**EX - 7: Greedy Approach**

**Aim:** Implementation of different real time problems using Greedy approach.

1. Fractional Knapsack problem
2. Job sequencing with deadlines
3. Minimum spanning trees: Kruskal’s and Prims’s algorithms
4. Single Source Shortest path algorithm: Dijkstra’s algorithm

**Program Description:**

**Greedy algorithms** are a class of algorithms that make **locally optimal**choices at each step with the hope of finding a **global optimum** solution. In these algorithms, decisions are made based on the information available at the current moment without considering the consequences of these decisions in the future. The key idea is to select the best possible choice at each step, leading to a solution that may not always be the most optimal but is often good enough for many problems.

**Working of Greedy Algorithm:**

Divide and Conquer Algorithm can be divided into four steps:

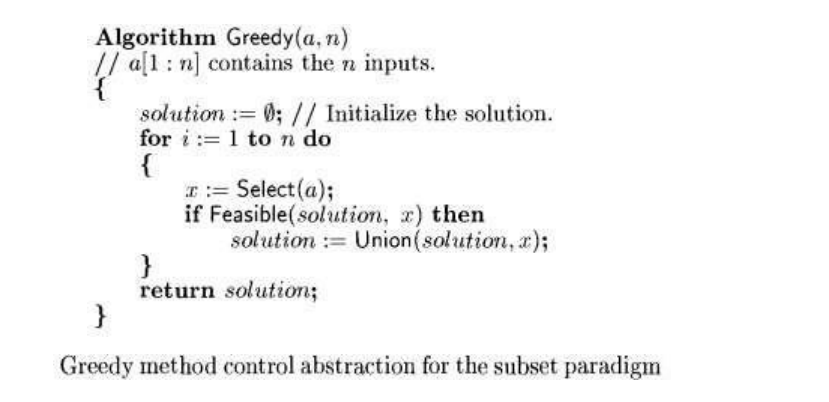
**1.Define the problem:** Clearly state the problem to be solved and the objective to be optimized.

**2. Identify the greedy choice:**Determine the locally optimal choice at each step based on the current state.

**3. Make the greedy choice:** Select the greedy choice and update the current state.

**4. Repeat:** Continue making greedy choices until a solution is reached.

**General Algorithm specification of Greedy approach:**

****

**Applications of Greedy approach;**

1. **Implementation of Fractional Knapsack problem:**

The fractional knapsack problem is also one of the techniques which are used to solve the knapsack problem. In fractional knapsack, the items are broken in order to maximize the profit. The problem in which we break the item is known as a Fractional knapsack problem.

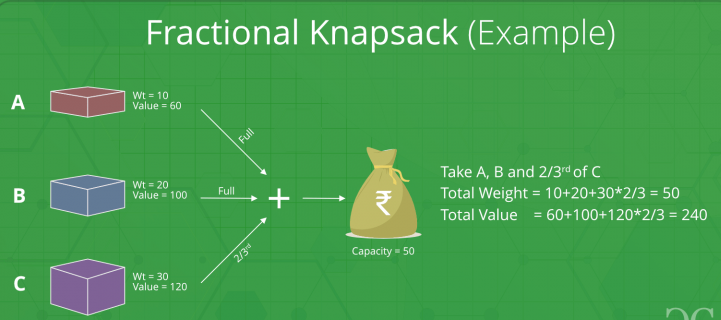
In this, given weights and values of **n** items, we need to put these items in a knapsack of capacity **w** to get the **maximum** total value in the knapsack. Return a double value representing the maximum value in knapsack.

This problem can be solved with the help of using two techniques:

* Brute-force approach: The brute-force approach tries all the possible solutions with all the different fractions but it is a time-consuming approach.
* Greedy approach: In Greedy approach, we calculate the ratio of profit/weight, and accordingly, we will select the item. The item with the highest ratio would be selected first.

There are basically three approaches to solve the problem:

* The first approach is to select the item based on the maximum profit.
* The second approach is to select the item based on the minimum weight.
* The third approach is to calculate the ratio of profit/weight.

