2.	In Prim's algorithm the next node is always selected from previously selected node.	In Kruskal's algorithm the minimum weight edge is selected independent of earlier selection.
3.	In Prim's algorithm graph must be connected.	The Kruskal's algorithm can work with disconnected graph.
4.	The time complexity of Prim's algorithm is $O(V^2)$	The time complexity of Kruskal's algorithm is O(logV).

Example of Prim's Algorithm – Refer Q.21, Q.22.

Example of Kruskal's Algorithm – Refer Q.24, Q.25.

Q.21 Explain the Prim's algorithm with the appropriate example.

☞ **[JNTU : Part B, April-11, Marks 7, Nov.-16, Marks 10]**

Ans. : Consider the graph given below :

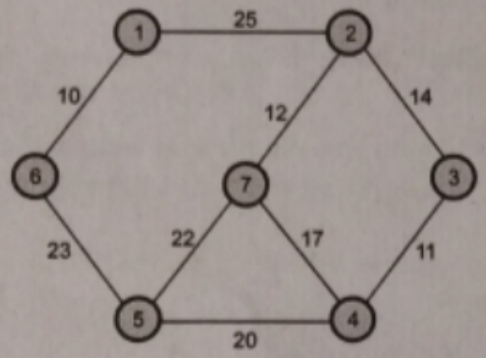
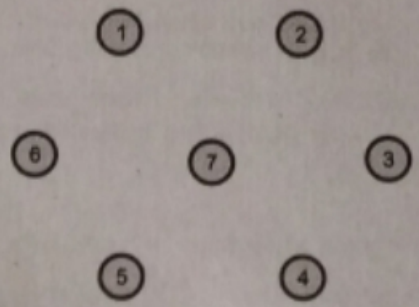


Fig. Q.21.1 Graph

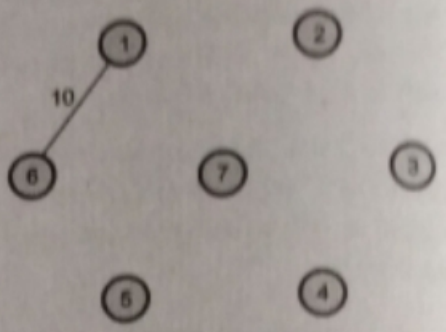
Now, we will consider all the vertices first. Then we will select an edge with minimum weight. The algorithm proceeds by selecting adjacent edges with minimum weight. Care should be taken for not forming circuit.

Step 1 :



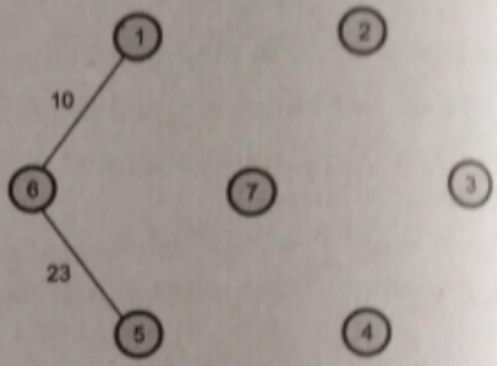
Total weight = 0

Step 2 :



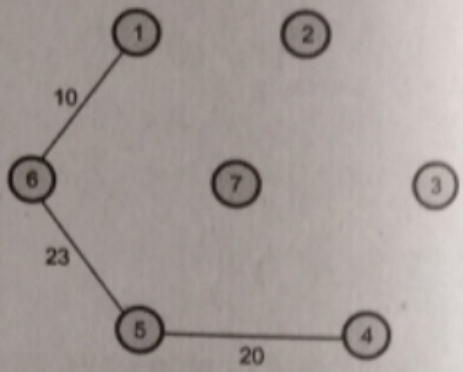
Total weight = 10

Step 3 :



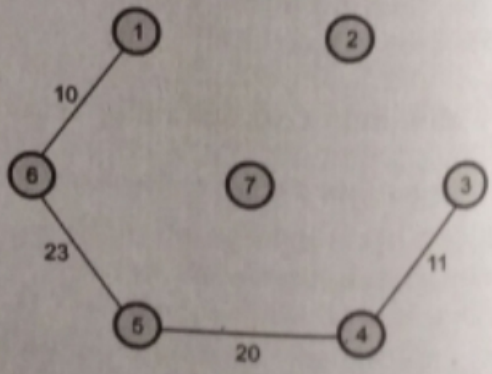
Total weight = 33

Step 4 :



