7,5)

∞ > 24 True

near[7] = 5

i = 3

near[j] ≠ 0 and cost[j, near[j]);

j = 2,3,4,7.

Cost(2,1) = 28

Cost(3,1) = ∞

Cost(4,5) = 22 ✓ minimum cost

Cost(7,5) = 24

Now j = 4

T[3,1] = 4

T[3,2] = near[4] = 5

mincost = mincost + cost[j, near[j])

= 10 + 25 + 22

near[4] = 0

i = 2,3,7

if (near[2] ≠ 0 and (cost[2, near[2]] > cost[2,4])

28 > ∞

Cost[3, near[3]] > cost[3,4])

Cost(3,1) > cost(3,4)

∞ > 12 True

near[3] = 4

Cost(7, near[7] > cost(7,4))

Cost[7,5] > cost(7,4)

24 > 18 True

near[7] = 4

i = 4

near[j] ≠ 0 and cost[j, near[j])

j = 2,3,7.

Cost(2,1) = 28

Cost(3,4) = 12 ✓ mincost

Cost(7,4) = 18

now j = 3

T(4,1) = 3

T(4,2) = near[3] = 4.

mincost = 10 + 25 + 22 + 12

near[3] = 0.

for k = 1 to n

if (near[k] ≠ 0) and (cost[k, near[k]] > cost[k, j])

k = 2, 7

cost(2, near[2] > cost[2, 3])

cost(2, 1) > cost[2, 3]

28 > 16 True

near[2] = 3

k = 7

cost(7, near[7] > cost(7, 3))

cost(7, 4) > cost(7, 3)

18 > ∞

for i = 5

near[j] ≠ 0 cost[j, near[j]]

j = 2, 7.

cost(2, 3) = 16

cost(7, 4) = 18

mincost = 16

j = 2

T[5, 1] = 2

T[5, 2] = near[2] = 3

mincost = 10 + 25 + 22 + 12 + 16

near[2] = 0

for k = 1 to n

k = 7

cost[7, near[7]] > cost[7, 2])

cost[7, 4] > cost[7, 2]

18 > 14 True

near[7] = 2

i = 6

i = 2 to 6 - 1 = 5 True

near[j] ≠ 0 and cost

(j, near[j])

j = 7

T(6, 1) = 7

T(6, 2) = near[7];

mincost = 10 + 25 + 22 +

12 + 16 + 14

near[7] = 0

for k = 1 to n do

near[k] ≠ 0 and

(cost[k, near[k]]

> cost[k, j])

False

for i = 2 to n-1 False

return

mincost = 10 + 25 + 22 +

12 + 16 + 14

Prim's Algorithm

1. This method starts with a single vertex of a graph similar to tree and gradually adds the smallest edge to make the tree grow by one more vertex.

2. It is suitable when only a single tree, only a few edges are taken into consideration at a time.

3. Considered as more space efficient since only one tree is taken and the edges that connect to the vertices in the tree are examined.

4. It consumes more time

5. The tree that we are making and growing always remains connected.

6. It is faster for dense graphs

Kruskal's Algorithm

1. It starts with each and every vertex of the graph and adds the smallest edge that joins the two trees as the whole.

2. Kruskal's algorithm is suitable when all the edges and vertices are simultaneously taken into consideration.

3. Considered as time efficient because ordering of edges as per their weights is possible for travelling them quickly.

4. It consumes more space.

5. The tree that we making or growing is usually remains disconnected.

6. It is faster for sparse graphs.

kruskal's Algorithm :-

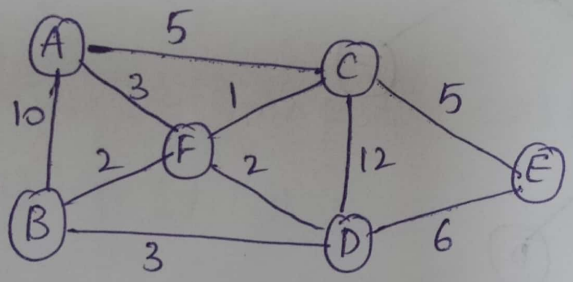
It is used to calculate the minimum cost spanning tree. It chooses an edge in the graph with minimum weight.

Steps : 1. We have to arrange all the nodes in ascending order based on the cost.

2. Select minimum cost edge from the list of sorted edges and add that edge to the spanning tree, if it is not forming any cycle. If any edge form any cycle, then discard the edge.

3. Repeat step 2 till all the edges covered.

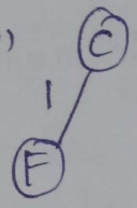
Example:

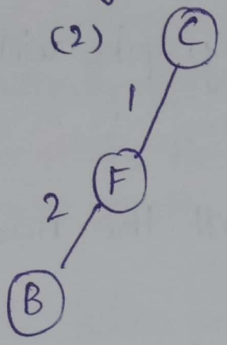


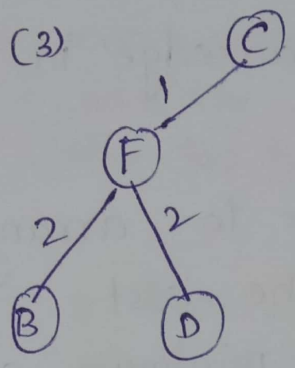
Sol: Minimum cost edges are arranged in ascending order:

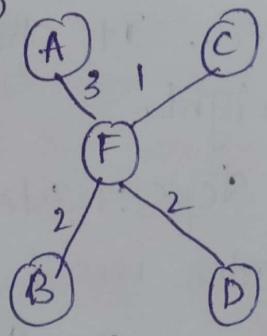
Edge	Cost
C-F	1
B-F	2
D-F	2
A-F	3
B-D	3
A-C	5
C-E	5
D-E	6
A-D	10
C-D	12

Step 2: Select minimum cost edge and add that edge to the spanning tree.

(1) 

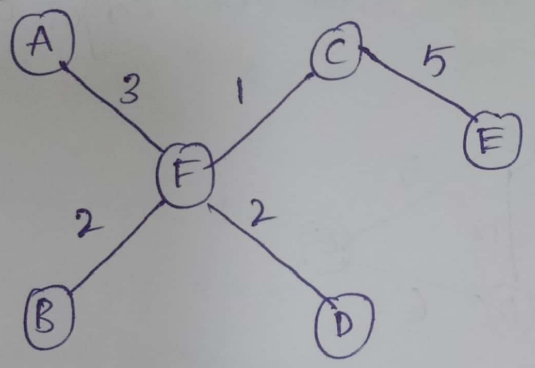
(2) 

(3) 

(4) 

(5) If you add B-D edge it will form cycle. So, discard the edge

(6) A-C also forms cycle, discard it.

(7) 

Cost = 13.