****

Program Code:

Input and Output:

1. **Job scheduling with deadline:**

Job scheduling algorithm is applied to schedule the jobs on a single processor to maximize the profits. The greedy approach of the job scheduling algorithm states that, “Given ‘n’ number of jobs with a starting time and ending time, they need to be scheduled in such a way that maximum profit is received within the maximum deadline”.

**Job Scheduling Algorithm:**

Set of jobs with deadlines and profits are taken as an input with the job scheduling algorithm and scheduled subset of jobs with maximum profit are obtained as the final

output.

Step1 − Find the maximum deadline value from the input set of jobs.

Step2 − Once, the deadline is decided, arrange the jobs in descending order of their

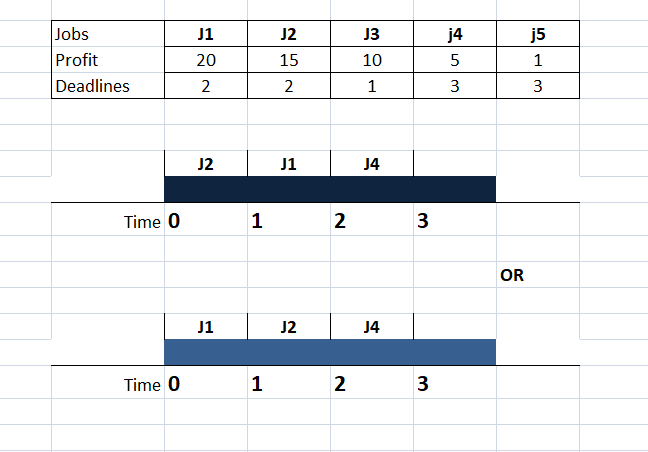
profits.

Step3 − Selects the jobs with highest profits, their time periods not exceeding the

maximum deadline.

Step4 − The selected set of jobs are the output.

Example:



Program Code:

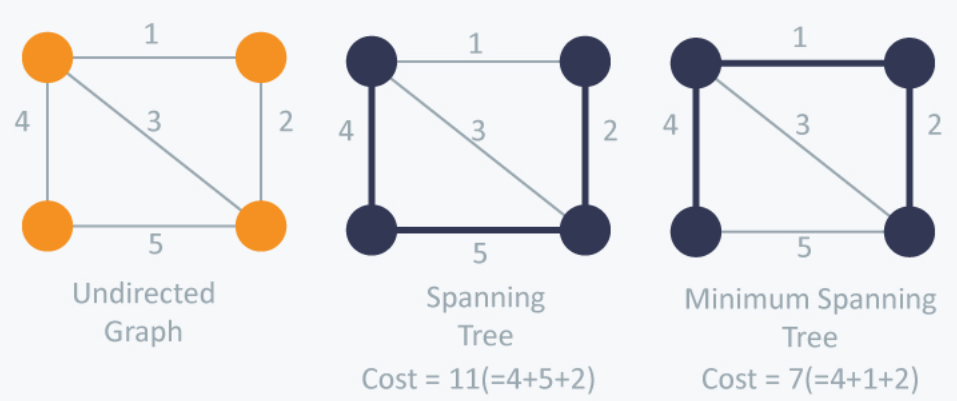
Input and Output:

1. **Minimum cost spanning trees algorithms:**

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.

A minimum spanning tree (MST) is defined as a spanning tree that has the minimum weight among all the possible spanning trees of a graph.

Example:



There are two famous algorithms for finding the Minimum Cost Spanning Tree:

* 1. Kruskal’s algorithm
  2. Prim’s algorithm

1.Kruskal’s minimum cost spanning tree using greedy approach:

Kruskal’s Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

**Algorithm Steps:**

* Sort the graph edges with respect to their weights.
* Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
* Only add edges which doesn't form a cycle , edges which connect only disconnected components.

Example:



Program code: Implement Kruskal’s algorithm using find and union data structure.