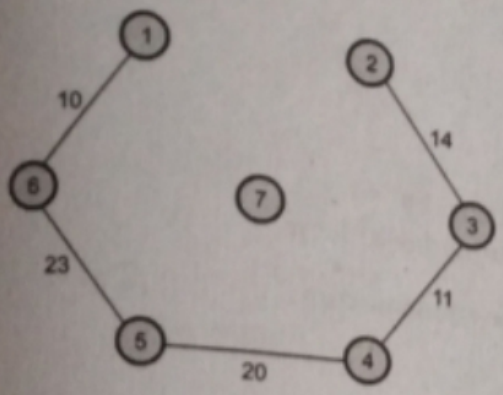
Total weight = 53

Step 5 :



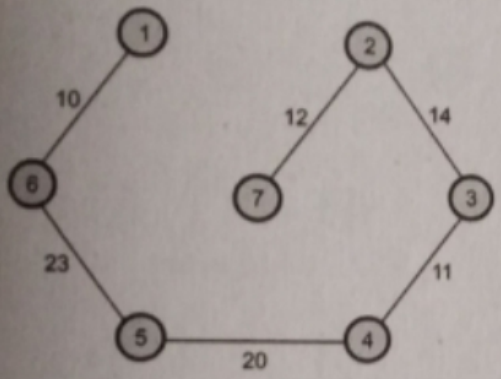
Total weight = 64

Step 6 :



Total weight = 78

Step 7 :



Total weight = 90

Q.22 Write Prim's Algorithm. ☞ [JNTU : Part B, Marks 5]

Ans. :

Algorithm

```
Prim(G[0...Size – 1,0...Size – 1],nodes)
//Problem Description : This algorithm is for
implementing
//Prim's algorithm for finding spanning tree
//Input : Weighted Graph G and total number of nodes
//Output : Spanning tree gets printed with total path
length
   total=0;
  //Initialize the selected vertices list
     for i←0 to nodes–1 do
       tree[i] ← 0
       tree[0] = 1;//take initial vertex
        for k←1 to nodes do
        {
           min_dist ← ∞
//initially assign minimum dist as infinity
for i← 0 to nodes–1 do
{
      for j←0 to nodes–1 do
   {
      if (G [i,j]AND ((tree[i]AND !tree[j])OR
       (!tree[i] AND tree[j])))then
       {
         if(G[i,j] < min_dist)then
                                  Select an edge such that one
                                  vertex is selected and other is not
                                  and the edge has the least weight.
           {                                  Obtained edge with minimum wt.
           min_dist←G[ij]
           v1← i
           v2← j                  Picking up those vertices
                                  yielding minimum edge.
         }
       }
   }
}

Write(v1,v2,min_dist);
    tree[v1] ← tree[v2] ← 1
  total ← total+min_dist
  }
  Write("Total Path Length Is",total)
```

Q.23 Give the time complexity of Prim's Algorithm. ☞ [JNTU : Part B, Marks 5]

Ans. : The algorithm spends most of the time in selecting the edge with minimum length. Hence the **basic operation** of this algorithm is to find the edge with **minimum path length.** This can be given by following formula.

Q.10 Explain job-sequencing with deadlines. Solve the following instance :

n = 5.

$(P_1, P_2, P_3, P_4, P_5) = (20, 15, 10, 5, 1)$

$(d_1, d_2, d_3, d_4, d_5) = (2, 2, 1, 3, 3)$

☞ **[JNTU : Part B, Dec.-12, Marks 8]**

Ans. :

Step 1 : We will arrange the profits P_i in descending order, along with corresponding deadlines.

Profit	20	15	10	5	1
Job	P_1	P_2	P_3	P_4	P_5
Deadline	2	2	1	3	3

Step 2 : Create an array J [ ] which stores the jobs. Initially J [ ] will be

1	2	3	4	5
0	0	0	0	0

J [5]

Step 3 : Add i^{th} Job in array J[ ] at index denoted by its deadline D_i.