Algorithm :-

- Algorithm Kruskal's (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 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;
}
```

Time Complexity : $O(E \log V)$.

Single Source Shortest path problem :-

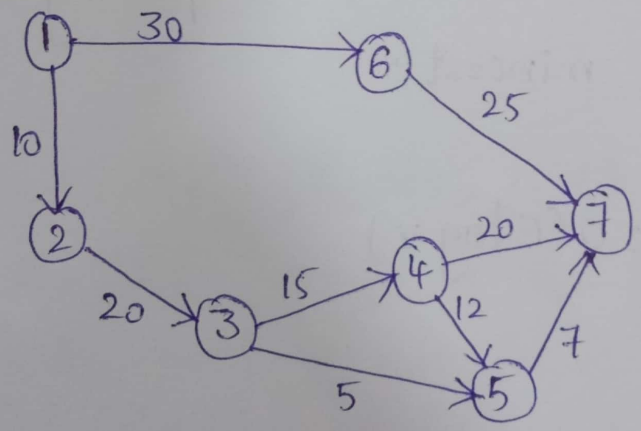
The single source shortest path can be defined as a problem where in shortest path between source vertex and other vertices is identified.

The most important algorithm for solving this problem.

Dijkstra's Algorithm :-

Dijkstra's single source shortest path algorithm finds the shortest path from a given source to all remaining nodes of a digraph.

In each stage of a computation, the nodes of the graph associates themselves with labels. Some labels in this are permanent which represents the shortest path from source node to that node. At the experimental level, the source node is usually set as a permanent label and its value is kept zero.



Let $V = \{1, 2, 3, 4, 5, 6, 7\}$

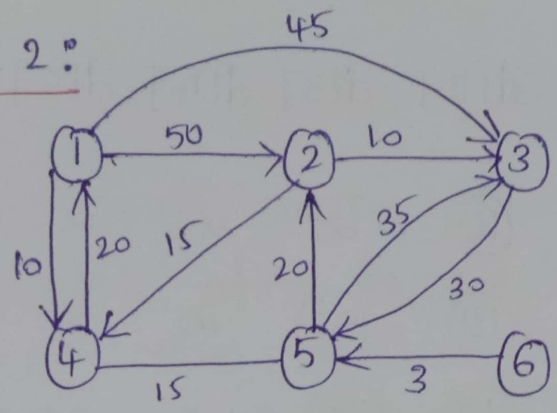
Selected Vertex	Visited set	d[2]	d[3]	d[4]	d[5]	d[6]	d[7]
1	{1}	(10)	∞	∞	∞	30	∞
2	{1,2}	(10)	(30)	∞	∞	30	∞
3	{1,2,3}	(10)	(30)	45	35	(30)	∞
6	{1,2,3,6}	(10)	(30)	45	(35)	(30)	∞
5	{1,2,3,6,5}	(10)	(30)	45	(35)	(30)	(42)
7	{1,2,3,6,5,7}	(10)	(30)	45	(35)	(30)	(42)
4	{1,2,3,6,5,7,4}	(10)	(30)	(45)	(35)	(30)	(42)

Vertex	Cost
2	10
3	30
4	45
5	35
6	30
7	42

Dijkstra's Algorithm :-

- It works only for connected graphs.
- It works only for those graph that do not contain any negative weight edge.
- It only provides the value or cost of the shortest paths.
- It works for directed as well as undirected graph.

Example 2:



$V = \{1, 2, 3, 4, 5, 6\}$

Selected Vertex	Visited set	2	3	4	5	6
1	{1}	50	45	(10)	∞	∞
4	{1,4}	50	45	(10)	(25)	∞
5	{1,4,5}	(45)	45	(10)	(25)	∞
2	{1,4,5,2}	(45)	45	(10)	(25)	∞
3	{1,4,5,2,3}	(45)	45	(10)	(25)	∞
6	{1,4,5,2,3,6}	(45)	(45)	(10)	(25)	∞

Vertex	Cost
2	45
3	45
4	10
5	25
6	∞

Algorithm :-

```
Algorithm Shortestpaths (v, cost, dist, n)
Let V = {1, 2, .... n} and source = 1
 S = {1}
for (i=2; i<=n; i++)
D[i] = C[1,j];
for (i=1; i<=n-1; i++)
{
choose a vertex wεV-S such that D[w] is a
                                minimum;
   S = SU{w};
   for each vertex VεV-S
   D[V] = min (D[V], D[w] + C[w,V])
  }
```

Time Complexity :-

Single source shortest path algorithm takes $O(n^2)$ time.

Example 2, ~~Algoriti~~ Algorithm tracing :

V = {1, 2, 3, 4, 5, 6}

S = {1}

i = 2;

D[2] = C[1,2] = 50

D[3] = C[1,3] = 10

D[4] = C[1,4] = ∞

D[5] = C[1,5] = 45

D[6] = C[1,6] = ∞

for i=1, w = ~~2~~, 3, ~~4~~, ~~5~~, ~~6~~ ← shortest value

w = 3

S = {1, 3}

V = {2, 4, 5, 6}

$D[2] = \min\{D[2], D[3] + C[1,2]\} = \min\{50, 10 + \infty\} = 50$

$D[4] = \min\{D[4], D[3] + C[3,4]\} = \min\{\infty, 10 + 15\} = 25$ ✓

$D[5] = \min\{D[5], D[3] + D[3,5]\} = \min\{45, 10 + \infty\} = 45$

$D[6] = \min\{D[6], D[3] + C[3,6]\} = \min\{\infty, 10 + \infty\} = \infty$

i = 2, w = ~~2~~, 4, ~~5~~, ~~6~~

w = 4

S = {1, 3, 4}

V = {2, 5, 6}, $D[2] = \min\{D[2], D[4] + C[4,2]\} = 45$

$D[5] = \min\{D[5], D[4] + C[4,5]\} = 45$

$D[6] = \min\{D[6], D[4] + C[4,6]\} = \infty$

i = 3 w = 2, 5, 6

w = 5

S = {1, 3, 4, 5}