Input and Output:

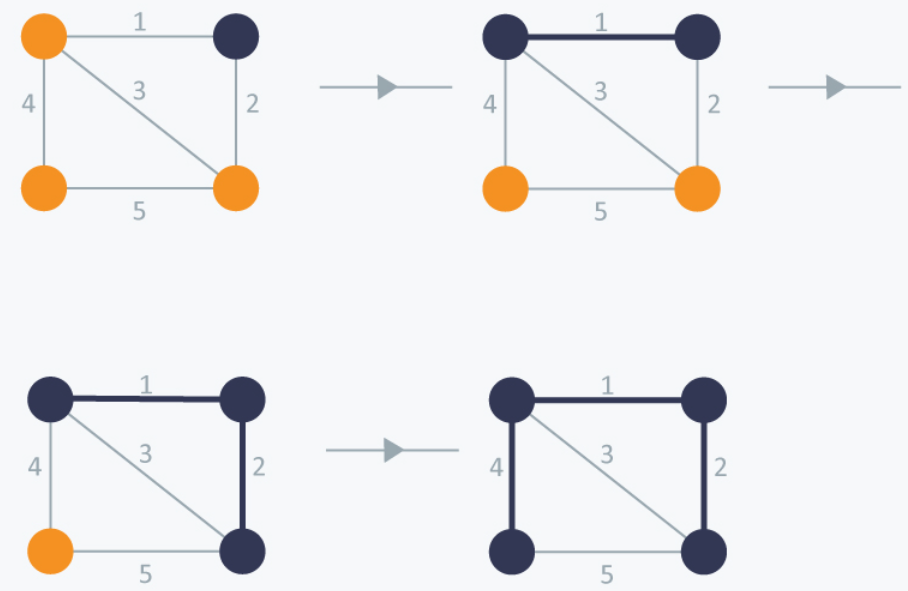
2. **Prim’s minimum cost spanning tree using greedy approach:**

Prim’s Algorithm also use Greedy approach to find the minimum spanning tree. In Prim’s Algorithm we grow the spanning tree from a starting position. Unlike an **edge** in Kruskal's, we add **vertex** to the growing spanning tree in Prim's.

**Algorithm Steps:**

* Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
* Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices, that are connected to growing spanning tree, into the Priority Queue.
* Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.

Example:



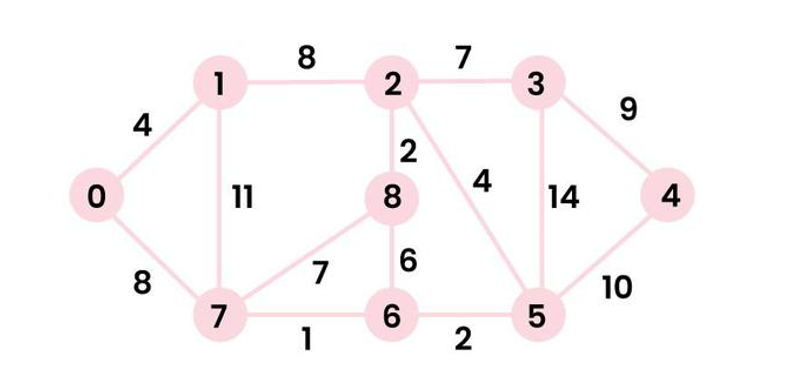
Program code:

Input and output:

4. **Single Source Shortest path algorithm: Dijkstra’s algorithm:**

The Single-Source Shortest Path (SSSP) problem consists of finding the shortest paths between a given vertex v and all other vertices in the graph.

Given a weighted graph and a source vertex in the graph is ‘0’, the **shortest paths** from the source to all the other vertices in the given graph.



***Output:****0 4 12 19 21 11 9 8 14****Explanation:****The distance from 0 to 1 = 4.  
The minimum distance from 0 to 2 = 12. 0->1->2  
The minimum distance from 0 to 3 = 19. 0->1->2->3  
The minimum distance from 0 to 4 = 21. 0->7->6->5->4  
The minimum distance from 0 to 5 = 11. 0->7->6->5  
The minimum distance from 0 to 6 = 9. 0->7->6  
The minimum distance from 0 to 7 = 8. 0->7  
The minimum distance from 0 to 8 = 14. 0->1->2->8*

Since the **shortest path** from vertex s to vertex t is then defined as any path p with weight w (p) = δ(s,t).



The Dijkstra is an iterative algorithm that finds the shortest path from a particular origin node to all other nodes in a graph. It follows greedy approach.

