Assignment - 5
PROGRAM STRUCTURES AND ALGORITHMS

A) Create 20 Question 5-point Sample Final

1) Given a set of equations and inequalities between variables, devise an algorithm to determine if a solution exists to the set of conditions. Design the algorithm with a time complexity better than O (n log n), where n represents the number of equations and inequalities.

Solution:

Create disjoint sets for each variable, initially pointing to themselves as their parent.

For each equation in the input:

Apply the Union operation to variables in the equation to unify variables with equal values into the same subset.

For each inequality in the input:

Identify the parent of the two variables involved in the inequality.

If both variables share the same parent, the inequality contradicts the previous equation and indicates no solution.

If no contradictions are found:

Traverse all variables and assign their parent's value to each variable to ensure all elements in the same subset have the same value.

Output the values of all variables as the solution, representing the values that satisfy the given equations and inequalities.

Complexity Analysis:

The Union and Find operations have a time complexity of O(log*V), where V is the number of variables.

Overall running time is O(V + n log*V), where n is the number of equations and inequalities.

As the iterated logarithm (log*V) grows significantly slower than logarithmic functions, the algorithm's time complexity remains better than O(n log n) even for large values of n.

2) Define a recurrence relation for the best-case scenario of Quick Sort, where each partitioning splits the array into one empty subarray and one subarray with (n - 1) elements. If solvable using the Master Theorem, apply it to determine the runtime; otherwise, state that the Master Theorem is not applicable.

Solution:

For the best-case scenario of Quick Sort, where each partitioning step splits the array into one empty subarray and one subarray with (n - 1) elements, the recurrence relation is:

$T(n)=T(n-1)+\Theta(n)$

T(n) represents the time complexity for an input of size n.

T(n−1) denotes the time taken to sort an array with (n - 1) elements.

Θ(n) accounts for the time taken for partitioning.

Applying the Master Theorem:

The Master Theorem is not applicable in this case, as the recurrence relation does not conform to the standard form required by the theorem (i.e., T(n)=aT(n/b)+f(n)).

Explanation:

The best-case scenario for Quick Sort, where each partitioning creates one empty subarray and one subarray with (n - 1) elements, results in a recurrence relation where the sorting time for an array of size n is represented by $T(n)=T(n-1)+\Theta(n)$. This means that the algorithm has to sort (n - 1) elements and then partition the array in linear time Θ(n).

Unlike the standard form required by the Master Theorem, this recurrence relation doesn't fit the format T(n)=aT(n/b)+f(n). Therefore, the Master Theorem cannot be directly applied to solve this recurrence relation.

3)

Solve the following recurrence using the Master Theorem:

T(n) = 4T(n/2) + n^2 log n

Solution:

Applying the Master Theorem to the given recurrence,

a = 4, b = 2, and f(n) = n^2 log n.

Comparing f(n) to n^log_b(a),

n^log_b(a) = n^log_2(4) = n^2

Compare f(n) to n^2 log n. Since f(n) is not polynomially smaller or larger than n^2 log n, the Master Theorem doesn't directly apply.

use the regularity condition to check the case:

af(n/b) ≤ kf(n) for some constant k < 1, and all sufficiently large n.

4(f(n/2)) ≤ kf(n)

f(n/2) ≤ (k/4)f(n)

n^2/2 log(n/2) ≤ (k/4)n^2 log n

n^2 log(n/2) ≤ (k/4)n^2 log n

$n^2 \log n - n^2 \leq (k/4)n^2 \log n$

$n^2 \log n \leq (k/4)n^2 \log n + n^2$

Dividing both sides by $n^2 \log n$:

$1 \leq (k/4) + 1/n$

As n approaches infinity, the right-hand side approaches (k/4), so k must be less than 4.

Now, the three cases are:

If $k < 1/4$ (i.e., $k = 0$), then $T(n) = \text{Theta}(n^2 \log n)$.

If $k = 1/4$, then $T(n) = \text{Theta}(n^2 \log n * \log n)$.

If $k > 1/4$, then $T(n) = \text{Theta}(n^2 \log n)$.

4)

Consider the following scenario for scheduling tasks on a single machine. Each task Ti takes ti units of time to complete and has a deadline by which it needs to be finished. The penalty incurred for not meeting a task's deadline is proportional to the time it finishes after the deadline; specifically, it incurs a penalty of pi . (fi - di), where pi is the penalty rate, fi is the finish time, and di is the deadline for task Ti. You're given three tasks with the following details:

task	Time ti	Deadline di	Penalty Rate pi
1	4	3	2
2	2	1	5
3	3	2	3

Answer:

The scheduling strategy that minimizes the total penalty incurred for all tasks in this scenario is the Earliest Deadline First (EDF) strategy.

Explanation:

Earliest Deadline First (EDF) Strategy: This strategy schedules tasks based on their deadlines, aiming to complete tasks with earlier deadlines first.

Given the tasks' completion times (ti), deadlines (di), and penalty rates (pi), using EDF ensures tasks are scheduled in a way that minimizes the penalties incurred for late completion.

Comparison with Other Strategies:

Scheduling based on smallest ti: Prioritizing tasks based on smallest completion times doesn't account for deadlines or penalty rates and might lead to higher penalties for missing deadlines.

Scheduling based on smallest di: Similar to above, scheduling based on smallest deadlines alone doesn't consider penalty rates, potentially resulting in higher penalties for late completions.

Scheduling based on smallest pi: Focusing on smallest penalty rates overlooks deadlines and might not ensure timely task completions, leading to increased penalties.

By employing the EDF strategy, tasks are sequenced to meet their respective deadlines, thus minimizing the total penalty incurred across all tasks by completing them in a manner that prioritizes meeting deadlines.

5) Illustrate a polynomial-time algorithm to detect cycles in a directed graph using Depth-First Search (DFS). Draw an example directed graph and demonstrate the algorithm's execution to detect a cycle.

Solution:

Algorithm for Cycle Detection (DFS):

Start DFS traversal from any unvisited node.

Mark the current node as visited and in the recursion stack.

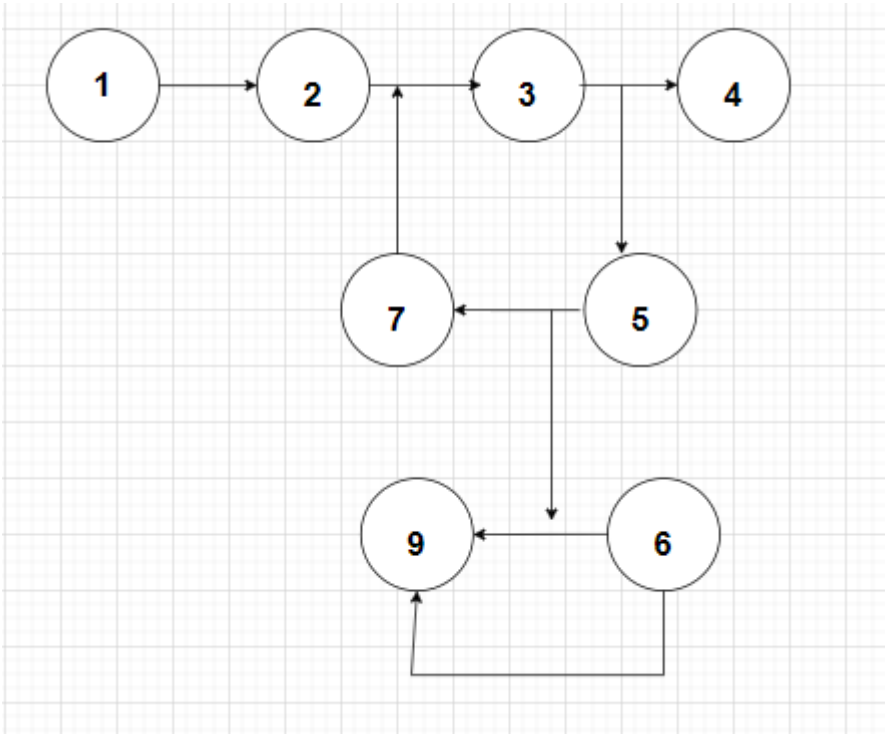
Visit all adjacent vertices.

If an adjacent vertex is visited and in the recursion stack, a cycle is detected.

If an adjacent vertex is visited but not in the recursion stack, continue traversal.

Repeat steps 2-5 until all nodes are visited.

Consider the directed graph below:



Execution of Cycle Detection Algorithm (DFS):

Start DFS from node 1.

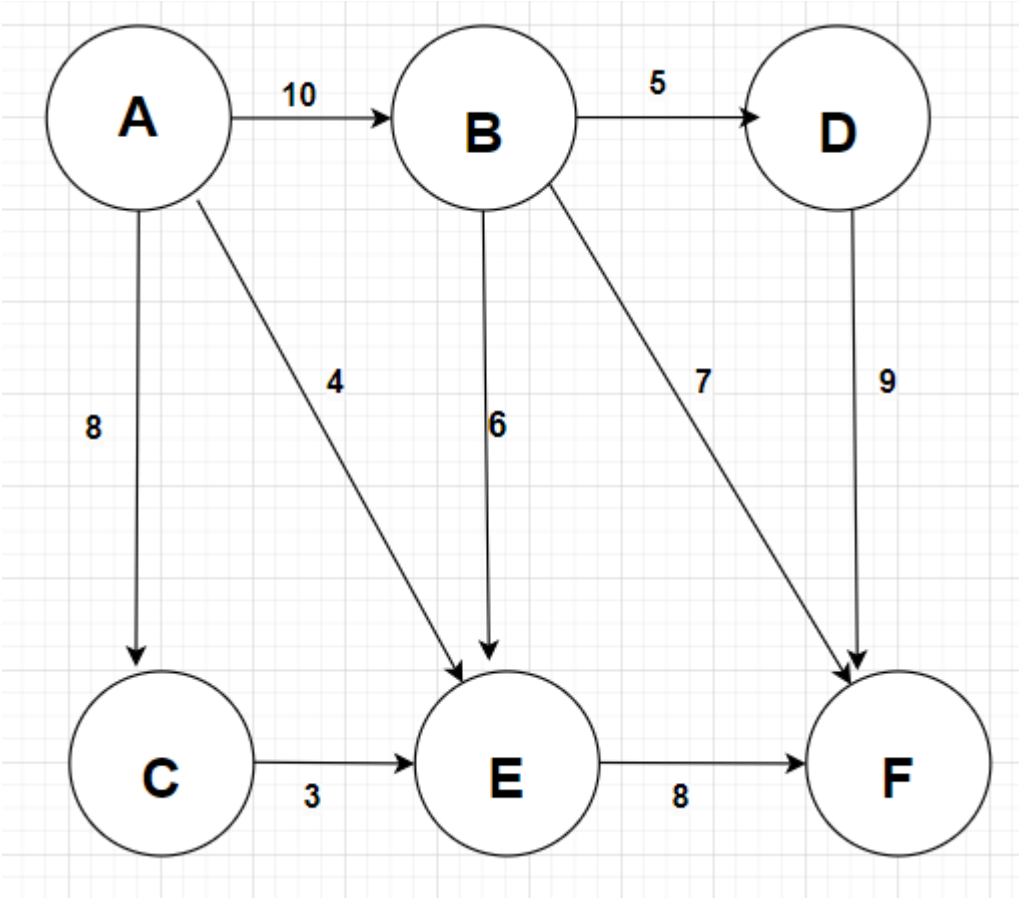
Visit 2, 3, 4, 5, 6, 9.

While visiting 7, 2 is already visited and in the recursion stack, indicating a cycle (1 -> 2 -> 7 -> 5 -> 6 -> 9 -> 7).

The cycle includes nodes 2, 7, 5, 6, and 9, forming a loop in the directed graph.

6)

Consider the following directed graph representing a network flow problem:



Using the Ford-Fulkerson algorithm, find the maximum flow from source node A to sink node F in the given graph.

Solution:

Step 1: Initialization: Initialize flow on each edge as 0.

Step 2: Augmenting Paths: Start with the initial flow.

Find augmenting paths from source A to sink F.

Update the flow along the augmenting paths.

Step 3: Update Residual Graph: Create a residual graph based on the current flow.

Step 4: Repeat Steps 2 and 3: Repeat finding augmenting paths and updating the flow until no more augmenting paths are available.

Example Augmenting Path:

A -> B -> D (Capacity = 5)

A -> C -> E -> F (Capacity = 3)

Flow Updates:

Initially, all edges have zero flow.

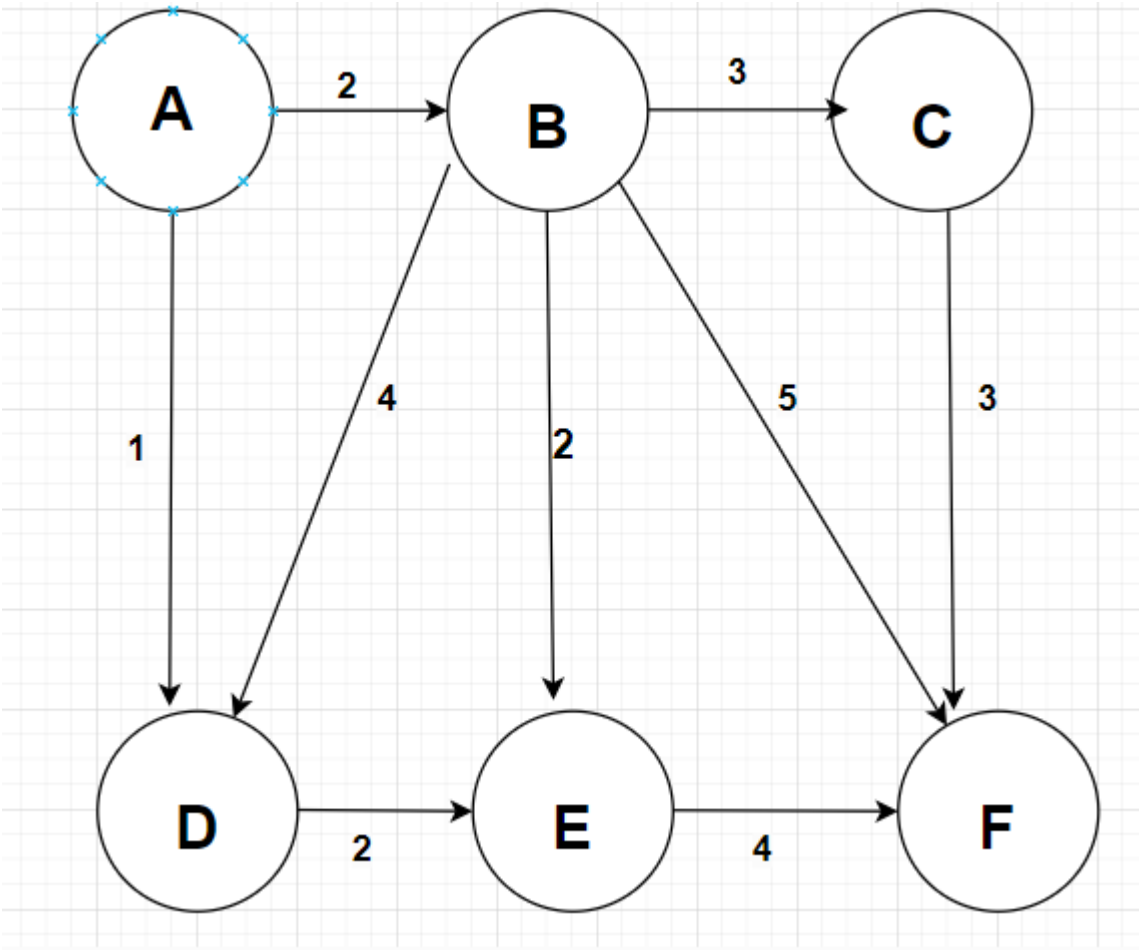
Augmenting path A -> B -> D: Flow = 5 (Capacity = 5).

Augmenting path A -> C -> E -> F: Flow = 3 (Capacity = 3).

Total maximum flow from A to F = 5 + 3 = 8 units.

The maximum flow from source node A to sink node F in the given graph is 8 units using the Ford-Fulkerson algorithm.

7)



Starting at node A, determine the sequence of nodes visited by Dijkstra's algorithm to find the shortest path to all other nodes in the given graph.

Solution:

To find the sequence of nodes visited by Dijkstra's algorithm starting from node A:

Initialization: Set the distance from the source node (A) to itself as 0, and all other nodes' distances as infinity.

Iteration:

Begin from node A and explore its neighboring nodes (B and D).

Update the distances from A to its neighbors (B: 2, D: 1).

Select the node with the shortest distance (D) as the next node to visit.

From node D, explore its neighboring nodes (A, E).

Update the distances from the source node A to these neighbors (A: 1, E: 3).

Select the next unvisited node with the shortest distance (A).

Continue this process until all nodes are visited.

Final Sequence: The correct sequence of nodes visited by Dijkstra's algorithm starting from node A to find the shortest paths to all other nodes is: A, D, B, E, C, F8)

Describe how the Ford-Fulkerson algorithm deals with the problem of finding augmenting paths and updating flow values in a flow network. Provide a detailed example demonstrating the algorithm's execution.

8) What is the primary approach used to demonstrate the NP-completeness of a problem?

Answer:

The primary method employed to establish the NP-completeness of a problem is through a polynomial-time reduction from a known NP-complete problem to the given problem.

To demonstrate NP-completeness, consider the Subset Sum problem known to be NP-complete. This problem asks whether there exists a subset of a set of integers whose sum equals a given target value.

Let's take another problem, the Knapsack problem, and aim to show its NP-completeness by reduction from the Subset Sum problem.

Subset Sum: Given a set of integers and a target value, determine if there's a subset with a sum equal to the target.

Knapsack Problem: Given a set of items, each with a weight and value, determine the maximum value that can be obtained by selecting a subset of items that doesn't exceed a given weight limit.

Reduction:

Reduction Mapping: We'll demonstrate a polynomial-time reduction from Subset Sum to Knapsack by showing that if we can solve Knapsack efficiently, we can solve Subset Sum efficiently.

Transformation: For a Subset Sum instance (S, target), create a Knapsack instance with weights and values equal to the integers in set S and the weight limit set to the target value.

Equivalence: Solving the transformed Knapsack instance would provide a subset of integers whose sum equals the target, which essentially solves the Subset Sum problem.

This reduction illustrates that if Knapsack could be solved efficiently, Subset Sum could also be solved efficiently. Therefore, Knapsack is at least as hard as Subset Sum, establishing its NP-completeness through the polynomial-time reduction.

9)

Given the weights and values of five items in the table below, select a subset of items with the maximum combined value that will fit in a knapsack with a weight limit, W, of 10. Use dynamic programming. Show your work.

ITEM i	VALUE v_i	Weight w_i
1	2	3
2	3	4
3	5	6
4	1	2
5	4	5

Capacity of knapsack W = 10

Select which of the following statements are true:

A. Either {1,4} OR {2,3} is a subset of items with the maximum combined value.

B. The maximum combined value is 9.

C. Item 3 will be in the subset of items with the maximum combined value.

D. The maximum combined value of items 1, 2, and 3 is 10 if the capacity of the knapsack is W = 10.

E. The maximum combined value of items 1 and 2 is 5 if the capacity of the knapsack is W = 10.

Solution:

A. Either {1,4} OR {2,3} is a subset of items with the maximum combined value. (T)

C. Item 3 will be in the subset of items with the maximum combined value. (T)

D. The maximum combined value of items 1, 2, and 3 is 10 if the capacity of the knapsack is W = 10. (T)

E. The maximum combined value of items 1 and 2 is 5 if the capacity of the knapsack is W = 10. (T)

Algorithm: given two arrays w[3, 4, 6, 2, 5] and v[2, 3, 5, 1, 4]:

	0	1	2	3	4	5	6	7	8	9	10
1	0	0	0	2	2	2	2	2	2	2	2
2	0	0	0	3	3	3	3	5	5	5	5
3	0	0	0	3	3	3	5	5	5	5	8
4	0	0	1	3	3	4	5	5	6	6	8
5	0	0	1	3	3	4	5	5	6	7	9

To achieve the maximum value of 9, items {1,4} or {2,3} can be selected.

This demonstrates the process of selecting the items to achieve the maximum combined value within the weight limit of the knapsack using dynamic programming.

10)

For an algorithm with a time complexity of O(N^2), if the input size is doubled from N to 2N, how does the time taken by the algorithm change? Provide an explanation without using multiple-choice options.

Answer:

When the input size for an algorithm with a time complexity of O(N^2) is doubled from N to 2N, the time taken by the algorithm increases by a factor of 4.

Explanation:

The time complexity of O(N^2) implies that the algorithm's runtime grows proportionally to the square of the input size.

When the input size is doubled from N to 2N:

For input size N, the time taken would be proportional to N^2.

For input size 2N, the time taken becomes proportional to (2N)^2 = 4N^2.

The ratio of the time taken for the doubled input size (2N) to the initial input size (N) is: $(2N)^2/N^2 = 4N^2/N^2 = 4$

Therefore, the time taken by the algorithm increases by a factor of 4 when the input size is doubled.

11)

Explain how the Gale-Shapley algorithm handles cases where preferences are not strictly ordered or include ties. Illustrate with an example where ties or incomplete preferences exist, and demonstrate the algorithm's behavior in such scenarios.

Solution:

The Gale-Shapley algorithm can accommodate tied preferences or incomplete preference lists by allowing individuals to propose or be matched to multiple partners during the process.

Example:

Consider three men (M1, M2, M3) and three women (W1, W2, W3) with the following preferences:

M1: [W1, W2]

M2: [W2, W1, W3]

M3: [W3, W2]

W1: [M1, M2]

W2: [M1, M3, M2]

W3: [M2, M3]

In this scenario, there are ties in preferences and an incomplete preference list for M2 and W3.

The algorithm handles this by allowing individuals to propose to multiple partners and women to hold multiple suitors until stable matches are achieved. For instance:

M1 proposes to W1 (preferred) and gets accepted.

M2 proposes to W2 (first choice), gets rejected, proposes to W3, and gets accepted.

M3 proposes to W3 (preferred) and gets accepted.

The resulting stable matching could be: M1-W1, M2-W3, M3-W2. This demonstrates how the algorithm accommodates ties or incomplete preferences by allowing multiple proposals and maintaining stability in the final matching

12)

Discuss the efficiency and time complexity of Kruskal's algorithm for finding the minimum spanning tree (MST) in a graph. Explain the factors influencing its time complexity and scenarios where Kruskal's algorithm performs optimally or sub optimally.

Answer:

Kruskal's algorithm demonstrates efficiency in finding the minimum spanning tree in a graph. Its time complexity is primarily influenced by:

Sorting: Sorting edges based on their weights, which typically takes O(E log E) time, where E is the number of edges.

Disjoint-Set Operations: Utilizing disjoint-set operations to detect cycles and maintain connectivity, which takes nearly O(log V) time per operation on average, where V is the number of vertices.

Iterating Through Sorted Edges: Iterating through sorted edges takes O(E) time.

The overall time complexity is often dominated by sorting and can be expressed as O(E log E) or O(E log V), depending on the number of edges and vertices. Kruskal's algorithm performs optimally for sparse graphs (where E is close to V) but might be less efficient for dense graphs due to the higher number of edges.

Factors influencing its efficiency include the graph's density, the implementation of disjoint-set data structures, and the initial sorting of edges. When edges are already sorted or the graph is sparse, Kruskal's algorithm tends to perform more optimally.

13)

Consider the Preflow-Push algorithm applied to find the maximum flow in a directed graph from node A to node E. Which sequence of steps accurately represents the flow of operations in the algorithm?

A. Initialization -> Push A-B -> Push A-D -> Push D-C -> Relabel C -> Push C-E -> Relabel E

B. Initialization -> Push A-B -> Push A-D -> Push D-C -> Relabel C -> Push C-E -> Relabel E -> Relabel C

C. Initialization -> Push A-B -> Push A-D -> Push D-C -> Relabel C -> Push C-E -> Relabel E -> Push C-E

D. Initialization -> Push A-B -> Push A-D -> Push D-C -> Relabel C -> Push C-E -> Push A-B

Correct Answer: Initialization -> Push A-B -> Push A-D -> Push D-C -> Relabel C -> Push C-E -> Relabel E

Explanation: The Preflow-Push algorithm involves initializing flow values, pushing flows between nodes, relabeling nodes based on heights, and iterating until an optimal flow is achieved. Option A accurately represents the sequence of operations in the algorithm to determine the maximum flow from node A to E in the given graph.

14) You are given an undirected weighted graph with V vertices and E edges. Implement Prim's algorithm to find the Minimum Spanning Tree (MST) of the graph. Provide the steps of the algorithm and the final MST for a given example graph.

Prim's algorithm is a greedy algorithm used to find the Minimum Spanning Tree of a graph. Steps of Prim's algorithm:

Initialize an empty set to represent the MST

Choose an arbitrary starting vertex as the initial node for the MST

Create a priority queue to store edges with their weights

Add all edges connected to the initial node to the priority queue

While the priority queue is not empty and the MST does not contain all vertices:

Extract the edge with the minimum weight from the priority queue

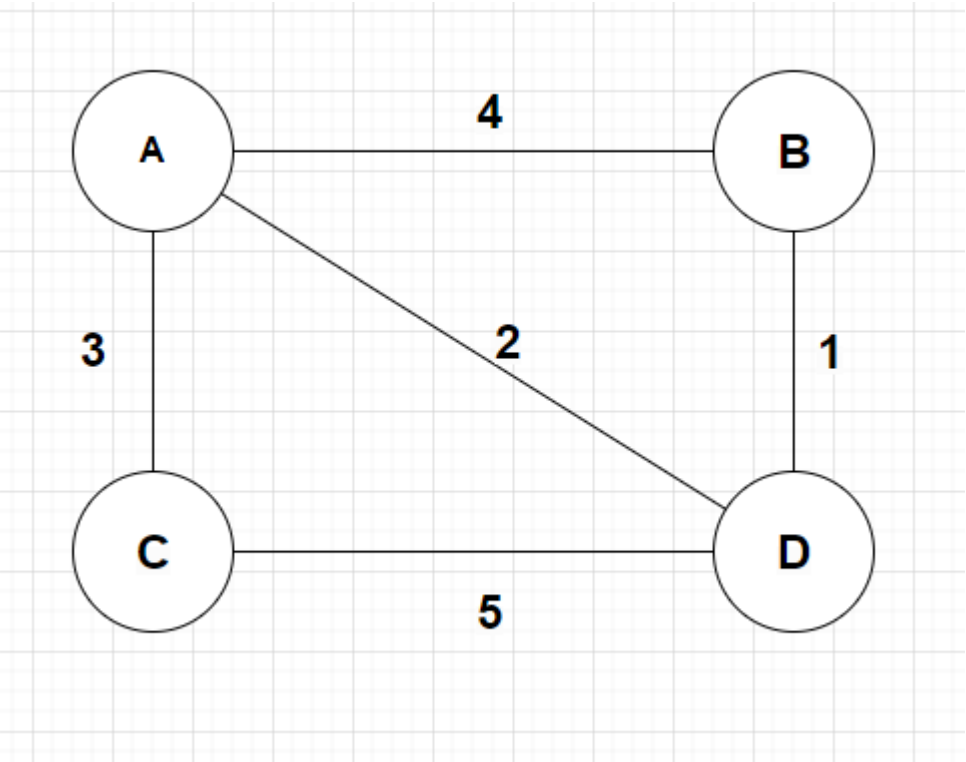
If adding this edge to the MST does not create a cycle, add it to the MST

Add all edges connected to the newly added vertex to the priority queue

The MST is now complete

Prim's algorithm example:

Consider the following weighted, undirected graph:



The graph can be represented as an adjacency matrix:

A B C D

A 0 4 3 2

B 4 0 1 0

C 3 1 0 5

D 2 0 5 0

Prim's algorithm:

1. Initialize an empty MST and start with vertex A.

MST: []

2. Add edges (A, B, 4), (A, C, 3), and (A, D, 2) to the priority queue.

Priority Queue: [(A, B, 4), (A, C, 3), (A, D, 2)]

3. Extract the edge (A, D, 2) with the minimum weight and add it to the MST.

MST: [(A, D, 2)]

4. Add edges (B, D, 1) and (C, D, 5) to the priority queue.

Priority Queue: [(A, B, 4), (A, C, 3), (B, D, 1), (C, D, 5)]

5. Extract the edge (B, D, 1) with the minimum weight and add it to the MST.

MST: [(A, D, 2), (B, D, 1)]

6. Add edge (A, C, 3) to the priority queue.

Priority Queue: [(A, B, 4), (A, C, 3), (C, D, 5)]

7. Extract the edge (B, D, 1) with the minimum weight and add it to the MST.

MST: [(A, D, 2), (B, D, 1), (A, C, 3)]

8. Add edge (A, B, 4) to the priority queue.

Priority Queue: [(A, B, 4), (C, D, 5)]

9. Extract the edge (B, A, 4) with the minimum weight and add it to the MST.

MST: [(A, D, 2), (B, D, 1), (A, C, 3), (A, B, 4)]

10. Extract the edge (C, D, 5) with the minimum weight and add it to the MST.

MST: [(A, D, 2), (B, D, 1), (A, C, 3), (A, B, 4), (C, D, 5)]

The final MST is represented by the edges:

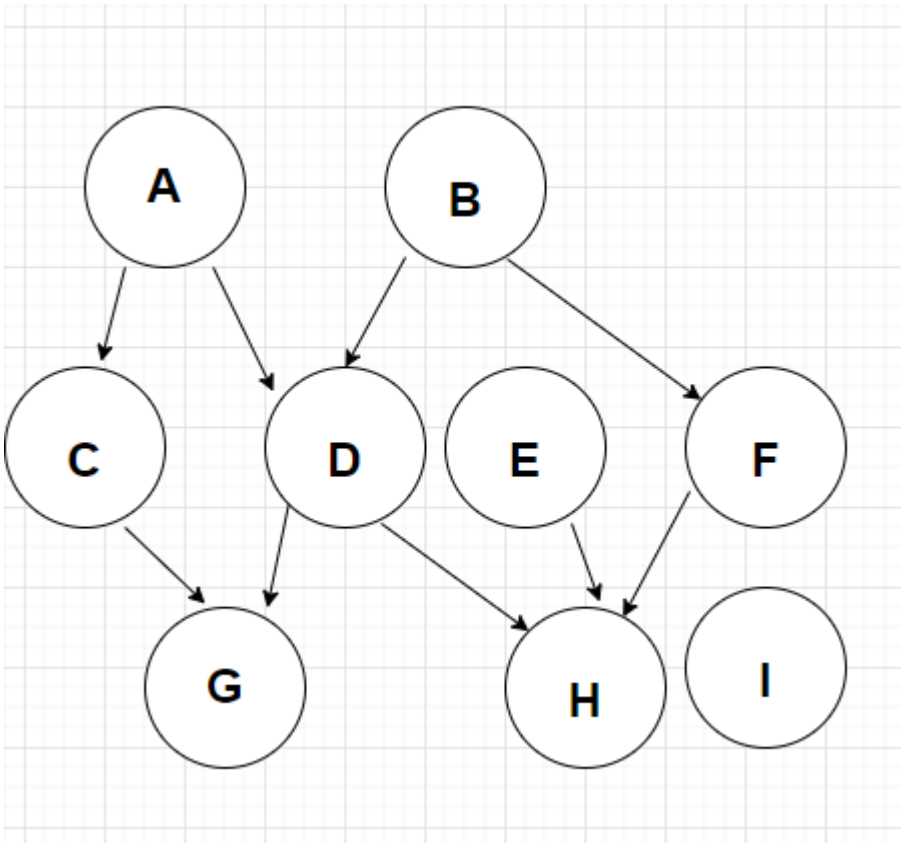
(A, D, 2) (B, D, 1) (A, C, 3) (A, B, 4) (C, D, 5)

This MST contains all the vertices of the original graph with the minimum total weight, and it is a tree with no cycles, which is the objective of Prim's algorithm.

15)

In a directed acyclic graph (DAG), a topological ordering represents a linear ordering of its vertices where for every directed edge from vertex u to vertex v, u comes before v in the ordering.

Given the DAG:



Solution:

valid topological orderings:

A, B, C, D, E, F, G, H, I:

This ordering follows the direction of edges, ensuring that each node appears after all its incoming edges' sources. It starts with A and B, moves to their children, and so on, satisfying the topological sorting rules.

A, B, C, D, E, F, I, G, H:

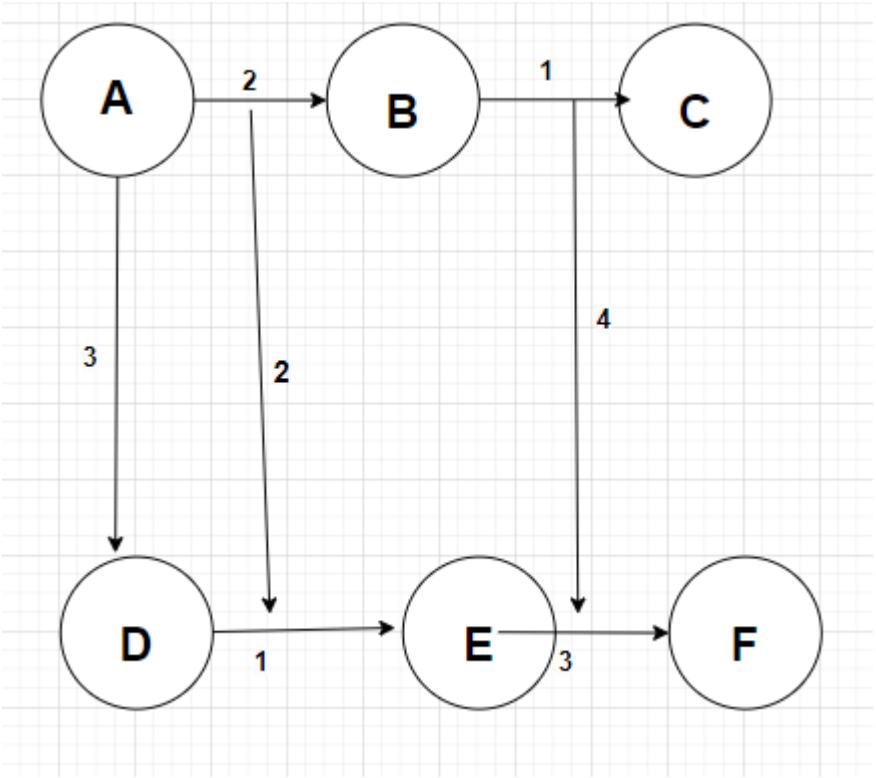
This ordering maintains the topological sequence, starting with A and B, proceeding through C, D, E, F, and then to I before G and H. It also adheres to the rules of topological sorting.

A, B, C, D, E, F, I, H, G:

Similarly, this sequence follows the direction of edges, ensuring that each node comes after all its incoming edges' sources. It starts with A and B, goes through C, D, E, F, I, and then to H before G, satisfying the topological sorting requirements.

These valid topological orderings exhibit different sequences that respect the direction of edges, meeting the criteria of topological sorting in a directed acyclic graph.

16) Use Kruskal's algorithm to find a minimum spanning tree for the connected weighted graph provided below. Show the step-by-step process and the resulting minimum spanning tree.



Kruskal's Algorithm for Minimum Spanning Tree:

A. Create a forest T (a set of trees), where each vertex in the graph is a separate tree.

B. Create a set S containing all the edges in the graph.

C. Sort edges by weight.

While S is nonempty, and T is not yet spanning:

I. Remove an edge with minimum weight from S.

II. If that edge connects two different trees, add it to the forest, combining two trees into a single tree; otherwise, discard that edge.

Steps:

Connect B-D (1).

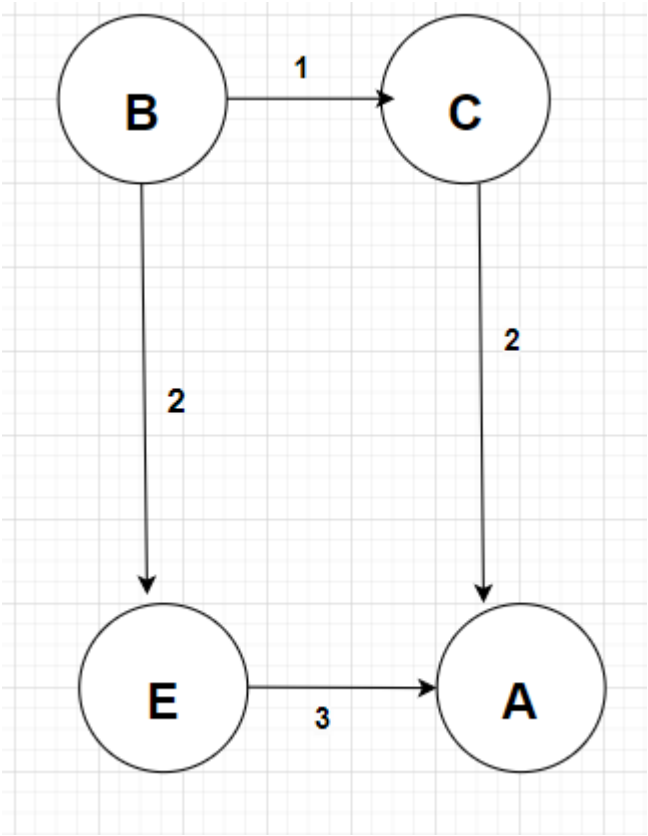
Connect A-E (2).

Connect A-B (2).

Connect C-E (2) or A-C (2) (Note: multiple choices at weight 2).

Connect C-F (3) or E-F (3) (Note: multiple choices at weight 3).

Resulting Minimum Spanning Tree (MST):

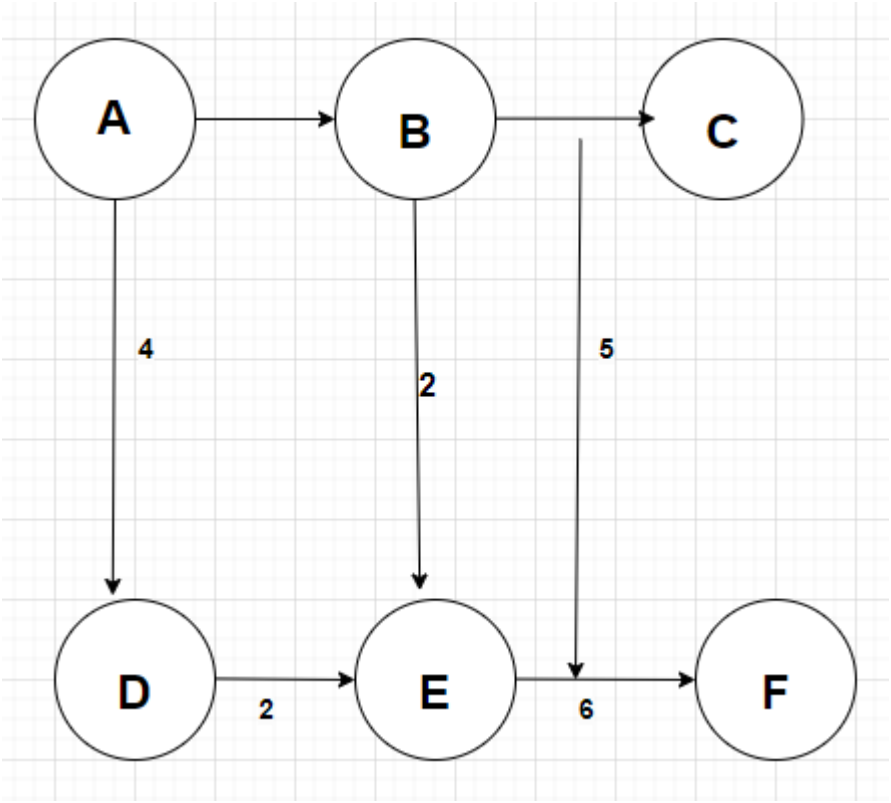


The MST formed contains 5 edges, connecting all 6 vertices in the graph with the least total weight possible.

17)

Apply Prim's algorithm to find the minimum spanning tree for the weighted graph given below. Show each step of the algorithm and the resulting minimum spanning tree.

Weighted Graph:



Solution:

Prim's Algorithm for Minimum Spanning Tree:

Start from an arbitrary vertex (e.g., A) and add it to the minimum spanning tree.

Repeat until all vertices are included:

Choose the edge with the minimum weight that connects the tree to a vertex not in the tree.

Add the vertex and edge to the tree.

Steps:

Start from vertex A.

Connect A-B (1).

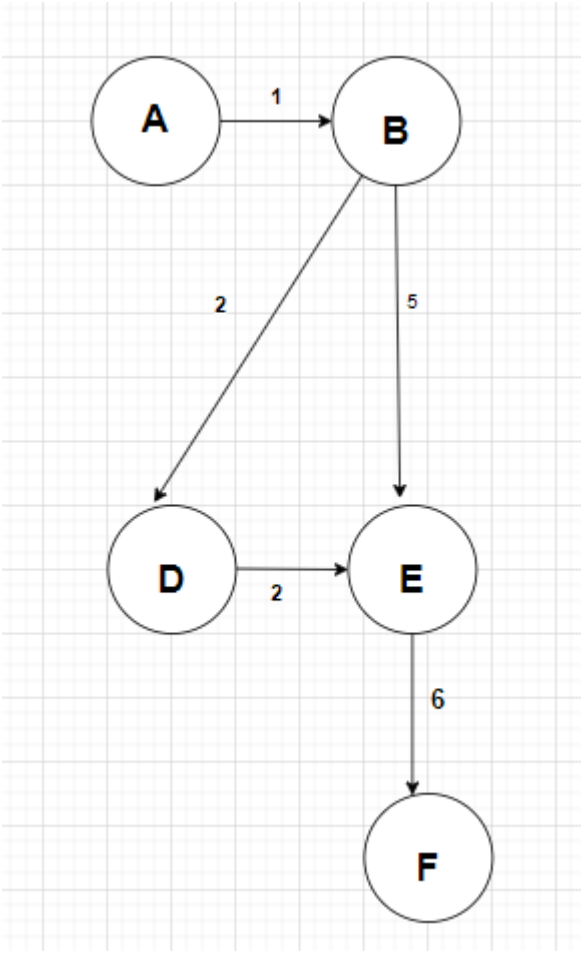
Connect B-E (2).

Connect E-D (4).

Connect E-C (5).

Connect C-F (6).

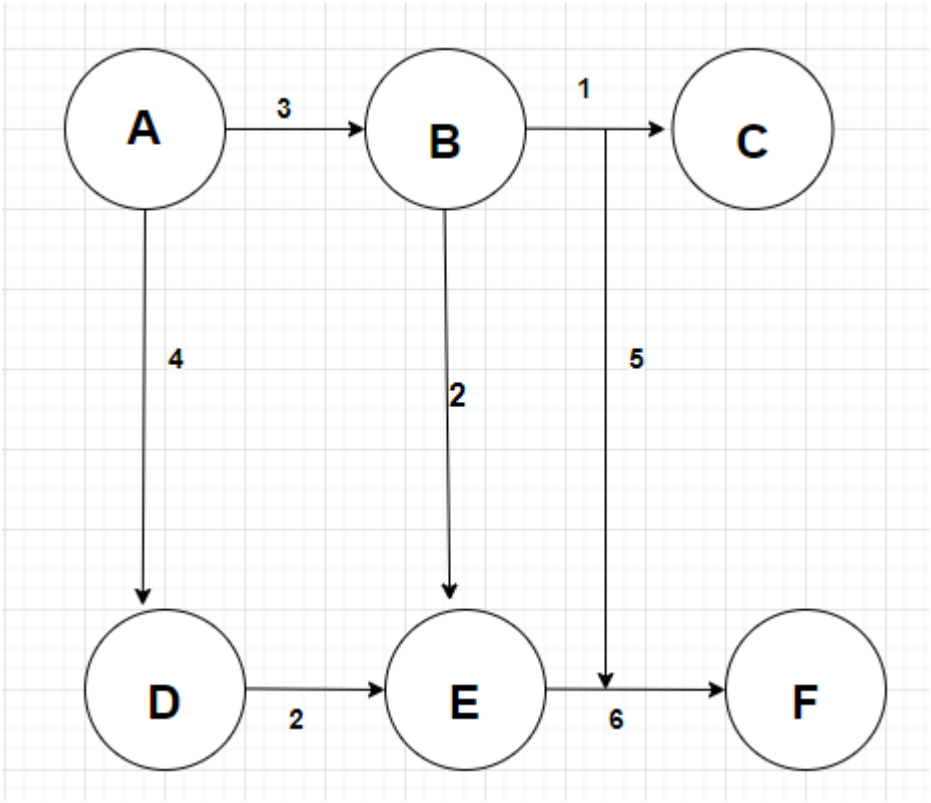
Resulting Minimum Spanning Tree (MST):



18)

Apply Dijkstra's algorithm to find the shortest path from vertex A to all other vertices in the weighted graph below. Show each step of the algorithm and the final shortest paths.

Weighted Graph:



Solution:

Dijkstra's Algorithm for Shortest Path:

Initialize distances from the source (A) to all other vertices as infinity except for A (0).

Repeat until all vertices are visited:

Select the vertex with the minimum distance from the set of unvisited vertices.

Update distances to its neighbors if a shorter path is found.

Steps:

Initialize distances: A: 0, B: ∞, C: ∞, D: ∞, E: ∞, F: ∞.

Choose A and update distances: A-B: 3, A-D: 4.

Choose B and update distances: B-E: 2.

Choose E and update distances: E-C: 5, E-F: 8.

Shortest Paths from A to Other Vertices:

A-B-C: Distance = 3

A-D: Distance = 4

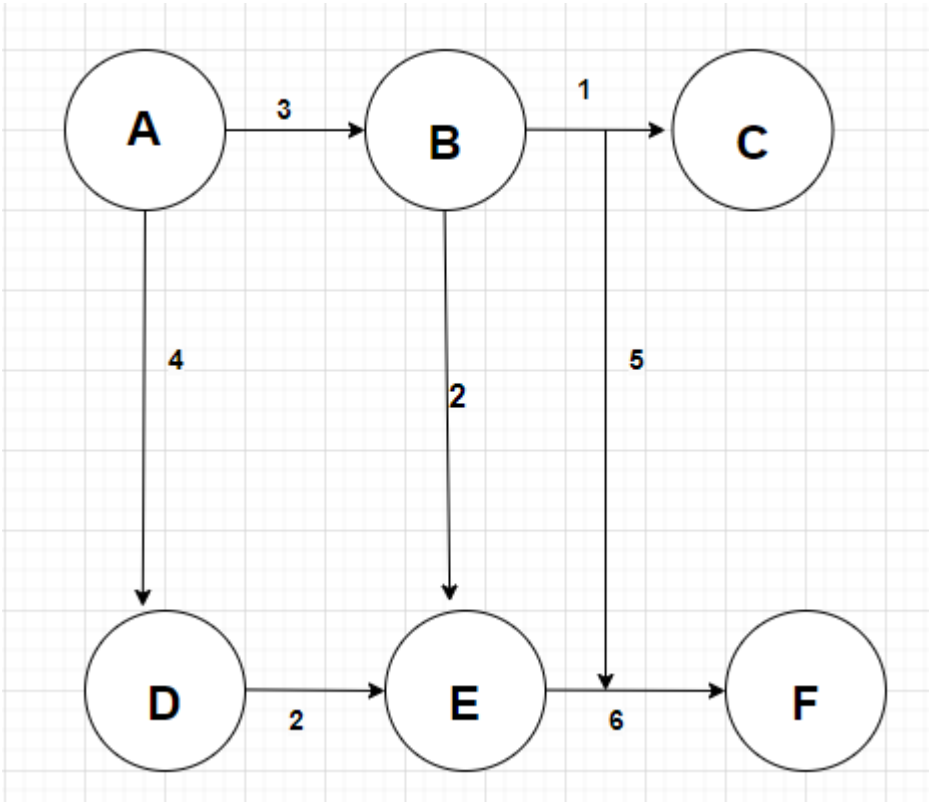
A-B-E-C: Distance = 5

A-B-E-F: Distance = 8

19)

Apply the Bellman-Ford algorithm to find the shortest path from vertex A to all other vertices in the weighted graph given below. Show each step of the algorithm and the final shortest paths.

Weighted Graph:



Solution:

Bellman-Ford Algorithm for Shortest Path:

Initialize distances from the source (A) to all other vertices as infinity except for A (0).

Relax edges repeatedly (n-1 times) to find shortest paths.

Steps:

Initialize distances: A: 0, B: ∞, C: ∞, D: ∞, E: ∞, F: ∞.

Relax edges: A-B: 3, A-D: 4.

Relax edges again (n-1 times):

B-E: 2.

E-C: 5, E-F: 8.

Shortest Paths from A to Other Vertices:

A-B-C: Distance = 3

A-D: Distance = 4

A-B-E-C: Distance = 5

A-B-E-F: Distance = 8

The Bellman-Ford algorithm successfully identifies the shortest paths from vertex A to all other vertices in the graph.

20) Solve the 0/1 Knapsack problem using the branch and bound technique for the following items and their respective weights and values:

Item	Weight	Value
A	6	15
B	3	8
C	2	5
D	5	10
E	4	9

Given a knapsack capacity of 10 units, determine the maximum value that can be obtained using the branch and bound technique.

Solution:

The branch and bound technique involves creating a search tree where each node represents a decision to include or exclude an item in the knapsack. The nodes are explored based on an upper bound and a lower bound to prune branches.

The solution space tree is explored, and the maximum value that can be obtained within the capacity constraint is found by backtracking through the tree:

Optimal Solution:

Items selected: A, C, D

Total value: 15 + 5 + 10 = 30 units

Therefore, the maximum value that can be obtained using the branch and bound technique is 30 units.