UNIT-II

Graphs: Breadth First Search, Depth First Search, Spanning trees, connected and bi-connected components.

Greedy Method: General method, applications - Job sequencing with deadlines, knapsack problem, Minimum cost spanning trees, Single source shortest path problem.

****

Prerequisites: **BFS, DFS, Spanning Tree, Spanning trees**

Breadth First Search(BFS):

NOTE: Refer the BFS concept In Data Structures. How BFS Traversal Perform in Graphs

- In breadth first search we start at a vertex **v** and mark it as having been reached (**visited**). The vertex **v** is at this time said to be unexplored (unvisited).
- A vertex is said to have been explored by an algorithm when the algorithm has visited all vertices adjacent from it. All unvisited vertices adjacent from **v** are visited next.
- These are new unexplored vertices. Vertex **v** has now been explored. The newly visited vertices haven't been explored and are put onto the end of a list of unexplored vertices.
- The first vertex on this list is the next to be explored. Exploration continues until no unexplored vertex is left. The list of unexplored vertices operates as a queue and can be represented using any of the standard **queue representation**.

Example:

- If the graph is represented by its adjacency lists as in Figure **(1-a)**, then the vertices get visited in the order1,2, 3,4, 5, 6, 7, 8.
- A breadth first search of the directed graph of **Figure(1-b)** starting at vertex 1 results in only the vertices1,2, and 3 being visited. Vertex 4 cannot be reached from 1.

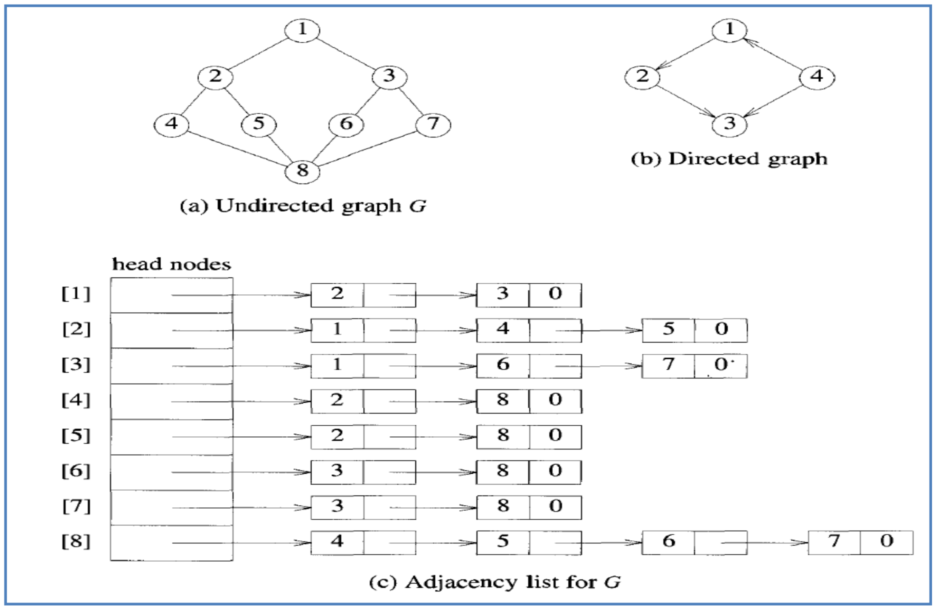


Figure -1

- Let w be a vertex in **V** such that d(v,w) = r + 1. Let **u** be a vertex that immediately precedes w on a shortest **v to w** path.
- Then d(v,u) = r and so u gets visited by BFS. We can assume u !=v and r > 1. Hence, immediately before u gets visited, it is placed on the queue q of unexplored vertices.
- The algorithm doesn't terminate until q becomes empty. Hence, u is removed from q at sometime and all unvisited vertices adjacent from it get visited in the for loop of **line11**of **Algorithm-1**. Hence, w gets visited.
- **Note**: **d(v, w) = r + 1**: This indicates that the distance between two vertices, v and w, is r+1. The distance here is typically the number of edges in the shortest path from v to w.

```
Algorithm BFS(v)
// A breadth first search of G is carried out beginning
// at vertex v. For any node i, visited[i] = 1 if i has
// already been visited. The graph G and array visited[ ]
// are global; visited[ ] is initialized to zero.
{
    u := v; // q is a queue of unexplored vertices.
    visited[v] := 1;
    repeat
    {
        for all vertices w adjacent from u do
        {
            if (visited[w] = 0) then
            {
                Add w to q; // w is unexplored.
                visited[w] := 1;
            }
        }
        if q is empty then return; // No unexplored vertex.
        Delete u from q; // Get first unexplored vertex.
    } until(false);
}
```

Algorithm-1: Pseudo code for breadth first search

```
Algorithm BFT(G, n)
// Breadth first traversal of G
{
    for i := 1 to n do // Mark all vertices unvisited.
        visited[i] := 0;
    for i := 1 to n do
        if (visited[i] = 0) then BFS(i);
}
```

Algorithm-2: Pseudo code for breadth first graph traversal

Time complexity

1. Using Adjacency List:

Average Case:

In BFS, the algorithm explores a graph level by level, visiting vertices and their neighbors. The time complexity depends on how the graph is represented.

- **Initialization:** You initialize a visited array to mark whether a vertex has been visited. This takes **O(n)** time, where **n** is the number of vertices.
- **Exploring Vertices:** BFS explores each vertex once. For each vertex **u**, you examine all its adjacent vertices by iterating through its adjacency list. The total time spent examining all adjacency lists is **proportional to the sum of the degrees of all vertices**, i.e., $\sum_{u \in V} d(u)$, where d(u) is the degree (or out-degree) of vertex **u.**
- Since the sum of the degrees of all vertices in a graph is equal to **2e** for an undirected graph and **e** for a directed graph (where e is the number of edges), the time for examining all edges is **O(e)**.
- **Total Time Complexity:** The total time complexity for BFS using an adjacency list is therefore:

O(n+e)

  - ✓ O(n) for initializing the visited array.
  - ✓ O(e) for processing the edges (visiting the neighbors of each vertex).

1. Sparse Graph:

A **sparse graph** is one in which the number of edges e is relatively small compared to the number of vertices n. In other words, the graph does not have many connections between its vertices.

Characteristics of Sparse Graphs:

- **Low edge-to-vertex ratio**: The number of edges is much smaller than the number of vertices squared, i.e., $e \ll n^2$.
- **Edge count e** is often O(n) or smaller. For example, trees (a connected acyclic graph) are always sparse with **$e=n-1$**.

2. Dense Graph:

A **dense graph** is one in which the number of edges **e** is relatively large compared to the number of vertices **n**. In other words, there are many edges between the vertices.

Characteristics of Dense Graphs:

- **High edge-to-vertex ratio**: The number of edges is close to the maximum number of edges possible, i.e., **$e \approx n^2$**.
- **Edge count e** is large, often approaching $O(n^2)$. For example, a **complete graph** (where every vertex is connected to every other vertex) has **$e=O(n^2)$**.

Best Case:

- The **best case** for BFS occurs when the graph is as sparse as possible (e.g., there are no edges, or only one edge exists).
- For a graph with no edges (i.e., a forest of disconnected components or a completely disconnected graph), the algorithm would simply visit each vertex without needing to explore any edges.

  Time Complexity (Best Case):
  - Initialization of the visited array: O(n)
  - Exploring edges: O(0), since there are no edges.Total time: **O(n)**.
  - If there are edges Total time **O(n+e)**

Worst Case:

- The **worst case** occurs in a **dense graph** where all vertices are connected (e.g., a complete graph). In this case, BFS will need to explore all vertices and edges.

  Time Complexity (Worst Case):
  - ✓ Initialization of the visited array: O(n)
  - ✓ Exploring all edges: O(e), where e is the number of edges.
  - ✓ For a complete graph, e is approximately **$n(n-1)/2$**, leading to a time complexity of **$O(n^2)$**.
  - ✓ Total time: **O(n+e)**.

2. Using Adjacency Matrix:

Average Case:

In an adjacency matrix, the graph is represented as a **n×n** matrix where the entry **matrix[u][v]** is non-zero (or true) if there is an edge between vertices u and **v.**

- **Initialization:** Just like with adjacency lists, BFS requires **O(n)** time to initialize the visited array.
- **Exploring Vertices:** For each vertex **u,** BFS needs to check all possible edges. This means for each vertex, you check all **n** entries in the corresponding row of the adjacency matrix to find its neighbors.
  - Checking each row of the matrix takes **O(n)** time (since the matrix has **n** rows and columns).
  - Since there are **n** vertices to process, this results in a total of **$O(n^2)$** time for exploring all vertices.
- **Total Time Complexity:** The total time complexity for BFS using an adjacency matrix is therefore:

  $O(n^2)$

- **O(n)** for initializing the visited array.
- **O(n^2)** for checking all potential edges in the adjacency matrix.

Best Case:

- The **best case** for BFS with an adjacency matrix would occur in a graph with few edges, where most of the matrix entries are zero (indicating no edges).
- However, regardless of the number of edges, BFS must check all n possible neighbors for each vertex, so the time complexity will still depend on n, even if there are few edges.

  Time Complexity (Best Case):

  - Initialization of the visited array: O(n)
  - Checking each row of the adjacency matrix to find neighbors**: O(n)** for each vertex.
  - Since you check all n rows for each vertex, the time complexity is **O(n^2)** even in the best case (when there are few edges).

    Total time: O(n^2).

Worst Case:

- The **worst case** occurs when the graph is fully connected (i.e., a complete graph), so the adjacency matrix will have almost all entries filled (except for the diagonal in an undirected graph). BFS will still need to examine all the rows of the matrix for each vertex.

  Time Complexity (Worst Case):

  - Initialization of the visited array: O(n)
  - Checking all **n** rows for each vertex: **O(n)** per vertex.

    Total time: O(n^2).

Depth First Search (DFS):

NOTE: Refer the DFS concept In Data Structures. How DFS Traversal Perform in Graphs

- ❖ A depth first search of a graph differs from a breadth first search in that the Exploration of a vertex **v** is suspended as soon as a new vertex is reached. At this time the exploration of the new vertex **u** begins.
- ❖ When this new vertex has been explored, the exploration of v continues. The search terminates when all reached vertices have been fully explored.
- ❖ This search process is best described recursively as in **Algorithm -3.**

```
Algorithm DFS(v)
// Given an undirected (directed) graph $G = (V, E)$ with
// $n$ vertices and an array visited[ ] initially set
// to zero, this algorithm visits all vertices
// reachable from $v$. $G$ and visited[ ] are global.
{
    visited[v] := 1;
    for each vertex $w$ adjacent from $v$ do
    {
        if (visited[w] = 0) then DFS(w);
    }
}
```

Algorithm -3 Depth first search of a graph

Example: depth first search of the graph of Figure-1(a)starting at vertex1 and using the adjacency lists of Figure-1(c) results in the vertices being visited in the order 1,2, 4, 8, 5, 6, 3,7.