1. Dijkstra's Algorithm begins at the node we select (the source node), and it examines the graph to find the shortest path between that node and all the other nodes in the graph.
2. The Algorithm keeps records of the presently acknowledged shortest distance from each node to the source node, and it updates these values if it finds any shorter path.
3. Once the Algorithm has retrieved the shortest path between the source and another node, that node is marked as 'visited' and included in the path.
4. The procedure continues until all the nodes in the graph have been included in the path. In this manner, we have a path connecting the source node to all other nodes, following the shortest possible path to reach each node.

Dijkstra's Algorithm updates the new weights based on the **Principle of Relaxation.**

d[v]=2

U

V

0

2

Cost[u,v]=4

If (d[u]+c[u,v] <d[v]

d[v]=d[u]+c[u,v]

**Pseudo code for Dijkstra algorithm:**

    // Initialize distances to all nodes as infinity, except for the source node.  
 **1. distances = map infinity to all nodes**  
**2. distances = 0**

    // Initialize an empty set of visited nodes and a priority queue to keep track of the nodes to

visit.  
   3. visited = empty set  
  **4. queue = new PriorityQueue()**  
**5. queue.enqueue(source, 0)**

    // Loop until all nodes have been visited.  
  **6. while queue is not empty:**  
       // Dequeue the node with the smallest distance from the priority queue.  
     **current = queue.dequeue()**

        // If the node has already been visited, skip it.  
      **if current in visited:**  
**continue**

        // Mark the node as visited.  
    7. **visited.add(current)**

       // Check all neighboring nodes to see if their distances need to be updated.  
    8. **for neighbor in Graph.neighbors(current):**  
           // Calculate the tentative distance to the neighbor through the current node.  
    9. **tentative\_distance = distances[current] + Graph.distance(current, neighbor)**

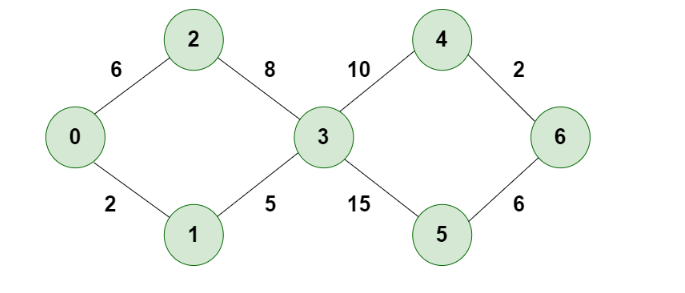
           // If the tentative distance is smaller than the current distance to the neighbor, update the

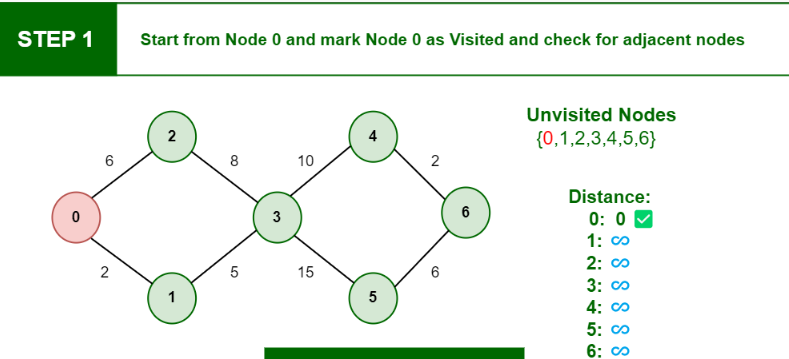
distance.  
       **10. if tentative\_distance < distances[neighbor]:**  
            **distances[neighbor] = tentative\_distance**