First job is P_1, its deadline is 2.

Hence insert P_1 in the array J[ ] at 2^{nd} index.

1	2	3	4	5
	P_1			

Step 4 : Next job is P_2 whose deadline is 2 but J[2] is already occupied. We will search for empty location < J[2]. The J[1] is empty. Insert P_2 at J[1].

Step 5 : Next job is P_3 whose deadline is 1. But as J[1] is already occupied discard this job.

Step 6 : Next job is P_4 with deadline 3. Insert P_4 at J[3].

1	2	3	4	5
P_2	P_1	P_4		

Step 7 : Next job is P_5 with deadline 3. But as there is empty slot at < = J[3], we will discard this job.

Step 8 : Thus the optimal sequence which we will obtain will be $P_2 - P_1 - P_4$. The maximum profit will be 40.

First-In-First-Out Branch and Bound (FIFO BB) is a method used to solve the 0/1 Knapsack Problem. In this approach, items are considered in the order they appear, and the exploration of the state space tree follows a First-In-First-Out (FIFO) strategy. The procedure involves the following steps:

1. **Initialization**: Start with an empty knapsack and set the current node to the root node of the state space tree.
2. **Branching**: Expand the current node by creating two child nodes representing the decision to include or exclude the next item.
3. **Feasibility Check**: Check if the weight of the knapsack in each child node does not exceed the knapsack capacity.
4. **Bound Calculation**: Calculate the upper bound for each child node based on the remaining capacity of the knapsack and the potential profit from the remaining items.
5. **Queue Management**: Add the child nodes to a priority queue based on their bounds, ensuring that the node with the lowest bound is explored first.
6. **Exploration**: Select the next node from the priority queue and repeat steps 2-5 until a termination condition is met.
7. **Termination**: Terminate the algorithm when the priority queue becomes empty or when all nodes have been explored. The optimal solution is the node with the highest profit encountered during the exploration.

Now, let's apply the FIFO BB procedure to the given knapsack instance:

Knapsack instance:

- n = 4 (number of items)
- m = 15 (knapsack capacity)
- Profits: (p1, p2, p3, p4) = (10, 10, 12, 18)
- Weights: (w1, w2, w3, w4) = (2, 4, 6, 9)

To draw the portion of the state space tree and find the optimal solution, we'll follow the FIFO BB procedure:

scss Copy code

```
          (0/0)
        /       \
    (10/2)       (0/0)
    /    \       /    \
(20/6) (10/2)  (12/6)  (0/0)
/   \    /   \
(30/11) (20/6) (22/11) (10/2)
```

Explanation:

- Each node is represented as (Profit/Weight) indicating the cumulative profit and weight of items included in the knapsack at that decision point.
- The root node (0/0) represents the initial state with no items included.
- The tree is expanded based on the FIFO strategy, considering items in the order (p1, p2, p3, p4).
- The tree is pruned based on the constraint that the weight should not exceed the knapsack capacity m=15.
- The exploration continues until all nodes are explored or until the termination condition is met.
- The optimal solution is the node with the highest profit encountered during the exploration, which in this case is (30/11), indicating a profit of 30 units and a weight of 11 units. Therefore, the optimal solution includes items 1, 2, and 4.

To draw the portion of the state space tree generated by Limited Discrepancy Branch and Bound (LCBB) for the given knapsack instance, we need to consider the possible combinations of items that can be included in the knapsack within the weight limit M=12. Since the LCBB algorithm explores the state space tree in a systematic way, we'll draw the tree with each node representing a decision point (whether to include or exclude an item) and the branches representing the possible choices.

Let's start by listing the items and their corresponding profits and weights:

- Items: (P1, P2, P5)
- Profits: (12, 10, 5, 9, 3)
- Weights: (w1, w2, w5): (3, 5, 2, 5, 3)
- Knapsack Capacity (M): 12

Now, let's draw the state space tree:

scss Copy code

```
                    (0/0)
                   /     \
              (1/3)       (0/0)
             /     \      /    \
        (3/8)      (1/3) (2/10) (0/0)
       /    \      /    \
  (6/13)  (3/8)  (4/18)  (1/3)
          /    \
      (9/21)  (6/13)
```

Explanation:

- Each node is represented as (Profit/Weight) indicating the cumulative profit and weight of items included in the knapsack at that decision point.
- The root node (0/0) represents the initial state with no items included.
- Each level of the tree represents a decision point where the algorithm chooses whether to include the next item or not.
- The branches represent the choices: including the next item (with the corresponding profit and weight added to the current node) or excluding it (no change in profit or weight).
- The tree is pruned based on the constraint that the weight should not exceed the knapsack capacity M=12.
- The LCBB algorithm would explore this tree, considering feasible solutions and bounding the search based on the profit of the current best solution found.

This tree represents only a portion of the state space explored by LCBB for the given knapsack instance. The algorithm would continue exploring further until it finds the optimal solution or reaches a termination condition.

To solve this problem optimally, we can use the Earliest Deadline First (EDF) algorithm. This algorithm schedules jobs based on their deadlines, completing the jobs with earlier deadlines first. If multiple jobs have the same deadline, they are scheduled based on their profits.

Given the jobs J1...J6, their deadlines, and associated profits:

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

a) To check if all jobs are completed in the optimal schedule, we need to schedule the jobs using the EDF algorithm and then verify if the schedule satisfies all deadlines and maximizes profit.

b) Optimal Schedule for Maximum Profit:

1. Job J4 (Deadline: 2, Profit: 300)
2. Job J6 (Deadline: 2, Profit: 100)
3. Job J3 (Deadline: 3, Profit: 190)
4. Job J2 (Deadline: 3, Profit: 180)
5. Job J5 (Deadline: 4, Profit: 120)
6. Job J1 (Deadline: 5, Profit: 200)

c) Maximum Earned Profit:
Profit = 300 + 100 + 190 + 180 + 120 + 200
Profit = 1090

So, the maximum earned profit is 1090, achieved by scheduling the jobs in the above optimal schedule.

Sure! Let's consider a simplified example of the Traveling Salesperson Problem:

Suppose a salesperson needs to visit four cities: A, B, C, and D. The distances between these cities are as follows:

- Distance from A to B: 10 miles
- Distance from A to C: 15 miles
- Distance from A to D: 20 miles
- Distance from B to C: 25 miles
- Distance from B to D: 30 miles
- Distance from C to D: 35 miles

The task is to find the shortest possible route that visits each city exactly once and returns to the starting city.

One possible solution could be:

1. Start at city A.
2. Visit city B (10 miles).
3. Visit city C (25 miles).
4. Visit city D (35 miles).
5. Return to city A (20 miles).

Total distance traveled = 10 + 25 + 35 + 20 = 90 miles.

However, this solution might not be the optimal one. The optimal solution would involve examining all possible permutations of city visits and selecting the one with the shortest total distance. For this small example, it's feasible to manually calculate all permutations, but for larger instances of the TSP, computational methods are necessary to find the optimal solution efficiently.

The Traveling Salesperson Problem (TSP) is a classic problem in computer science and mathematics that seeks to find the most efficient route for a salesperson to visit a set of cities exactly once and then return to the starting city. The objective is to minimize the total distance traveled.

Formally, given a list of cities and the distances between each pair of cities, the task is to find the shortest possible route that visits each city exactly once and returns to the original city. The problem is considered NP-hard, meaning that there is no known polynomial-time algorithm that can solve all instances of the problem optimally.

The importance of the TSP extends beyond its practical applications in sales and logistics; it serves as a benchmark for optimization algorithms. Many real-world optimization problems can be modeled as variations of the TSP, making it a fundamental problem in combinatorial optimization and operations research.

Efforts to solve the TSP have led to the development of various approximation algorithms, heuristic methods, and exact algorithms that aim to find good solutions within a reasonable amount of time. Some popular approaches include brute-force search, dynamic programming, genetic algorithms, simulated annealing, and ant colony optimization.

Despite the computational challenges it poses, the Traveling Salesperson Problem remains an important area of research due to its relevance to real-world routing and optimization problems and its role in algorithmic development.