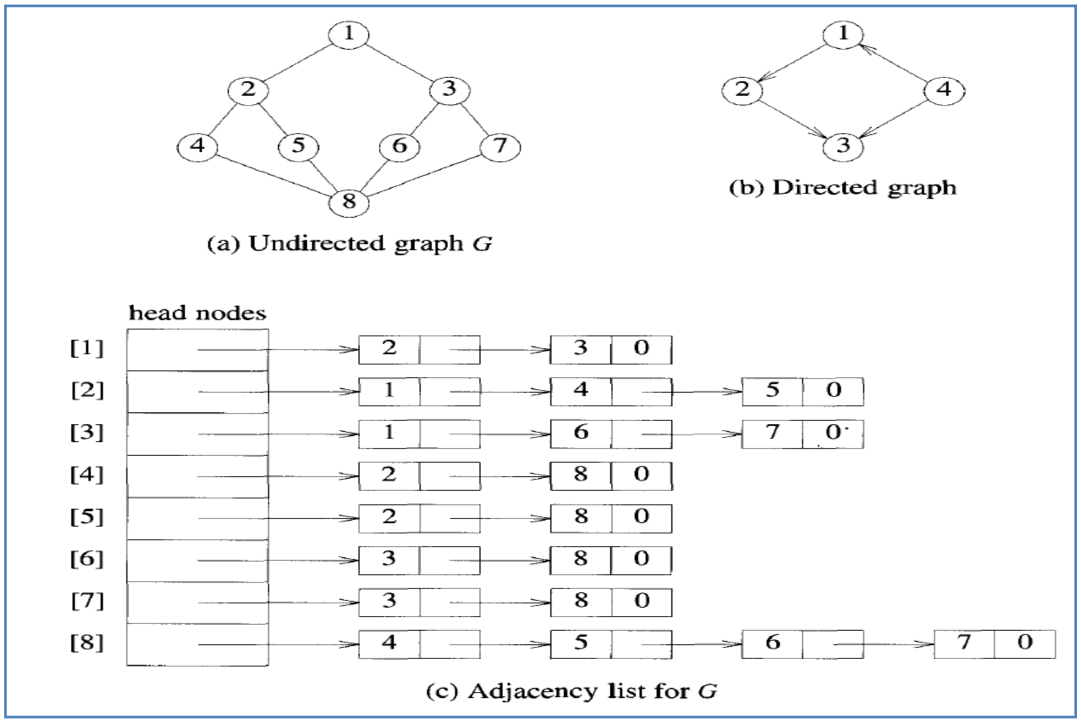


Figure-1

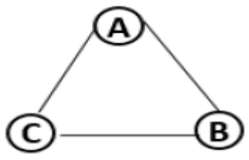
Time complexity:

- ❖ One can easily prove that DFS visits all vertices reachable from vertex **v.** If T(n,e) and S(n,e) represent the maximum time and maximum additional Space taken by DFS for an n-vertex e-edge graph, then **S(n,e) = O(n)** and
- ❖ **T(n,e)= O(n+ e)** if adjacency lists are used and **T(n,e)= O(n^2)** if Adjacency matrices are used. A depth first traversal of a graph is carried out by repeatedly calling DFS, with a new unvisited starting vertex each time.
- ❖ The algorithm for this (DFT) differs from BFT only in that the call to BFS (i) is replaced by a call to DFS (i).

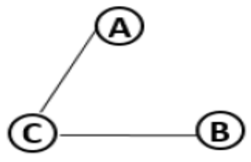
Spanning trees:

- ➢ 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..
- ➢ By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.
- ➢ A spanning tree consists of (n-1) edges, where 'n' is the number of vertices (or nodes). Edges of the spanning tree may or may not have weights assigned to them. All the possible spanning trees created from the given graph G would have the same number of vertices, but the number of edges in the spanning tree would be equal to the number of vertices in the given graph minus 1.
- ➢ A complete undirected graph can have $\mathbf{n^{n-2}}$ number of spanning trees where **n** is the number of vertices in the graph. Suppose, if **n = 5**, the number of maximum possible spanning trees would be $\mathbf{5^{5-2} = 125}$.

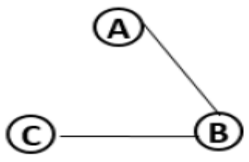
Example:1



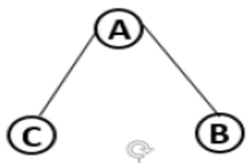
The maximum number of spanning trees for above graph is $3^{3-2} = 3^1 = 3$
The spanning trees are shown below



Spanning tree 1



Spanning tree 2



Spanning tree 3

Applications of the Spanning Tree:

A spanning tree is preferred to discover the shortest path to link all nodes of the graph. In this section, we will discuss some of the most famous and common applications of the spanning tree:

- A spanning tree is very helpful in the civil network zone and planning.
- It is used in network routing protocol.

Properties of spanning-tree:

Some of the properties of the spanning tree are given as follows -

- There can be more than one spanning tree of a connected graph G.
- A spanning tree does not have any cycles or loop.
- A spanning tree is **minimally connected,** so removing one edge from the tree will make the graph disconnected.
- A spanning tree is **maximally acyclic,** so adding one edge to the tree will create a loop.
- There can be a maximum $\mathbf{n^{n-2}}$ number of spanning trees that can be created from a complete graph.
- A spanning tree has **n-1** edges, where 'n' is the number of nodes.
- If the graph is a complete graph, then the spanning tree can be constructed by removing maximum **(e-n+1)** edges, where 'e' is the number of edges and 'n' is the number of vertices.

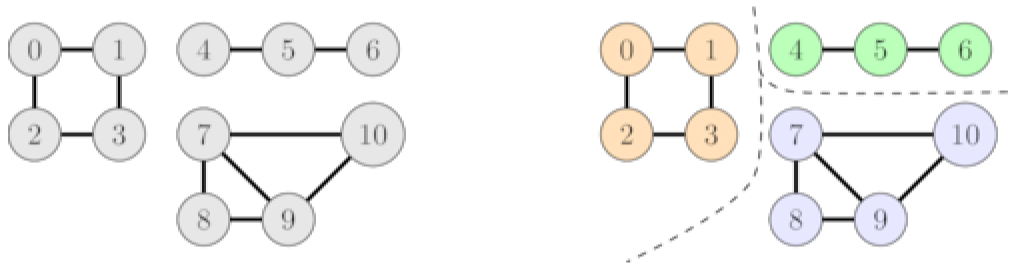
So, a spanning tree is a subset of connected graph G, and there is no spanning tree of a disconnected graph.

Connected components:

- ❖ In graph theory, a connected component (or just component) of an undirected graph is a sub graph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the super graph.
- ❖ For example, the graph shown in the illustration has three connected components. A vertex with no incident edges is itself a connected component. A graph that is itself connected has exactly one connected component, consisting of the whole graph.
- ❖ In order to find a connected component of an undirected graph, we can just pick a vertex and start doing a search **(BFS or DFS)** from that vertex

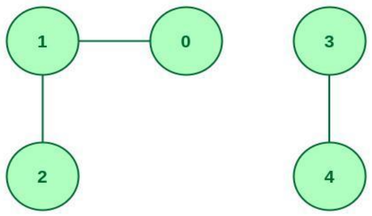
Example-1

The following figure shows a graph on the left and its connected components on the right. Nodes on the same component are colored with the same color.



On the above graph, if we call BFS on node 0 we will visit nodes 0, 1, 2 and 3. In general, when we call BFS on node u, it will visit the whole connected component of u.

Example-2



Explanation: There are 2 different connected components.

They are {0, 1, 2} and {3, 4}

Bi-connected components:

- ❖ A graph G is bi-connected if and only if it contains no articulation points. The graph of **Figure-4(a)** is not bi-connected.
- ❖ The graph of **Figure-5** is Bi-connected. The presence of **articulation points** in a connected graph can be an undesirable feature in many cases.

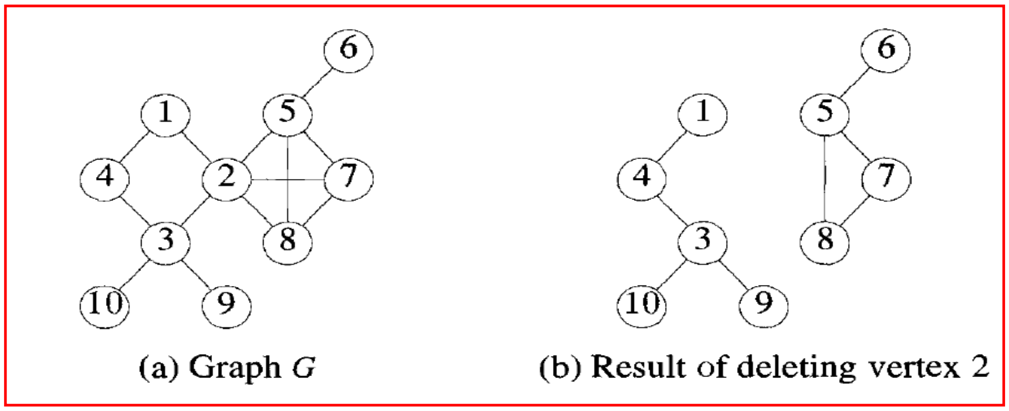


Figure-4: An example graph- not bi-connected

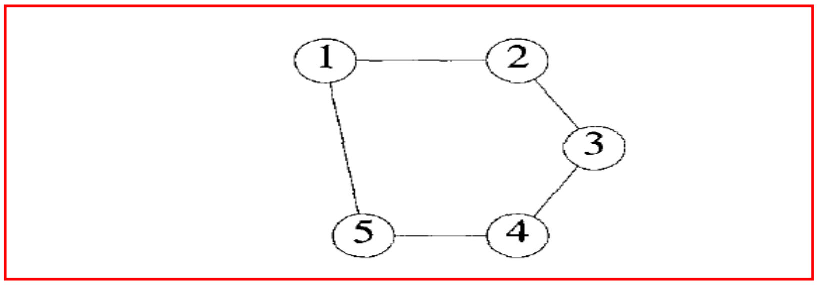
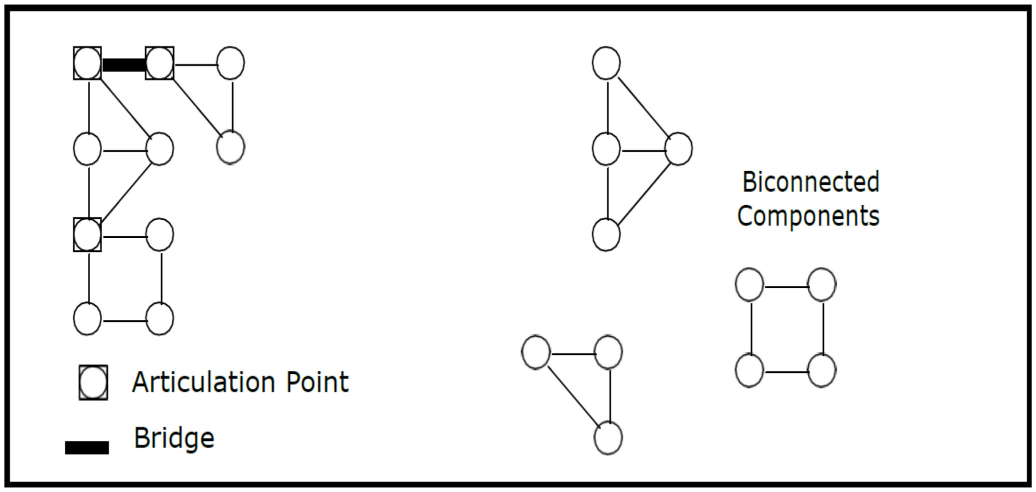


Figure-5: An example graph- Bi-connected

Let G = (V, E) be a connected undirected graph. Consider the following definitions:

- **Articulation Point (or Cut Vertex):** An articulation point in a connected graph is a vertex (together with the removal of any incident edges) that, if deleted, would break the graph into two or more pieces.
- **Bridge:** Is an edge whose removal results in a disconnected graph.
- **Bi-connected:** A graph is bi-connected if it contains no articulation points. In a bi-connected graph, two distinct paths connect each pair of vertices. A graph that is not bi-connected divides into bi-connected components. This is illustrated in the following figure:



Articulation Points and Bridges

❖ Using the above scheme to transform the graph of **Figure-4(a)** into a bi-connected graph requires us to

- ✓ Add edges(4,10) and (10,9) (corresponding to the articulation point 3)
- ✓ Add edge(1,5)(corresponding to the Articulation point 2), and
- ✓ Add edge (6,7) (corresponding to point 5).

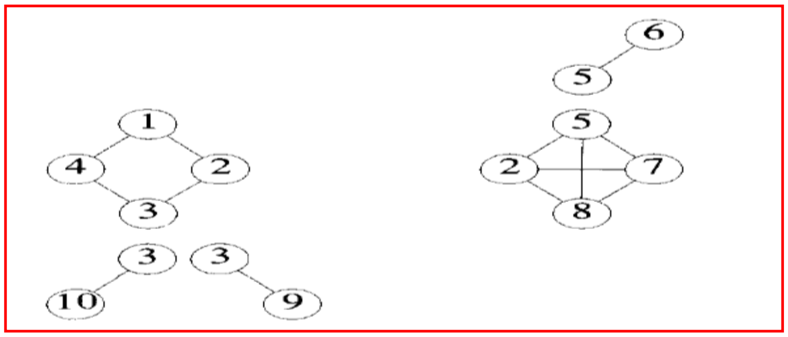


Figure-6 :- Bi-connected components of graph of Figure-4(a)

Now, let us attack the problem of **identifying the articulation points and Bi-connected components** of a connected graph G with **n > 2** vertices. The Problem is efficiently solved by considering a **depth first spanning tree of G**.

- ➢ Figure-7(a) and (b) shows a depth first spanning tree of the graph of Figure-4(a). In each figure there is a number outside each vertex. These numbers correspond to the order in which a depth first search visits these Vertices and are referred to as the **depth first numbers** (**dfns**) of the vertex.
- ➢ Thus, dfn[1]= 1,dfn[4] = 2,dfn[6] = 8,and soon. In Figure-7(b) solid edges form the depth first spanning tree. These edges are called tree edges.
- ➢ Broken edges (i.e. all the remaining edges) are called back edges.

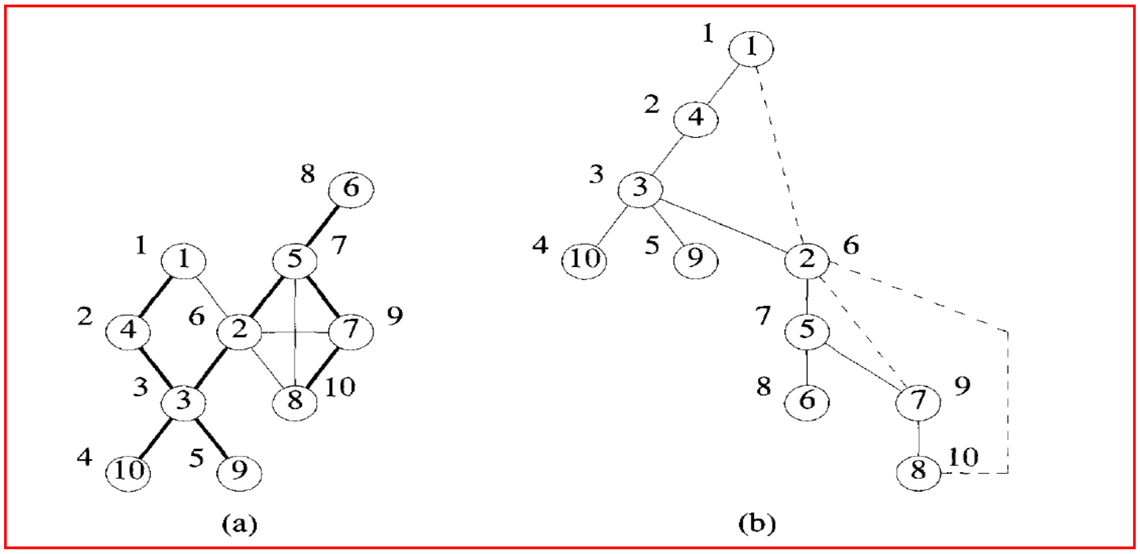


Figure-7: A depth first spanning tree of the graph of Figure-4(a)

- Let us consider the typical case of vertex **v**, where v is not a leaf and v is not the root.
- Let w1, w2, ... wk be the children of v. For each child there is a sub tree of the DFS tree rooted at this child.
- If for some child, there is no back edge going to a proper ancestor of v, and then if we remove v, this sub tree becomes disconnected from the rest of the graph, and hence v is an articulation point.

L (u) = min {DFN (u), min {L (w) |w is a child of u}, min {DFN (w) |(u, w) is a back edge}}.

L (u) is the lowest depth first number that can be reached from 'u' using a path of descendents followed by at most one back edge. It follows that, If 'u' is not the root then 'u' is an articulation point iff 'u' has a child 'w' such that: **L (w) ≥ DFN (u).**

Algorithm for finding the Articulation points:

```
Algorithm Art(u, v)
// u is a start vertex for depth first search. v is its parent if any
// in the depth first spanning tree. It is assumed that the global
// array dfn is initialized to zero and that the global variable
// num is initialized to 1. n is the number of vertices in G.
{
    dfn[u] := num; L[u] := num; num := num + 1;
    for each vertex w adjacent from u do
    {
        if (dfn[w] = 0) then
        {
            Art(w, u); // w is unvisited.
            L[u] := min(L[u], L[w]);
        }
        else if (w ≠ v) then L[u] := min(L[u], dfn[w]);
    }
}
```

Algorithm-7: Pseudo code to compute dfn and L

To identify the articulation points, we use: **Figure-7**:

L (u) = min {DFN (u), min {L (w) |w is a child of u}, min {DFN (w) |w is a vertex to which there is back edge from u}}

L (1) = min {DFN (1), min {L (4)}} = min {1, L (4)} = min {1, 1} = 1

L (4) = min {DFN (4), min {L (3)}} = min {2, L (3)} = min {2, 1} = 1

L (3) = min {DFN (3), min {L (10), L (9), L (2)}} =
= min {3, min {L (10), L (9), L (2)}} = min {3, min {4, 5, 1}} = 1

L (10) = min {DFN (10)} = 4

L (9) = min {DFN (9)} = 5

L (2) = min {DFN (2), min {L (5)}, min {DFN (1)}}
= min {6, min {L (5)}, 1} = min {6, 6, 1} = 1

L (5) = min {DFN (5), min {L (6), L (7)}} = min {7, 8, 6} = 6

L (6) = min {DFN (6)} = 8
L (7) = min {DFN (7), min {L (8}, min {DFN (2)}}
= min {9, L (8) , 6} = min {9, 6, 6} = 6

L (8) = min {DFN (8), min {DFN (5), DFN (2)}}

= min {10, min (7, 6)} = min {10, 6} =6

Therefore, L (1: 10) = (1, 1, 1, 1, 6, 8, 6, 6, 5,4)

Vertex	1	2	3	4	5	6	7	8	9	10
dfn(u)	1	**6**	**3**	2	**7**	8	9	10	5	4
L(u)	1	1	1	1	6	8	6	6	5	4

Finding the Articulation Points:

Vertex 1: Vertex 1 is not an articulation point. It is a root node. Root is an articulation point if it has two or more child nodes.

Vertex 2: is an articulation point as child 5 has L (5) = 6 and DFN (2) = 6, So, the condition L (5) = DFN (2) is true.

Vertex 3: is an articulation point as child 10 has L (10) = 4 and DFN (3) =3, So, the condition L (10) > DFN (3) is true.

Vertex 4: is not an articulation point as child 3 has L (3) = 1and DFN(4)=2,So,theconditionL(3)≥DFN (4)isfalse.

Vertex 5: is an articulation point as child 6 has L (6) = 8, and DFN(5)=7,So,theconditionL(6)>DFN (5) istrue.

Vertex 7: is not an articulation point as child 8 has L (8) = 6,and DFN(7)=9,So,theconditionL(8)≥DFN (7) isfalse.

Vertex 6, Vertex 8, Vertex 9 and Vertex 10 are leaf nodes.

Therefore, the articulation points are {2, 3, 5}.

Greedy Method

- ❖ The greedy method is one of the strategies like Divide and conquer used to solve the problems.
- ❖ This method is used for solving **optimization problems**. An optimization problem is a problem that demands either maximum or minimum results.
- ❖ Let's understand through some terms.
- ❖ The Greedy method is the simplest and straightforward approach. It is not an algorithm, but it is a technique. The main function of this approach is that the decision is taken on the basis of the currently available information. Whatever the current information is present, the decision is made without worrying about the effect of the current decision in future.
- ❖ This technique is basically used to determine the feasible solution that may or may not be optimal. The feasible solution is a subset that satisfies the given criteria. The optimal solution is the solution which is the best and the most favorable solution in the subset.
- ❖ In the case of feasible, if more than one solution satisfies the given criteria then those solutions will be considered as the feasible, whereas the optimal solution is the best solution among all the solutions.

General method:

The greedy algorithm works in various stages and considers one input at a time while proceeding. At each stage, we try to find a feasible solution for our problem, a solution that is good for the algorithm at that point. This means that we make a locally optimal choice for each sub-task. These local optimal choices eventually lead us to the global optimal choice for our problem which we call the final optimal solution.

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

Algorithm-8: The pseudo-code for the simplest greedy algorithm

Here, A = an array of size n.

Greedy method consists of 3 functions (steps).

- **Select:** it selects an input from array a[ ] (candidate set) and puts in the variable x.
- **Feasible:** it is a Boolean function which checks whether the selected input meets the constraints or not.
- **Union:** if the selected input i.e. 'x' makes the solution feasible, then x is included in the solution and objective function get updated

The above is the greedy algorithm. Initially, the solution is assigned with zero value. We pass the array and number of elements in the greedy algorithm. Inside the for loop, we select the element one by one and checks whether the solution is feasible or not. If the solution is feasible, then we perform the union.

Let's understand through an example.

Suppose there is a problem 'P'. I want to travel from A to B shown as below:

P : A → B

The problem is that we have to travel this journey from A to B. There are various solutions to go from A to B. We can go from A to B by walk, car, bike, train, aeroplane, etc. There is a constraint in the journey that we have to travel this journey within 12 hrs. If I go by train or aeroplane then only, I can cover this distance within 12 hrs. There are many solutions to this problem but there are only two solutions that satisfy the constraint.

The problem that requires either minimum or maximum result then that problem is known as an optimization problem. Greedy method is one of the strategies used for solving the optimization problems.

Job sequencing with deadlines:

- There is set of n-jobs. For any job i, is a integer deadline **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-1: Consider the following 5 jobs and their associated deadline and profit

Jobs	J1	J2	J3	J4	J5
Deadlines	2	1	3	2	1
Profits	60	100	20	40	20

- Sort the jobs according to their profit in descending order.
- Note! If two or more jobs are having the same profit then sorts them as per their entry in the job list

Jobs	J2	J1	J4	J3	J5
Deadlines	1	2	2	3	1
Profits	100	60	40	20	20

- Find the maximum deadline value
- Looking at the jobs we can say the max deadline value is 3. So, **dmax = 3**
- As **dmax = 3** so we will have THREE slots to keep track of free time slots. Set the time slot status to EMPTY

time slot	1	2	3
status	EMPTY	EMPTY	EMPTY

- Total number of jobs is 5. So we can write n = 5.
- Note! If we look at job j2, it has a deadline 1. This means we have to complete job j2 in time slot 1 if we want to earn its profit.
- Similarly, if we look at job j1 it has a deadline 2. This means we have to complete job j1 on or before time slot 2 in order to earn its profit.

- Similarly, if we look at job j3 it has a deadline 3. This means we have to complete job j3 on or before time slot 3 in order to earn its profit. Our objective is to select jobs that will give us higher profit.

time slot	1	2	3
Jobs	J1	J2	J3
Profits	100	60	20

Total Profit is 180

Example-2:

Let n = 4, (P1,P2,P3,P4)=(100,10,15,27) and (d1,d2,d3,d4)= (2,1,2,1) The feasible solutions and their values are:

Feasible solution	**Processing sequence**	**Value**
j1 j2 (1, 2)	(2,1)	100+10=110
(1,3)	(1,3) or (3,1)	100+15=115
(1,4)	(4,1)	100+27=127
(2,3)	(2,3)	10+15=25
(3,4)	(4,3)	15+27=42
(1)	(1)	100
(2)	(2)	10
(3)	(3)	15
(4)	(4)	27

- In the example **solution '3'** is the optimal. In this solution only jobs 1&4 are processed and the value is 127. These jobs must be processed in the order **j4** followed by j1. Thus process of job 4 begins at time 0 and ends at time 1. And the processing of job 1 begins at time 1 and ends at time2.

Algorithm-9:

```
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 $J \cup \{i\}$ can be completed
            by their deadlines) then $J := J \cup \{i\}$;
    }
}
```

- Therefore both the jobs are completed within their deadlines. The optimization measure for determining the next job to be selected in to the solution is according to the profit.
- The next job to include is that which increases Σpi the most, subject to the constraint that the resulting **"j"** is the feasible solution. Therefore the greedy strategy is to consider the jobs in decreasing order of profits.

Example-3:

We are given the jobs, their deadlines and associated profits as shown-

Jobs	**J1**	**J2**	**J3**	**J4**	**J5**	**J6**
Deadlines	5	3	3	2	4	2
Profits	201	181	191	301	121	101

Answer the following questions-

- ✓ Write the optimal schedule that provides us the maximum profit.
- ✓ Can we complete all the jobs in the optimal schedule?
- ✓ What is the maximum earned profit?

Solution:

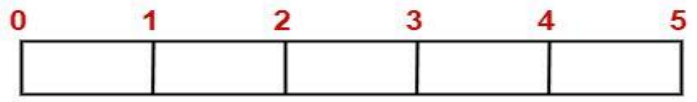
Step-01:

Firstly, we need to sort all the given jobs in decreasing order of their profit as follows.

Jobs	**J4**	**J1**	**J3**	**J2**	**J5**	**J6**
Deadlines	2	5	3	3	4	2
Profits	300	200	190	180	120	100

Step-02:

- For each step, we calculate the value of the maximum deadline.
- Here, the value of the maximum deadline is 5.
- So, we draw a Gantt chart as follows and assign it with a maximum time on the Gantt chart with 5 units as shown below.



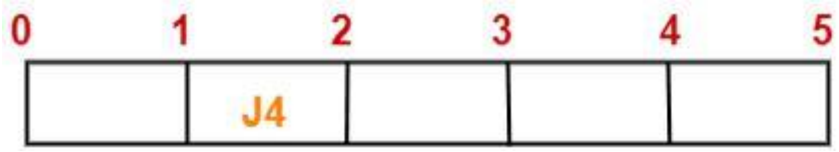
Gantt Chart

Now,

- We will be considering each job one by one in the same order as they appear in the Step-01.
- We are then supposed to place the jobs on the Gantt chart as far as possible from 0.

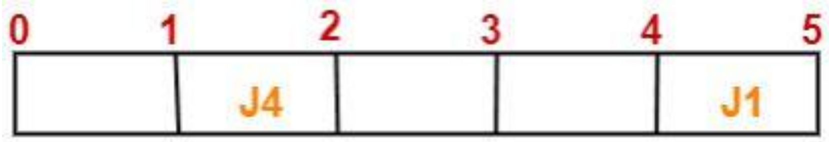
Step-03:

- We now consider job4.
- Since the deadline for job4 is 2, we will be placing it in the first empty cell before deadline 2 as follows.



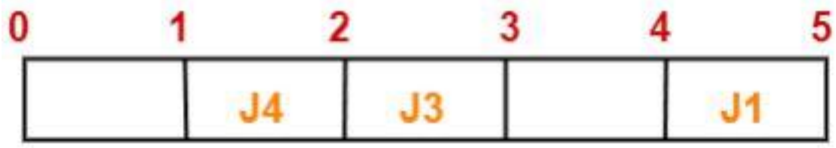
Step-04:

- Now, we go with job1.
- Since the deadline for job1 is 5, we will be placing it in the first empty cell before deadline 5 as shown below.



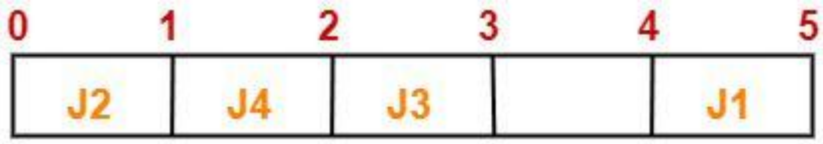
Step-05:

- We now consider job3.
- Since the deadline for job3 is 3, we will be placing it in the first empty cell before deadline 3 as shown in the following figure.



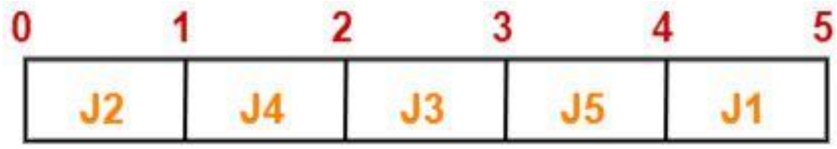
Step-06:

- Next, we go with job2.
- Since the deadline for job2 is 3, we will be placing it in the first empty cell before deadline 3.
- Since the second cell and third cell are already filled, so we place job2 in the first cell as shown below.



Step-07:

- Now, we consider job5.
- Since the deadline for job5 is 4, we will be placing it in the first empty cell before deadline 4 as shown in the following figure.



Now,

- We can observe that the only job left is job6 whose deadline is 2.
- Since all the slots before deadline 2 are already occupied, job6 cannot be completed.

Now, the questions given above can be answered as follows:

Part-01:

The optimal schedule is-

Job2, Job4, Job3, Job5, Job1

In order to obtain the maximum profit this is the required order in which the jobs must be completed.

Part-02:

- As we can observe, all jobs are not completed on the optimal schedule.
- This is because job6 was not completed within the given deadline.

Part-03:

Maximum earned profit = Sum of the profit of all the jobs from the optimal schedule

= Profit of job2 + Profit of job4 + Profit of job3 + Profit of job5 + Profit of job1

= 181 + 301 + 191 + 121 + 201

= 995 units

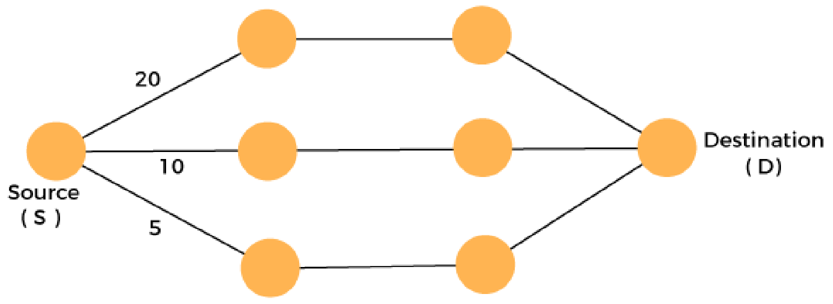
Analysis of the algorithm:

In the job sequencing with deadlines algorithm, we make use of two loops, one loop within another. Hence, the complexity of this algorithm would be **O(n^2).**

Disadvantages of using Greedy algorithm:

- Greedy algorithm makes decisions based on the information available at each phase without considering the broader problem. So, there might be a possibility that the greedy solution does not give the best solution for every problem.
- It follows the local optimum choice at each stage with a intend of finding the global optimum. Let's understand through an example.

Consider the graph which is given below:



We have to travel from the source to the destination at the minimum cost. Since we have three feasible solutions having cost paths as 10, 20, and 5. 5 is the minimum cost path so it is the optimal solution. This is the local optimum, and in this way, we find the local optimum at each stage in order to calculate the global optimal solution

Knapsack Problem

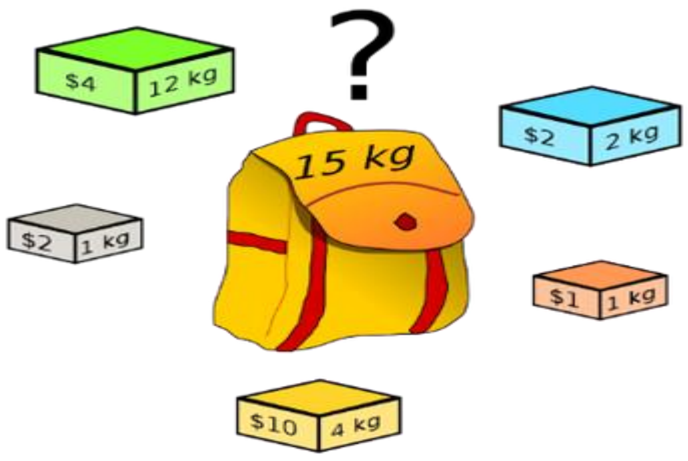
- ❖ Knapsack Problem Here knapsack is like a container or a bag. Suppose we have given some items which have some weights or profits.
- ❖ We have to put some items in the knapsack in such a way total value produces a maximum profit. For example, the weight of the container is 20 kg.
- ❖ We have to select the items in such a way that the sum of the weight of items should be either smaller than or equal to the weight of the container, and the profit should be maximum.

There are two types of knapsack problems:

1. 0/1 knapsack problem
2. Fractional knapsack problem

Fractional knapsack problem:

- ➢ The basic idea of the greedy approach is to calculate the ratio **profit/weight** for each item and sort the item on the basis of this ratio. Then take the item with the highest ratio and add them as much as we can (can be the whole element or a fraction of it).
- ➢ This will always give the maximum profit because; in each step it adds an element such that this is the maximum possible profit for that much weight.



Knapsack Problem

Example-1:

For the given set of items and knapsack **capacity = 60** kg, find the optimal solution for the fractional knapsack problem making use of greedy approach.

(or)

A thief enters a house for robbing it. He can carry a **maximal weight of 60 kg** into his bag. There are 5 items in the house with the following weights and values. What items should thief take if he can even take the fraction of any item with him?

Item	Weight	Value
1	5	30
2	10	40
3	15	45
4	22	77
5	25	90

Solution

Step-01: Compute the value / weight ratio for each item-

Items	Weight	Value	Ratio
1	5	30	6
2	10	40	4
3	15	45	3
4	22	77	3.5
5	25	90	3.6

Step-02: Sort all the items in decreasing order of their value / weight ratio-

I1	I2	I5	I4	I3
(6)	(4)	(3.6)	(3.5)	(3)

Step-03: Start filling the knapsack by putting the items into it one by one.

Knapsack Weight	Items in Knapsack	Cost
60	Ø	0
55	I1	30
45	I1, I2	70
20	I1, I2, I5	160

Now,

Knapsack weight left to be filled is 20 kg but item-4 has a weight of 22 kg.

Since in fractional knapsack problem, even the fraction of any item can be taken.

So, knapsack will contain the following items-

< I1 , I2 , I5 , (20/22) I4 >

Total cost of the knapsack

= 160 + (20/22) x 77

= 160 + 70

= 230 units

Example-2:

Let us consider that the capacity of the knapsack(bag) W = 60 and the list of items are shown in the following table –

Item	A	B	C	D
Profit	281	101	121	121
Weight	40	10	20	24
Ratio (piwi)	7	10	6	5

We can see that the provided items are not sorted based on the value of **piwi**, we perform sorting. After sorting, the items are shown in the following table.

Item	B	A	C	D
Profit	101	281	121	121
Weight	10	40	20	24
Ratio (piwi)	10	7	6	5

- Once we sort all the items according to the piwi, we choose all of B as the weight of B is less compared to that of the capacity of the knapsack.
- Further, we choose item A, as the available capacity of the knapsack is greater than the weight of A. Now, we will choose C as the next item.
- Anyhow, the whole item cannot be chosen as the remaining capacity of the knapsack is less than the weight of the chosen item – C.
- Hence, a fraction of C (i.e. (60 – 50)/20) is chosen.
- Now, we reach the stage where the capacity of the Knapsack is equal to the chosen items. Hence, no more items can be selected.
- The total weight of the chosen items is 40 + 10 + 20 * (10/20) = 60
- And the total profit is 101 + 281 + 121 * (10/20) = 380 + 60 = 440
- This is the optimal solution. We cannot gain more profit compared to this by selecting any different combination of items out of the provided items.

Algorithm:

- Let us apply the greedy method to solve the knapsack problem. We are given 'n' objects and a knapsack
- 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.

Algorithm FractionalKnapsack (m, n)

```
// m is the knapsack capacity      // n is the number of items
// P[1 : n] is an array of profits     // w[1 : n] is an array of weights

// Step 1: Create an array of items with their profit-to-weight ratios
for i := 1 to n do
   ratio[i] := P[i] / w[i]  // Calculate the profit-to-weight ratio for each item

// Step 2: Sort the items in descending order of the ratio
SortItemsByRatioDescending(ratio, P, w)

// Step 3: Initialize the knapsack solution and remaining capacity
totalProfit := 0.0      // Total profit that we will accumulate
remainingCapacity := m  // Remaining capacity of the knapsack
x[1 : n] := 0.0       // Solution vector, initially 0 (not selected)

// Step 4: Process each item in order of decreasing ratio
for i := 1 to n do
   if w[i] <= remainingCapacity then
      // If the item can be fully included
      x[i] := 1.0

      totalProfit := totalProfit + P[i]
      remainingCapacity := remainingCapacity - w[i]
   else
      // If the item cannot be fully included, take the fractional part
      x[i] := remainingCapacity / w[i]
      totalProfit := totalProfit + (x[i] * P[i])
      remainingCapacity := 0  // Knapsack is full, no more capacity left
      break  // Stop further processing, since the knapsack is full

// Step 5: Return the solution

return totalProfit, x
```

Algorithm-10

Explanation:

1. **Calculate the profit-to-weight ratio**: For each item, compute the ratio P[i]\{w[i] (profit per unit weight). This helps determine which items are more "valuable" in terms of profit per weight.
2. **Sort the items**: Sort the items based on the calculated ratio in descending order. This allows us to consider the most valuable items first.

3. **Greedy selection**:
   - **Fully Include Items**: If an item fits completely in the remaining capacity of the knapsack (i.e., w[i]≤ remaining capacity), include it fully.
   - **Partially Include Item**: If an item does not fully fit, take the fractional part that fits in the remaining capacity.
   - **Update Capacity**: After selecting an item (either fully or partially), reduce the remaining capacity of the knapsack.
4. **Return the total profit and the solution vector**: The total profit is the sum of the profits of the items selected (either fully or partially). The solution vector x[i] will store the proportion of each item that was included in the knapsack.

Example:
Given:

- **m** = 50 (knapsack capacity)
- **n** = 4 (4 items)
- **P** = [60, 100, 120, 150] (profits)
- **w** = [10, 20, 30, 40] (weights)

Step 1: Calculate the profit-to-weight ratio for each item:

- Item 1: 60/10=6.0
- Item 2: 100/20=5.0
- Item 3: 120/30=4.0
- Item 4: 150/40=3.75

Step 2: Sort by the ratio in descending order:
- Sorted order: Item 1 (6.0), Item 2 (5.0), Item 3 (4.0), Item 4 (3.75)

Step 3: Start filling the knapsack:
- Item 1 fits completely (weight = 10, remaining capacity = 50). Include it fully, totalProfit = 60, remaining capacity = 40.
- Item 2 fits completely (weight = 20, remaining capacity = 40). Include it fully, totalProfit = 160, remaining capacity = 20.
- Item 3 fits completely (weight = 30, remaining capacity = 20). This item doesn't fit completely, so include part of it: 20/30=0.6667. Add 0.6667×120=80 to the total profit, totalProfit = 240, remaining capacity = 0.

Return:

- Total Profit = 240
- Solution vector x = [1.0, 1.0, 0.6667, 0.0] (items 1 and 2 are fully included, and item 3 is partially included).

Time Complexity Analysis:

The time complexity of the Fractional Knapsack problem depends on the operations performed in the algorithm. Let's break down the steps involved:

Step-by-step breakdown:

1. **Calculating value-to-weight ratios**:
   - For each item, we need to calculate the value-to-weight ratio (vi/wi), which is done in **O(1)** time for each item.
   - If there are n items, this step takes **O(n)** time.
2. **Sorting the items**:
   - After computing the ratios, the next step is to sort the items in descending order of their value-to-weight ratio.
   - Sorting n items takes **O(n log n)** time using efficient sorting algorithms (like Merge Sort, Quick Sort, etc.).
3. **Greedy selection of items**:
   - After sorting, we go through the sorted list and either take a fraction of the item or the whole item. This step involves a simple linear scan of the sorted list and updating the knapsack's weight and value.
   - This takes **O(n)** time.

Total Time Complexity:

The overall time complexity is dominated by the sorting step, which is **O(n log n)**.

- **Best Case**: **O(n log n)**
  Even in the best case, the sorting operation dominates the complexity, so the best case is still **O(n log n)**.
- **Average Case**: **O(n log n)**
  The average case also depends on the sorting step, so the average case is **O(n log n)**.
- **Worst Case**: **O(n log n)**
  Similarly, the worst-case complexity is dominated by the sorting step, so it is **O(n log n)**

Applications

For multiple cases of resource allocation problems that have some specific constraints, the problem can be solved in a way that is similar to the Knapsack problem. Following are a set of examples.

- ✓ Finding the least wasteful way to cut down the basic materials
- ✓ portfolio optimization
- ✓ Cutting stock problems

Minimum cost spanning trees

Prim's Algorithm:

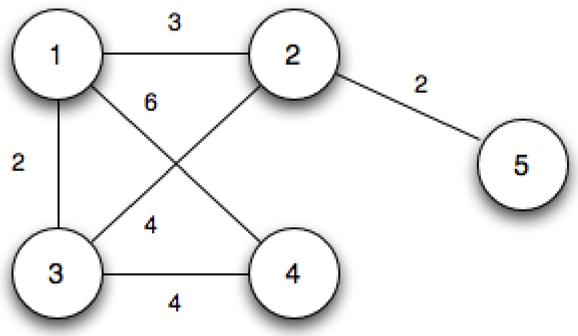
- A greedy method to obtain a minimum-cost spanning tree builds this tree Edge by edge.
- The next edge to include is chosen according to some optimization criterion. The simplest such criterion is to choose an edge that result in a minimum increase in the sum of the costs of the edges so far included.
- There are two possible ways to interpret this criterion. In the first, the set of edges so far selected form a tree. Thus, if A is the set of edges selected so far, then A forms a tree.
- The next edge(u,v) to be included in A is a minimum-cost edge not in A with the property that A U {(u,v)}is also a tree.

NOTE: Refer the Prim's Algorithm: concept In Data Structures. How a minimum-cost spanning tree builds using Prim's Algorithm

- Let us obtain a pseudo code algorithm to find a minimum-cost spanning tree using this method.
- The algorithm will start with a tree that includes only a minimum-cost edge of G.
- Then, edges are added to this tree one by one. The next **edge(i,j)** to be Added is such that **i** is a vertex already included in the tree, **j** is a vertex not yet included, and the cost of (i,j), cost[i,j], is minimum among all **edges {k,l)**such that vertex k is in the tree and vertex **l** is not in the tree.
- To Determine this edge(i,j) efficiently, we associate with each vertex j not yet Included in the tree a value near[j]. The value near[j]is a vertex in the tree Such that **cost [j, near[j]]** is minimum among all choices for near[j].
- We define near[j]= 0 for all vertices j that are already in the tree. The next edge to include is defined by the vertex j such that **near[j]!= 0** (j not already in the tree) and **cost[j, near[j]]** is minimum.

- In function Prim (**Algorithm 11**), line 9 selects a minimum-cost edge.
- Lines 10 to 15 initialize the variables so as to represent a tree comprising only the edge(k,l).
- In the for loop of line16 the remainder of the spanning tree is built up edge by edge. Lines18 and 19 select (j, near[j]) as the next edge to include. Lines 23 to 25 update near[ ].

Example-1: Consider Below graph



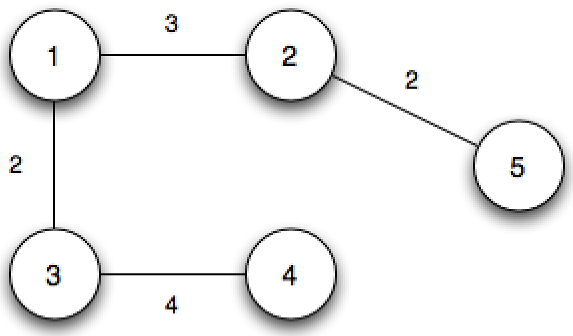


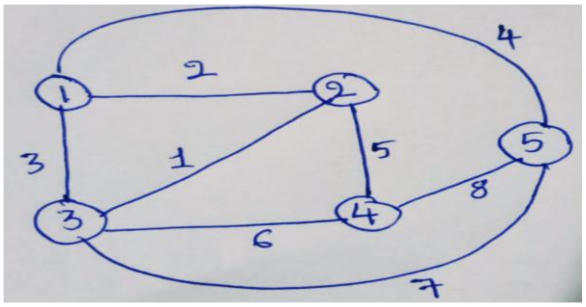
Figure: A graph and its minimum cost spanning tree

```
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 $\infty$ 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 $near[j] \neq 0$ 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 $((near[k] \neq 0)$ and $(cost[k, near[k]] > cost[k,j]))$
                then $near[k] := j$;
    }
    return $mincost$;
}
```

Algorithm-11: Algorithm Prim's minimum-cost spanning tree algorithm

Explanation

Consider a graph with 5 vertices (labeled 1 to 5), and the cost (or weight) of the edges between them is given by the adjacency matrix cost[][]. The value ∞ (represented as a large number) is used to indicate no edge between two vertices.



$$\textbf{cost[ ][ ]} = \begin{vmatrix} \infty & 2 & 3 & \infty & 4 \\ 2 & \infty & 1 & 5 & \infty \\ 3 & 1 & \infty & 6 & 7 \\ \infty & 5 & 6 & \infty & 8 \\ 4 & \infty & 7 & 8 & \infty \end{vmatrix}$$

Here, cost[i, j] represents the cost of the edge between vertex i and vertex j. If there is no edge, the cost is set to ∞.

Now assume that the edge (k, l) with the minimum cost in the set of edges E has been found. Let's say k = 1 and l = 2. This means the initial edge of the MST is between vertices 1 and 2, and mincost is set to the cost of this edge, which is cost[1, 2] = 2.

Initialize the near[] Array: // **Line No 12 to Line 15**

1. **near[] Array Setup:** The array near[] is initialized to track the closest vertex for each vertex that is not yet part of the MST. If the edge between vertex i and l has a lower cost than the edge between vertex i and k, near[i] is set to l; otherwise, it's set to k.
2. **Loop through all vertices:** The loop runs through all vertices to set their closest vertex (l or k). In the following, I'll go through each vertex (i = 1 to 5), checking the conditions and updating near[].

   Fr simplicity, let's update the near[] array with the given values:

   - **i = 1:**
     - cost[1, l] = cost[1, 2] = 2
     - cost[1, k] = cost[1, 1] = ∞ (no self-loop)
     - Since cost[1, l] < cost[1, k], set near[1] := 2.
   - **i = 2:**
     - cost[2, l] = cost[2, 2] = ∞ (self-loop, no edge)
     - cost[2, k] = cost[2, 1] = 2
     - Since cost[2, k] < cost[2, l], set near[2] := 1.
   - **i = 3:**
     - cost[3, l] = cost[3, 2] = 1
     - cost[3, k] = cost[3, 1] = 3
     - Since cost[3, l] < cost[3, k], set near[3] := 2.

- ○ **i = 4:**
  - cost[4, l] = cost[4, 2] = 5
  - cost[4, k] = cost[4, 1] = ∞
  - Since cost[4, l] < cost[4, k], set near[4] := 2.
- ○ **i = 5:**
  - cost[5, l] = cost[5, 2] = ∞ (no edge)
  - cost[5, k] = cost[5, 1] = 4
  - Since cost[5, k] < cost[5, l], set near[5] := 1.

3. **Set near[k] and near[l] to 0:** Finally, since k and l are already part of the MST, we set:
   1. near[k] := near[1] := 0
   2. near[l] := near[2] := 0

Final near[] Array:

After initializing the near[] array, it will look like this:

near[] = [0, 0, 2, 2, 1]

****

Explaining the loop for i = 2 to n - 1 // Lines No 16 to 28

Step-by-Step Explanation:

Assumptions:

We are working with a graph where:

- **cost[i, j]** represents the weight of the edge between vertex i and vertex j.
- **near[ ]** is an array that tracks the closest vertex not yet in the MST.
- **t[ ][ ]** stores the edges that are part of the MST.
- **mincost** stores the total cost of the MST

✓ **Initial MST vertices**: Let's assume that the initial minimum edge (k, l) = (1, 2) is selected, and mincost = 2.

✓ **near[ ]** array initialization (**as discussed before):**

for i = 2 to n - 1 Loop:

This loop finds and adds new edges to the MST until all vertices are added.

Step 1: Find the next closest vertex

For each iteration i, we need to find the next closest vertex j that is not yet in the MST.

- **For i = 2:**
  - We need to find the vertex j such that:
    - near[j] != 0 (meaning j is not yet in the MST),
    - cost[j, near[j]] is the minimum among all the remaining edges.

Looking at the current state of near[]:

near[ ] = [0, 0, 2, 2, 1]

- near[3] = 2, cost[3, 2] = 1
- near[4] = 2, cost[4, 2] = 5
- near[5] = 1, cost[5, 1] = 4

The minimum cost is cost[3, 2] = 1. So, **j = 3** is the vertex with the minimum edge cost to add to the MST.

Step 2: Add the edge to the MST

Now, we add the edge (3, 2) to the MST:

- t[2, 1] := 3 // First vertex of the edge
- t[2, 2] := 2 // Second vertex of the edge
- mincost := mincost + cost[3, 2] = 2 + 1 = 3 // Add the edge cost to mincost
- near[3] := 0 // Mark vertex 3 as added to the MST

Step 3: Update the near[] array

Now we need to update the near[] array for all remaining vertices to reflect the new closest connections. We check all the vertices k:

- **For k = 1**: near[1] = 0 (vertex 1 is already in the MST).
- **For k = 2**: near[2] = 0 (vertex 2 is already in the MST).
- **For k = 3**: near[3] = 0 (vertex 3 is already in the MST).
- **For k = 4**: near[4] = 2, cost[4, 2] = 5, cost[4, 3] = 6. Since cost[4, 3] > cost[4, 2], no update is needed.
- **For k = 5**: near[5] = 1, cost[5, 1] = 4, cost[5, 3] = 7. Since cost[5, 3] > cost[5, 1], no update is needed.

The updated near[] array remains:

near[] = [0, 0, 0, 2, 1]

For i = 3:

Repeat the same steps for i = 3:

- **Find the next closest vertex**:
    - The near[] array shows near[4] = 2 and cost[4, 2] = 5, and near[5] = 1 and cost[5, 1] = 4.
    - The minimum cost is cost[5, 1] = 4, so **j = 5**.
- **Add the edge to the MST**:
    - t[3, 1] := 5, t[3, 2] := 1
    - mincost := 3 + 4 = 7
    - near[5] := 0 (mark vertex 5 as added)
- **Update the near[] array**:
    - near[] = [0, 0, 0, 2, 0]

For i = 4:

Repeat the same steps for i = 4:

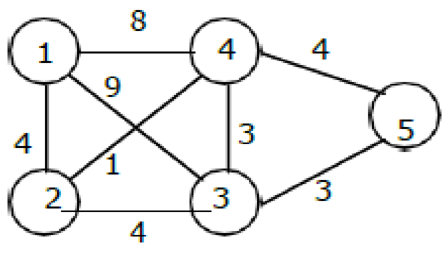
- **Find the next closest vertex**:
    - Now, near[4] = 2 and cost[4, 2] = 5.
    - j = 4.
- **Add the edge to the MST**:
    - t[4, 1] := 4, t[4, 2] := 2
    - mincost := 7 + 5 = 12
    - near[4] := 0 (mark vertex 4 as added)
- **Update the near[] array**:
    - near[] = [0, 0, 0, 0, 0]

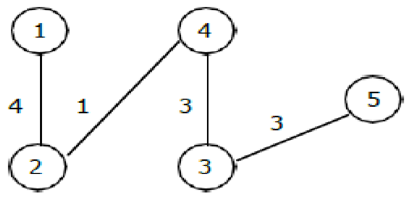
Final MST and Total Cost:

- The final **MST edges** are:
    - (1, 2), (2, 3), (1, 5), (2, 4)
- **Total cost of the MST** is mincost = 12

Example-2:

Considering the following graph, find the minimal spanning tree using prim‟s algorithm.





The minimal spanning tree obtained as:

The cost of Minimal spanning tree = 11.

<u>The steps as per the algorithm are as follows:</u>

Algorithm near (J) = k means, the nearest vertex to J is k.

The algorithm starts by selecting the minimum cost from the graph. The minimum cost edge is (2, 4).

K = 2, l = 4
Min cost = cost (2, 4) = 1

T [1, 1] =2

T [1, 2] =4

for i = 1 to 5	Near matrix	Edges added to min spanning tree:
Begin		T [1, 1] = 2 T [1, 2] = 4
i = 1 is cost (1, 4) < cost (1, 2) 8 < 4, No Than near (1) = 2	$\begin{array}{\|c\|c\|c\|c\|c\|} \hline 2 & & & & \\ \hline \end{array}$ $\begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array}$	
i = 2 is cost (2, 4) < cost (2, 2) 1 <∝, Yes So near [2] = 4	$\begin{array}{\|c\|c\|c\|c\|c\|} \hline 2 & 4 & & & \\ \hline \end{array}$ $\begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array}$	
i = 3 is cost (3, 4) < cost (3, 2) 1 < 4, Yes So near [3] = 4	$\begin{array}{\|c\|c\|c\|c\|c\|} \hline 2 & 4 & 4 & & \\ \hline \end{array}$ $\begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array}$	
i = 4 is cost (4, 4) < cost (4, 2) ∝< 1, no So near [4] = 2	$\begin{array}{\|c\|c\|c\|c\|c\|} \hline 2 & 4 & 4 & 2 & \\ \hline \end{array}$ $\begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array}$	
i = 5 is cost (5, 4) < cost (5, 2) 4 <∞, yes So near [5] = 4	$\begin{array}{\|c\|c\|c\|c\|c\|} \hline 2 & 4 & 4 & 2 & 4 \\ \hline \end{array}$ $\begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array}$	
end near [k] = near [l] = 0 near [2] = near[4] = 0	$\begin{array}{\|c\|c\|c\|c\|c\|} \hline 2 & 0 & 4 & 0 & 4 \\ \hline \end{array}$ $\begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array}$	

Explaining the loop for i = 2 to n - 1 // Lines No 16 to 28

for i = 2 to n-1 (4) do

i = 2

for j = 1 to 5
j = 1
near(1)≠0 and cost(1, near(1))
2 ≠0 and cost (1, 2) = 4

j = 2
near (2) = 0

j = 3
is near (3) ≠0
4 ≠0 and cost (3, 4) = 3

j = 4
near (4) = 0

J = 5
Is near (5) ≠0
4 ≠0 and cost (4, 5) = 4

select the min cost from the above obtained costs, which is 3 and corresponding J = 3

min cost = 1 + cost(3, 4)
= 1 + 3 = 4

T (2, 1) = 3
T (2, 2) = 4

2	0	0	0	4
1	2	3	4	5

T (2, 1) = 3
T (2, 2) = 4

Near [j] = 0
i.e. near (3) =0

for (k = 1 to n)

K = 1
is near (1) ≠0, yes
2 ≠0
and cost (1,2) > cost(1, 3)
4 > 9, No

K = 2
Is near (2) ≠0, No

K = 3
Is near (3) ≠0, No

K = 4
Is near (4) ≠0, No

K = 5
Is near (5) ≠0
4 ≠0, yes
and is cost (5, 4) > cost (5, 3)
4 > 3, yes
than near (5) = 3

2	0	0	0	3
1	2	3	4	5

	<u>**i = 3**</u> <u>for (j = 1 to 5)</u> J = 1 is near (1) ≠0 2 ≠0 cost (1, 2) = 4 J = 2 Is near (2) ≠0, No		
	J = 3 Is near (3) ≠0, no Near (3) = 0 J = 4 Is near (4) ≠0, no Near (4) = 0 J = 5 Is near (5) ≠0 Near (5) = 3 ➔3 ≠0, yes And cost (5, 3) = 3 Choosing the min cost from the above obtaining costs which is 3 and corresponding J = 5 Min cost = 4 + cost (5, 3) = 4 + 3 = 7 T (3, 1) = 5 T (3, 2) = 3 Near (J) = 0 ➔near (5) = 0 <u>for (k=1 to 5)</u> k = 1 is near (1) ≠0, yes and cost(1,2) > cost(1,5) 4 >∞, No K = 2 Is near (2) ≠0 no K = 3 Is near (3) ≠0 no K = 4 Is near (4) ≠0 no K = 5 Is near (5) ≠0 no	$\begin{array}{\|c\|c\|c\|c\|c\|} \hline 2 & 0 & 0 & 0 & 0 \\ \hline \end{array}$ $\begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array}$	T (3, 1) = 5 T (3, 2) = 3

i = 4 for J = 1 to 5 J = 1 Is near (1) ≠0 2 ≠0, yes cost (1, 2) = 4 j = 2 is near (2) ≠0, No		
J = 3 Is near (3) ≠0, No Near (3) = 0 J = 4 Is near (4) ≠0, No Near (4) = 0 J = 5 Is near (5) ≠0, No Near (5) = 0 Choosing min cost from the above it is only '4' and corresponding J = 1 Min cost = 7 + cost (1,2) = 7+4 = 11 T (4, 1) = 1 T (4, 2) = 2 Near (J) = 0 ➔Near (1) = 0 for (k = 1 to 5) K = 1 Is near (1) ≠0, No K = 2 Is near (2) ≠0, No K = 3 Is near (3) ≠0, No K = 4 Is near (4) ≠0, No K = 5 Is near (5) ≠0, No End.	0 | 0 | 0 | 0 | 0 1 2 3 4 5	T (4, 1) = 1 T (4, 2) = 2

Time complexity

- The time required by algorithm Prim is $O(n^2)$, where n is the number of Vertices in the graph G. To see this, note that line 9 takes $O(|E|)$ time and line10 takes $O(1)$ time.
- The for loop of line12 takes $O(n)$time. Lines18 and 19 and the for loop of line 23 require $O(n)$ time. So, each iteration of the for loop of line 16 takes $O(n)$ time.
- The total time for the for loop of line16 is therefore $O(n^2)$. Hence, Prim runs in **$O(n^2)$time.**

The **time complexity** of **Prim's algorithm** depends on the data structures used for managing the priority queue and how the graph is represented. Let's break down the time complexity for different implementations:

Using Adjacency Matrix + Simple Array for Priority Queue

In this case, the graph is represented using an **adjacency matrix**, and we use a simple array (or linear search) to find the vertex with the smallest key value during each iteration.

Steps:

- **Finding the minimum key vertex**: In each iteration, we need to find the vertex with the smallest key value (key[]). This requires scanning through all n vertices, which takes **O(n)** time.
- **Updating the key values**: For each vertex, we need to update the key values for all its neighbors. This takes **O(n)** time for each vertex (since we are checking all n vertices for neighbors in an adjacency matrix).
- **Total operations**: There are n vertices to process, so the time complexity for finding the minimum key vertex and updating the keys is **O(n)** for each iteration. Since there are n iterations, the total time complexity is: $O(n^2)$

Conclusion: Using an **adjacency matrix** and a **simple array** gives a time complexity of **O(n^2)**.

Kruskal's Algorithm:

NOTE: Refer the Kruskal's Algorithm: concept In Data Structures. How a minimum-cost spanning tree builds using Kruskal's Algorithm

- **Example**: Consider the graph of Figure-4(a).We begin with no edges selected Figure-5-a shows the current graph with no edges selected .
- Edge(1,6)is the first edge considered. It is included in the spanning tree being built. This yields the graph of Figure-5(b).
- Next, the edge(3,4) is selected and included in the tree (Figure-5(c)). The next edge to be considered is (2,7). Its inclusion in the tree being built does not create a cycle, so we get the graph of Figure-5(d).
- Edge(2,3)is considered next and included in the treeFigure-5(e). Of the edges not yet considered, (7,4) has the least cost. It is considered next. Its inclusion in the tree results in a cycle, so this edge is discarded.
- Edge(5,4) is the next edge to be added to the tree being built. This results in the configuration of Figure-5(f).
- The next edge to be considered is the edge(7,5).It is discarded, as its inclusion creates a cycle. Finally, edge(6,5) is considered and included in the tree being built.
- This Completes the spanning tree. The resulting tree(Figure-4(b))has cost 99.

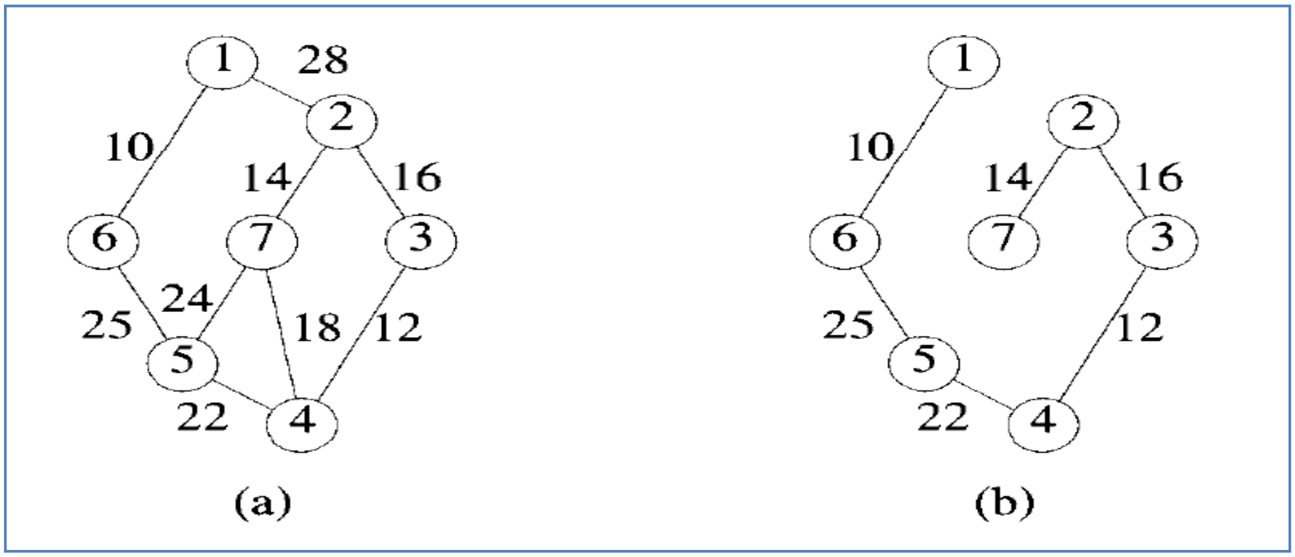


Figure-4: A graph and its minimum cost spanning tree

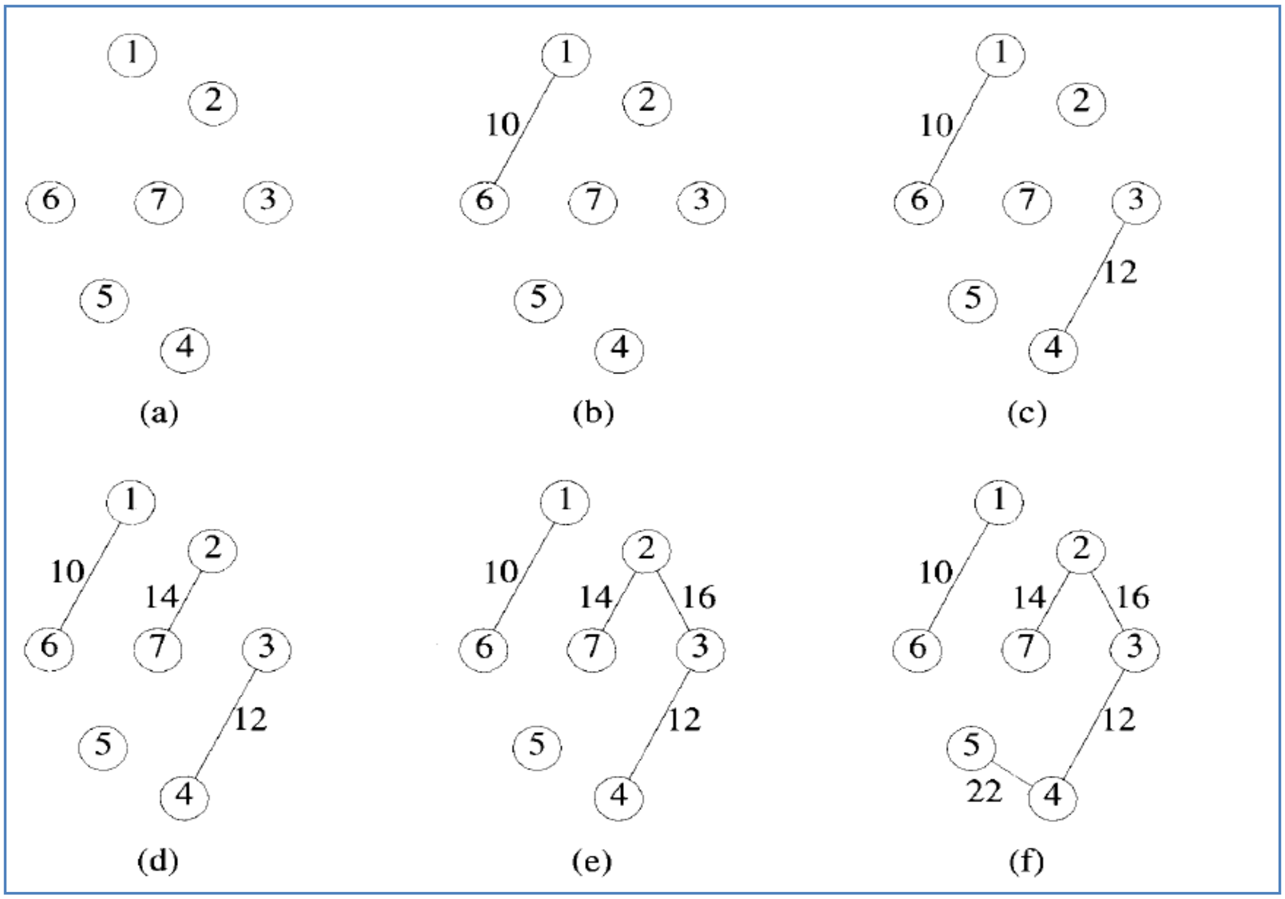


Figure-5: Stages in Kruskal's algorithm

- In **Algorithm -12** **line 6** an initial heap of edges is constructed.
- In **line7** each vertex is assigned to a distinct set (and hence to a distinct tree).
- The set **t** is the set of edges to be included in the minimum-cost spanning tree and **i** is the number of edges in **t**.
- The set **t** can be represented as a sequential list using a two-dimensional array **t[1:n-1 :2].**
- Edge(u,v) can be added to t by the assignments t[i,1]:=u; and t[i,2] :=v;.
- In the while loop of **line10**,edges are removed from the heap one by one in non decreasing order of cost. **Line 14** determines the sets containing u and v. **If j != k,** then vertices u and v are in different sets(and so in different trees)and edge (u,v) is included into t.
- The sets containing u and v are combined (line20). If u = v, the edge(u,v) is discarded as its inclusion into t would create a cycle.
- Line 23 determines whether a spanning tree was found. It follows that **i !=n – 1** iff the graph G is not connected. One can verify that the computing time is **O(E log E)** , where E is the edge set of G.

```
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 ($j \neq k$) then
        {
            $i := i + 1$;
            $t[i, 1] := u$; $t[i, 2] := v$;
            $mincost := mincost + cost[u, v]$;
            Union($j, k$);
        }
    }
    if ($i \neq n - 1$) then write ("No spanning tree");
    else return $mincost$;
}
```

Algorithm-12: Kruskal's algorithm

Time Complexity

Steps:

1. **Sort edges**: Sort all edges in the graph by their weights in non-decreasing order.
2. **Union-Find**: Use a disjoint-set (union-find) data structure to manage and merge components.
3. **Add edges**: Iteratively add edges to the MST, ensuring no cycle is formed. If an edge connects two different components, include it; otherwise, skip it.

Union-Find Operations:

- **Find**: Determine which component a vertex belongs to.
- **Union**: Merge two components.

Time Complexity:

- **Sorting the edges**: O(ElogE) where E is the number of edges.
- **Union-Find operations**: O(Eα(V)) where α(V) which grows very slowly and is nearly constant for practical input sizes.

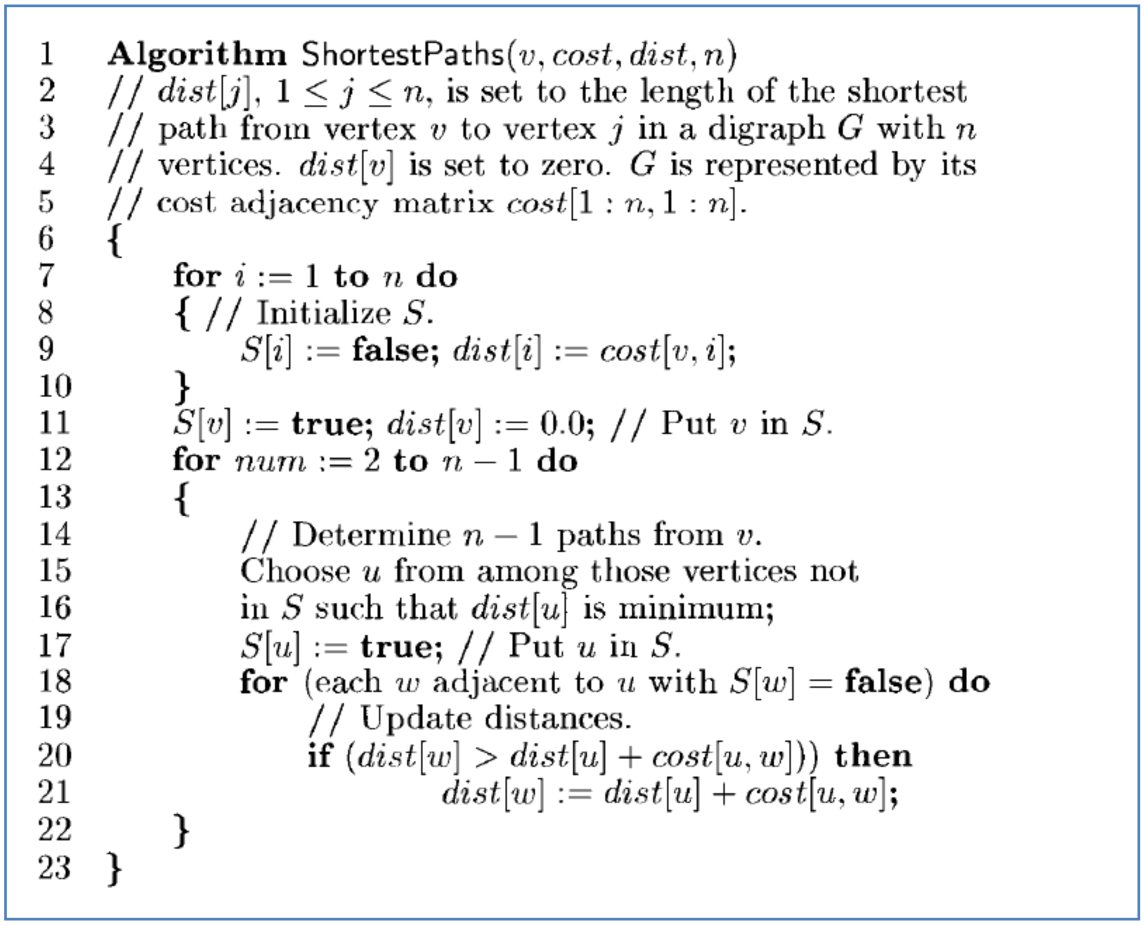
Overall Time Complexity::O(ElogE)

Single source shortest path problem

Dijkstra's Algorithms:

NOTE: Refer the Dijkstra's Algorithm: concept In Data Structures. How to find the shortest path between nodes in a weighted graph using Dijkstra's Algorithm

- A simple **Algorithm-13** for the single source shortest path problem.
- This algorithm (known as **Dijkstra's algorithm**) only determines the lengths of the shortest paths from $\mathbf{v_0}$ to all other vertices in G.
- In the function ShortestPaths(Algorithm-13)it is assumed that the n vertices of G are numbered **1** through n.
- The set S is maintained as a bit array with **S[i]= 0** if vertex **i** is not in S and **S[i]= 1if** it is.
- It is assumed that the graph itself is represented by its cost Adjacency matrix with cost[i, j]'s being the weight of the edge(i,j).
- The weight cost[i,j] is set to some large number, **∞,** in case the edge{i,j) is not in E(G).
- For i = j, cost[i,j] can be set to any non negative number without affecting the outcome of the algorithm



```
Algorithm ShortestPaths($v, cost, dist, n$)
// $dist[j]$, $1 \le j \le n$, 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
                    $dist[w] := dist[u] + cost[u, w]$;
    }
}
```

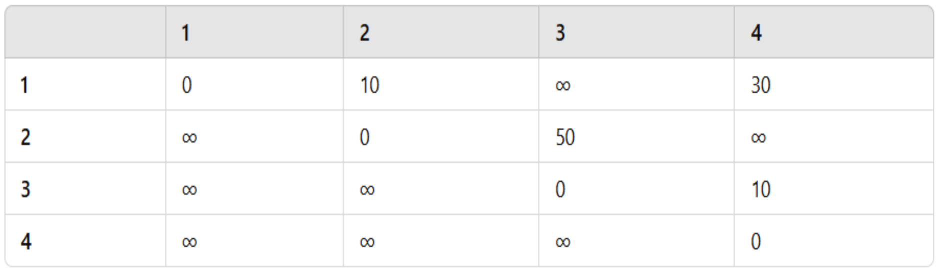
Algorithm-13: Greedy algorithm to generate shortest paths

Pseudocode Walkthrough with Example:

Input:

- A directed graph G with n = 4 vertices.
- The adjacency matrix cost, where cost[i][j] represents the weight of the edge from vertex i to vertex j.

Let's consider the following graph represented by the adjacency matrix:



	1	2	3	4
1	0	10	∞	30
2	∞	0	50	∞
3	∞	∞	0	10
4	∞	∞	∞	0

- Let's say the source vertex is v = 1.

Step-by-step execution:

1. **Initialization**:
    - dist[1] = 0, dist[2] = 10, dist[3] = ∞, dist[4] = 30
    - S[1] = true, S[2] = false, S[3] = false, S[4] = false
2. **First iteration** (num = 2):
    - Vertex u with the minimum dist[u] is 2, because dist[2] = 10 is the smallest of the non-processed vertices.
    - Now, S[2] = true.
    - Update the adjacent vertices of 2. Vertex 3 can be reached with a shorter path: dist[3] = dist[2] + cost[2][3] = 10 + 50 = 60.
3. **Second iteration** (num = 3):
    - Vertex u = 4 has the smallest dist[u] value of 30.
    - Now, S[4] = true.
    - Update the adjacent vertices of 4. Vertex 3 can be reached with a shorter path: dist[3] = dist[4] + cost[4][3] = 30 + 10 = 40.
4. **Third iteration** (num = 4):
    - Vertex u = 3 has the smallest dist[u] value of 40.
    - Now, S[3] = true.
    - There are no further vertices to update.

Final shortest path distances (dist[]):

- dist[1] = 0 (source to itself)
- dist[2] = 10 (direct path from 1 to 2)
- dist[3] = 40 (via vertex 4)
- dist[4] = 30 (direct path from 1 to 4)

Time Complexity

- ❖ The time taken by the algorithm on a graph with **n** vertices is $O(n^2)$.
- ❖ To See this, note that the for loop of line7 in Algorithm -13 takes O(n) time. The for loop of line12 is executed n-2 times.
- ❖ Each execution of this loop Requires O(n) time at lines 15 and 16 to select the next vertex and again at the for loop of line 18 to update dist.
- ❖ So the total time for this loop is **$O(n^2)$.** In case a list t of vertices currently not in s is maintained, then the Number of nodes on this list would at any time be n-num.
- ❖ This would Speed up lines15 and 16 and the for loop of line 18, but the asymptotic Time would remain **$O(n^2)$**

Time Complexity using adjacency matrix:

When the graph is represented as an **adjacency matrix**, Dijkstra's algorithm involves finding the minimum distance vertex by scanning all vertices in each iteration, and updating distances for all vertices adjacent to the current vertex.

Steps:

- **Finding the minimum distance vertex**: In each iteration, we need to find the vertex u with the smallest tentative distance (dist[u]). This can be done by scanning all n vertices, which takes **O(n)** time for each of the n iterations.
- **Updating distances**: For each vertex u, we update the distances for all its neighbors. In the case of an adjacency matrix, we check all n vertices for neighbors. This also takes **O(n)** time for each vertex.

Time Complexity Breakdown:

- **Finding the minimum vertex**: Takes **O(n)** time for each iteration (total of n iterations).
- **Updating distances**: Takes **O(n)** time for each iteration (total of n iterations).

Therefore, the total time complexity is: **$O(n^2)$**

This is true for **best**, **average**, and **worst** cases when using an adjacency matrix and simple linear search.