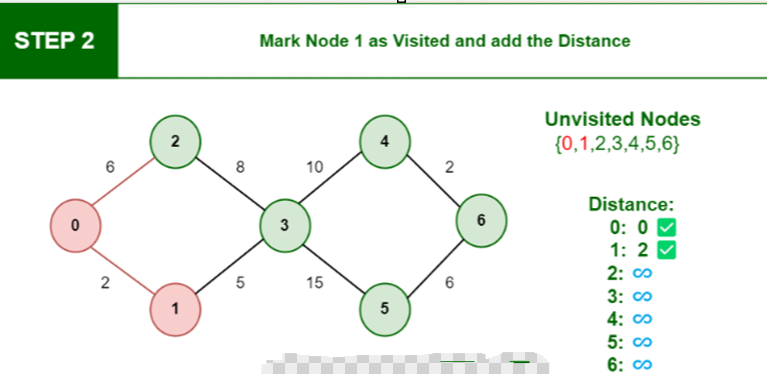
               // Enqueue the neighbor with its new distance to be considered for visitation in the future.  
             **queue.enqueue(neighbor, distances[neighbor])**

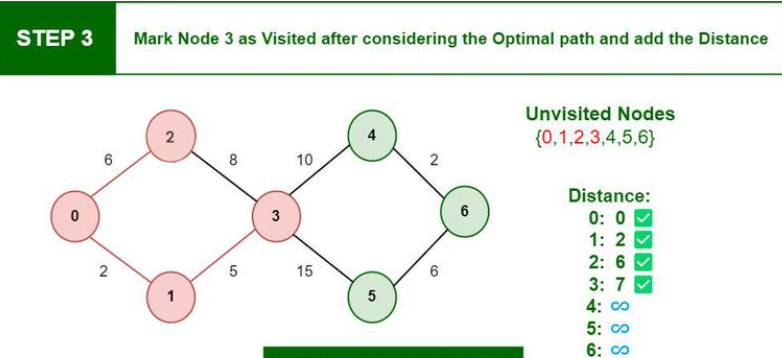
   // Return the calculated distances from the source to all other nodes in the graph.  
  **return distances**

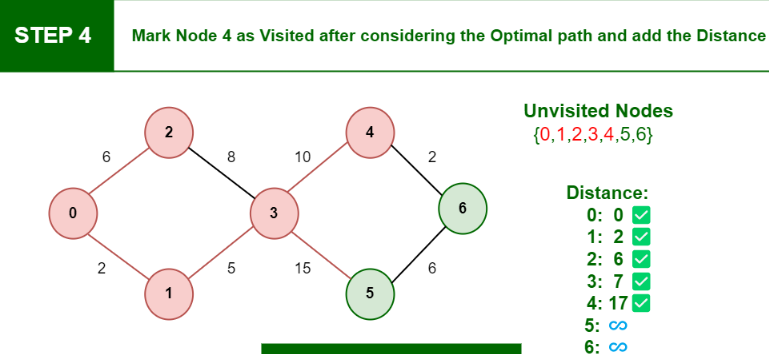
**Example:** Consider the given graph with source vertex ‘0’.

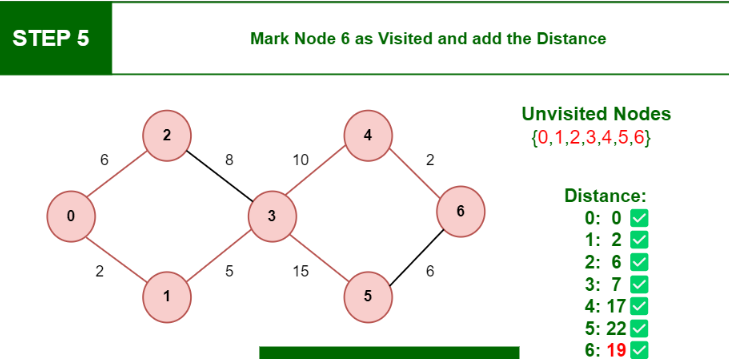












**Applications of Dijkstra’s Algorithm**

* To determine the quickest route
* In applications for social networking
* Within a phone network
* To locate the places on the map

**Complexity of Dijkstra’s Algorithm**

* The time complexity is: O(E Log V).
* Where E represents the edges and V indicates the vertices.
* The space complexity is: O(V)

The problem with negative weights arises from the fact that Dijkstra’s algorithm assumes that once a node is added to the set of visited nodes, its distance is finalized and will not change. However, in the presence of negative weights, this assumption can lead to incorrect results.

Program code:

Input and output: