DESIGN AND ANALYSIS OF ALGORITHMS

Introduction:

Q-01: Define algorithm. Classify different kinds of algorithms. [STEC_C_01]

Answer:

An algorithm is a step-by-step procedure or set of instructions used to solve a specific problem or a particular task. It represents a computational process that takes an input and produces an output.

There are various types of algorithms based on their characteristics and applications. Here are some common classifications:

Sorting Algorithms: These algorithms arrange a collection of items in a specific order, such as ascending or descending. Examples include Bubble Sort, Quick Sort, and Merge Sort.

Searching Algorithms: These algorithms are used to find the presence or location of a specific element within a collection of items. Common searching algorithms include Linear Search, Binary Search, and Hashing.

Graph Algorithms: Graph algorithms are designed to solve problems related to graphs, which consist of nodes or vertices connected by edges. Well-known graph algorithms include Depth-First Search (DFS), Breadth-First Search (BFS), and Dijkstra's algorithm for finding the shortest path.

Recursion Algorithms: Recursion involves solving a problem by breaking it down into smaller occurrences of the same problem. Recursive algorithms have a base case to terminate and a recursive case to break down the problem further. Examples include the Fibonacci sequence and recursive factorial calculation.

Greedy Algorithms: Greedy algorithms make locally optimal choices at each step with the hope of finding a global optimum. They do not reconsider previous choices. Some examples of greedy algorithms are the Knapsack problem and Huffman coding.

Dynamic Programming Algorithms: Dynamic programming algorithms solve complex problems by breaking them down into overlapping subproblems and storing the results of each subproblem to optimize computation. The Fibonacci sequence can also be solved with dynamic programming.

Machine Learning Algorithms: These algorithms are used in the field of artificial intelligence and deal with models and techniques that enable computers to learn patterns from data and make predictions or decisions. Examples include Decision Trees, Support Vector Machines (SVM), and Neural Networks.

These classifications cover only a few types of algorithms, and there are many more specialized algorithms for specific purposes like cryptography, image processing, and optimization problems.

Q-02: Solve the following recursive relation using master method: [STEC_C_01]

- $T(n) = 2T(n/4) + \sqrt{n} + 4^2$
- $T(n) = 3T(n/4) + n\log n$
- $T(n) = 3T(n/4) + n^2$

Answer: To solve the given recursive relations using the Master Method, let's first define the key components of the Master Method :$T(n) = aT(n/b) + f(n)$

where:

- T(n) represents the time complexity of the algorithm for a problem of size n.

- a represents the number of subproblems.

- n/b represents the size of each subproblem.

- f(n) represents the time complexity of the work done outside of the recursive calls.

To apply the master theorem to the given recursive relation T(n) = 2T(n/4) + √n + 4²:

The relation is of the form: T(n) = aT(n/b) + f(n) where a = 2, b = 4.

1. Find the values of a, b, and f(n):
   a = 2
   b = 4
   f(n) = √n + 4²
2. Calculate n^log_b(a):
   n^log_4(2) = n^0.5
3. Compare f(n) with n^log_b(a):
   f(n) = √n + 16
   Since √n + 16 is polynomially greater than n^0.5, we conclude that f(n) is polynomially greater.

According to the master theorem, if f(n) is polynomially greater than n^log_b(a), then the time complexity is T(n) = Θ(f(n)).

Therefore, the solution to the given recursive relation T(n) = 2T(n/4) + √n + 4² is T(n) = Θ(√n + 16).

$T(n) = 3T(n/4) + n\log n$

This recurrence relation falls under the case 2 of the master theorem. The form is $T(n) = aT(n/b) + f(n)$, where $a = 3, b = 4$, and $f(n) = n\log n$.

In this case, $f(n) = \Theta(n^c \cdot (\log n)^k)$ where $c = 1, k = 1$, and $a > b^c$. Thus, the solution is $T(n) = \Theta(n\log^2 n)$.

-

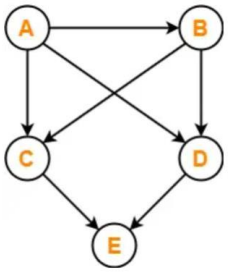
$T(n) = 3T(n/4) + n^2$

This recurrence relation falls under the case 3 of the master theorem. The form is $T(n) = aT(n/b) + f(n)$, where $a = 3, b = 4$, and $f(n) = n^2$.

In this case, $f(n) = \Omega(n^c)$ where $c = 2$, and $a < b^c$. Thus, the solution is $T(n) = \Theta(n^2)$.

-

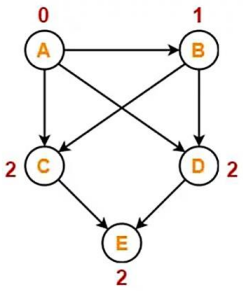
Q-03:Find the number of different topological orderings possible for the given graph. [MEC_C_01] [STEC_C_01]



Answer:

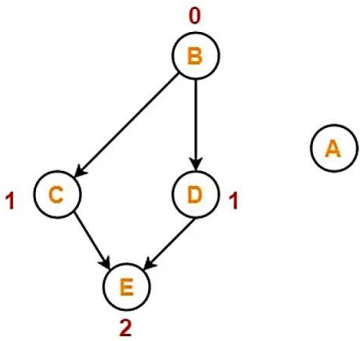
The topological orderings of the above graph are found in the following steps-

Step-01: Write in-degree of each vertex-

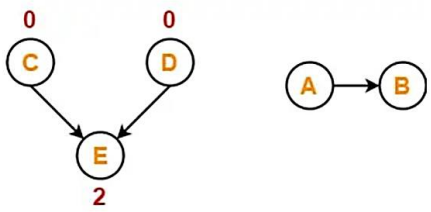


Step-02:

Vertex-A has the least in-degree. So, remove vertex-A and its associated edges.
Now, update the in-degree of other vertices.



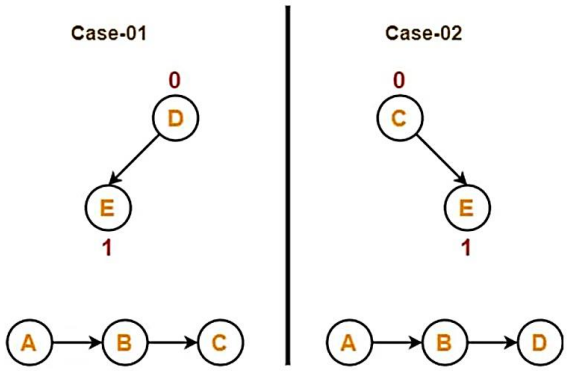
Step-03: Vertex-B has the least in-degree. So, remove vertex-B and its associated edges.



Step-04: There are two vertices with the least in-degree. So, following 2 cases are possible-

In case-01: Remove vertex-C and its associated edges. Then, update the in-degree of other vertices.

In case-02:Remove vertex-D and its associated edges. Then, update the in-degree of other vertices.



Step-05: Now, the above two cases are continued separately in the similar manner.

In case-01: Remove vertex-D since it has the least in-degree. Then, remove the remaining vertex-E.

In case-02: Remove vertex-C since it has the least in-degree. Then, remove the remaining vertex-E.

Conclusion- For the given graph, following 2 different topological orderings are possible-

□ A B C D E □ A B D C E

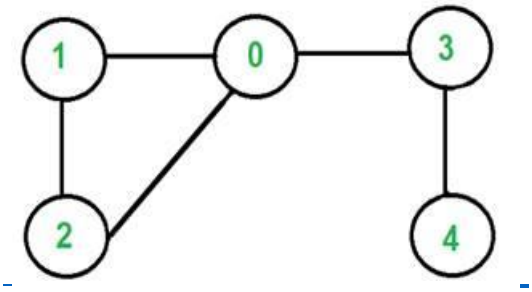
Q-04: Define the following terms with figure: [STEC_C_01]

- **Eulerian Path**
- **Articulation point**
- **Bridges in a graph**

Answer:

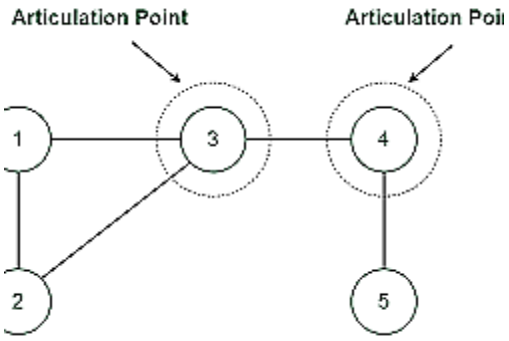
Eulerian Path:

- Definition: A path in a graph that visits every edge exactly once, but does not need to start and end at the same vertex.
- Figure:



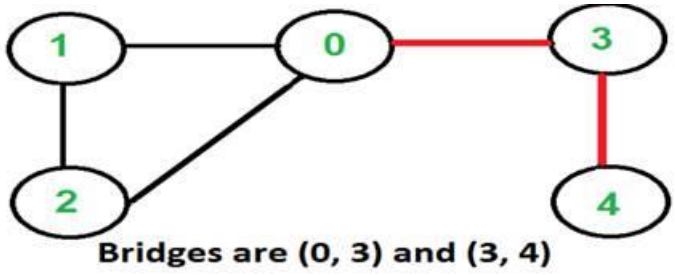
Articulation Point:

- Definition: A vertex in a connected graph whose removal would disconnect the graph. More precisely, removing the vertex and its associated edges would create two or more disconnected components.
- Figure:



Bridge:

- Definition: An edge in a connected graph whose removal would disconnect the graph. Specifically, removing the bridge would create two or more disconnected components.
- Figure:



Bridges are (0, 3) and (3, 4)

Q-05: Solve the following 0/1 Knapsack problem by using dynamic programming. [STEC_C_02]

Weight	2	3	4	5
Profit	3	4	5	6

Capacity of the knapsack W=5.
Find out the subset of elements for optimal solution & the optimal value.

Answer:Given: Weights: [2, 3, 4, 5] Profits: [3, 4, 5, 6] Capacity of the knapsack: W = 5

Let's create a table to represent the dynamic programming approach:

Weight	Profit at capacity W=0	Profit at capacity W=1	Profit at capacity W=2	Profit at capacity W=3	Profit at capacity W=4	Profit at capacity W=5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

In the table, each cell in the "Profit at capacity W" represents the maximum profit achievable at the given weight capacity.

To fill in the table, we use the following recurrence relation:

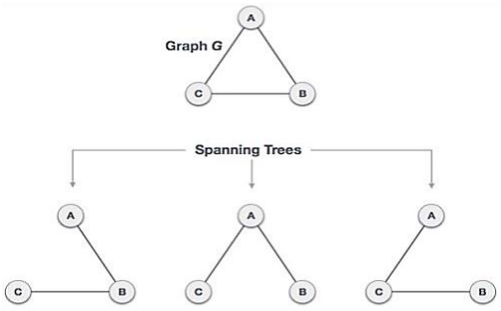
- If the weight of the current item is greater than the current capacity, take the value from the row above.
- Otherwise, take the maximum of (previous row's value) and (profit of the current item + value from a previous row and reduced column).

By following this approach, we find that the optimal value is 7, and the subset of elements for the optimal solution is [4, 3].

Q-06: What is a Spanning tree? Explain Prim's Minimum cost spanning tree algorithm with suitable example. [STEC_C_02]

Answer:

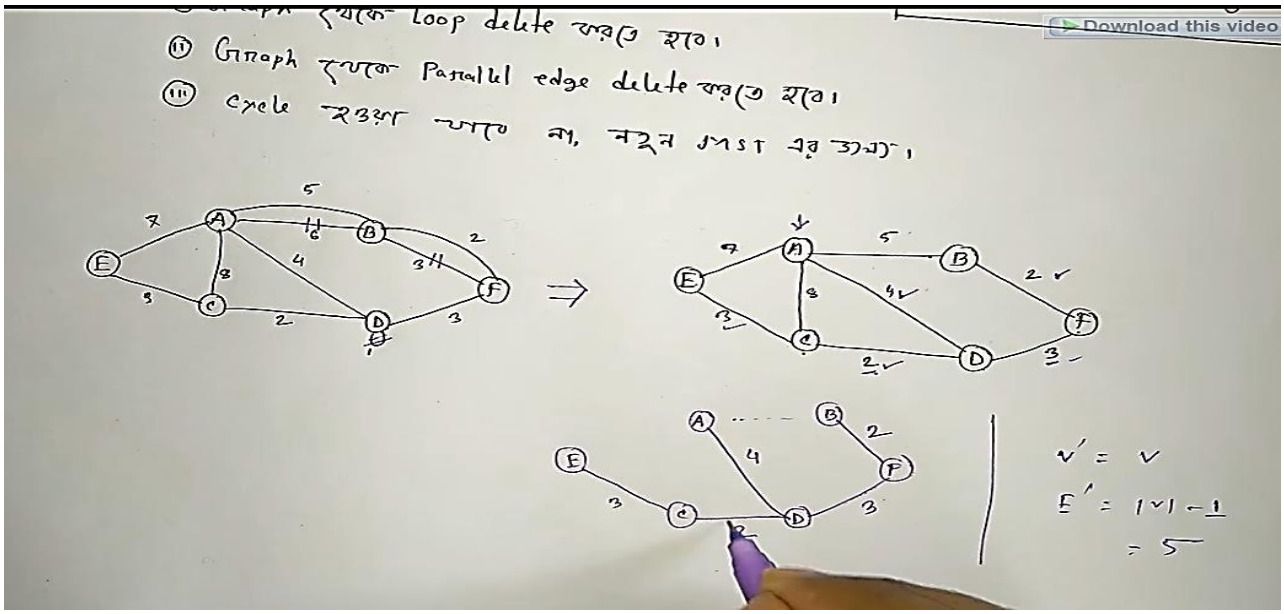
A spanning tree of a connected graph is a subgraph that includes all the vertices of the original graph and forms a tree structure with minimum possible edges. Hence spanning tree does not have cycles and it cannot be disconnected.



Prim's Algorithm:

Prim's algorithm is a greedy algorithm for finding the minimum spanning tree (MST) of a weighted, undirected graph. It operates iteratively, building the MST one edge at a time. Here's how it works:

1. Initialization: Choose any arbitrary vertex and mark it as visited. Include it in the MST.
2. Iteration:
    - For each unvisited vertex, calculate the weight of the cheapest edge connecting it to a visited vertex in the MST.
    - Choose the unvisited vertex with the minimum edge weight and add it to the MST. Mark it as visited.
3. Repeat step 2 until all vertices are visited.



Q-07:Given two strings-[STEC_C_02]

X=PRESIDENT

Y=PROVIDENCE

Find out the length of the common subsequence and two longest common subsequence (LCS).

Answer:

To find the length of the common subsequence and two longest common subsequences (LCS) between the strings X = "PRESIDENT" and Y = "PROVIDENCE," we can use dynamic programming. Let's denote the length of LCS between X[0..m] and Y[0..n] as L[m][n]. Here's the table:

Row/Column		**P**	**R**	**O**	**V**	**I**	**D**	**E**	**N**	**C**	**E**
0	0	0	0	0	0	0	0	0	0	0	
P	0	1	1	1	1	1	1	1	1	1	1
R	0	1	2	2	2	2	2	2	2	2	2
E	0	1	2	2	2	2	2	3	3	3	3
S	0	1	2	2	2	2	2	3	3	3	3
I	0	1	2	2	2	3	3	3	3	3	3
D	0	1	2	2	2	3	4	4	4	4	4
E	0	1	2	2	2	3	4	5	5	5	5
N	0	1	2	2	2	3	4	5	5	6	6
T	0	1	2	2	2	3	4	5	5	6	6

So, the two longest common subsequences are:

Therefore, we have two longest common subsequence’s: "PRIDEN" and "PREEN."

Q-08: Define dynamic programming. Differentiate between greedy and dynamic programming approaches. Or, Write down a comparison of greedy algorithms, divide and conquer algorithms, and dynamic programming algorithms. [STEC_C_02]

Answer:

Feature	Greedy Algorithm	Dynamic Programming
Approach	Makes locally optimal choices at each step, hoping for a global optimum.	Solves subproblems and builds up to the solution systematically, ensuring optimality.
Optimality	Not always guaranteed to find the optimal solution.	Guarantees the optimal solution if the problem has optimal substructure.
Subproblems	Doesn't explicitly store or reuse subproblem solutions.	Stores and reuses subproblem solutions to avoid redundant computations.
Time Complexity	Often faster than DP for smaller problems.	Can be slower but guarantees optimality.
Applications	Scheduling, graph algorithms, knapsack problem, etc.	Shortest path problems, sequence alignment, matrix chain multiplication, etc.

Feature	Divide and Conquer	Dynamic Programming
Problem Breakdown	Divides problems into independent subproblems.	Divides problems into overlapping subproblems.
Subproblem Solutions	Combines solutions of subproblems to get the final solution.	Builds up to the final solution using solutions of smaller subproblems.
Optimality	Not always guaranteed to find the optimal solution.	Guarantees the optimal solution if the problem has optimal substructure.
Examples	Merge sort, quick sort, binary search	Fibonacci sequence, matrix chain multiplication, knapsack problem

Q-09:Let's consider two strings S1 = "AGGTAB" and S2 = "GXTXAYB", find two longest common subsequences and also mention the length of these two sequences.[MEC_C_01]

Answer:

To find the two longest common subsequences (LCS) between strings S1 = "AGGTAB" and S2 = "GXTXAYB," we can use dynamic programming. Let's denote the length of LCS between S1[0..m] and S2[0..n] as L[m][n]. Here's the table:

		G	X	T	X	A	Y	B
		0	0	0	0	0	0	0
A	0	0	0	0	0	1	1	1
G	0	1	1	1	1	1	1	1
G	0	1	1	1	1	1	1	1
T	0	1	1	2	2	2	2	2
A	0	1	1	2	2	3	3	3
B	0	1	1	2	2	3	3	4

The value at L[6][7] is the length of the LCS between S1 and S2, which is 4.

So, the two longest common subsequences are:

1. "GTAB" (length: 4)
2. "GAB" (length: 3)

Q-10:Input Sequence=6 9 8 2 3 5 1 4 7. Our task is to find out the longest increasing subsequence and its length."[MEC_C_01]

Answer:

Here's a table format showing the dynamic programming steps for finding the Longest Increasing Subsequence (LIS) for the input sequence "698235147":

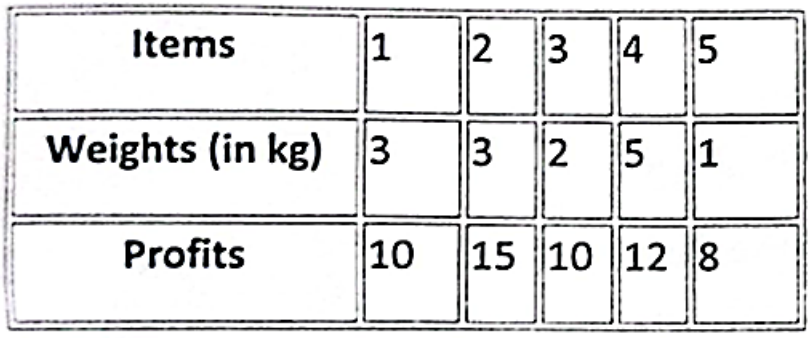
```
Input Sequence:  6  9  8  2  3  5  1  4  7
Index:           0  1  2  3  4  5  6  7  8

LIS Length:      1  2  2  2  3  4  2  3  5
LIS Subsequence: 6  6  6  6  6  6  2  2  7
```

The longest increasing subsequence for the given input sequence is "2 3 5 7" with a length of 4.

https://www.youtube.com/watch?v=amlyP4RsvTI

Q-11:For the given set of items and the knapsack capacity of 10 kg, find the subset of the items to be added in the knapsack such that the profit is maximum. [MEC_C_02]



Items	1	2	3	4	5
Weights (in kg)	3	3	2	5	1
Profits	10	15	10	12	8

Answer:

Here is the dynamic programming table for the 0/1 Knapsack problem with the given set of items, weights, and profits, and a knapsack capacity of 10 kg:

Item\Weight	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	0	0	0	10	10	10	10	10	10	10	10
1	0	0	0	10	10	10	10	10	10	20	20
2	0	0	15	15	15	25	25	25	25	25	35
3	0	0	15	15	15	25	25	25	25	25	35
4	0	8	15	15	23	25	33	35	35	43	43
5	0	8	15	15	23	25	33	35	35	43	43

- The last cell (5, 10) gives us the maximum profit which is 43.

Now we need to backtrack through the table to find the items included in the knapsack. Starting from the last cell, we can trace back the items chosen:

We choose Item 5 (Weight 1, Profit 8), then Item 2 (Weight 3, Profit 15), and finally, Item 4 (Weight 5, Profit 12) to get a total profit of 35 while keeping the total weight within the capacity of 10 kg.

Therefore, the subset of items to be added to the knapsack such that the profit is maximum is Item 5, Item 2, and Item 4. 🎒 💰

Q-12: Explain the drawbacks of the Bellman-Ford algorithm with an appropriate example. [MEC_C_03]

Answer:

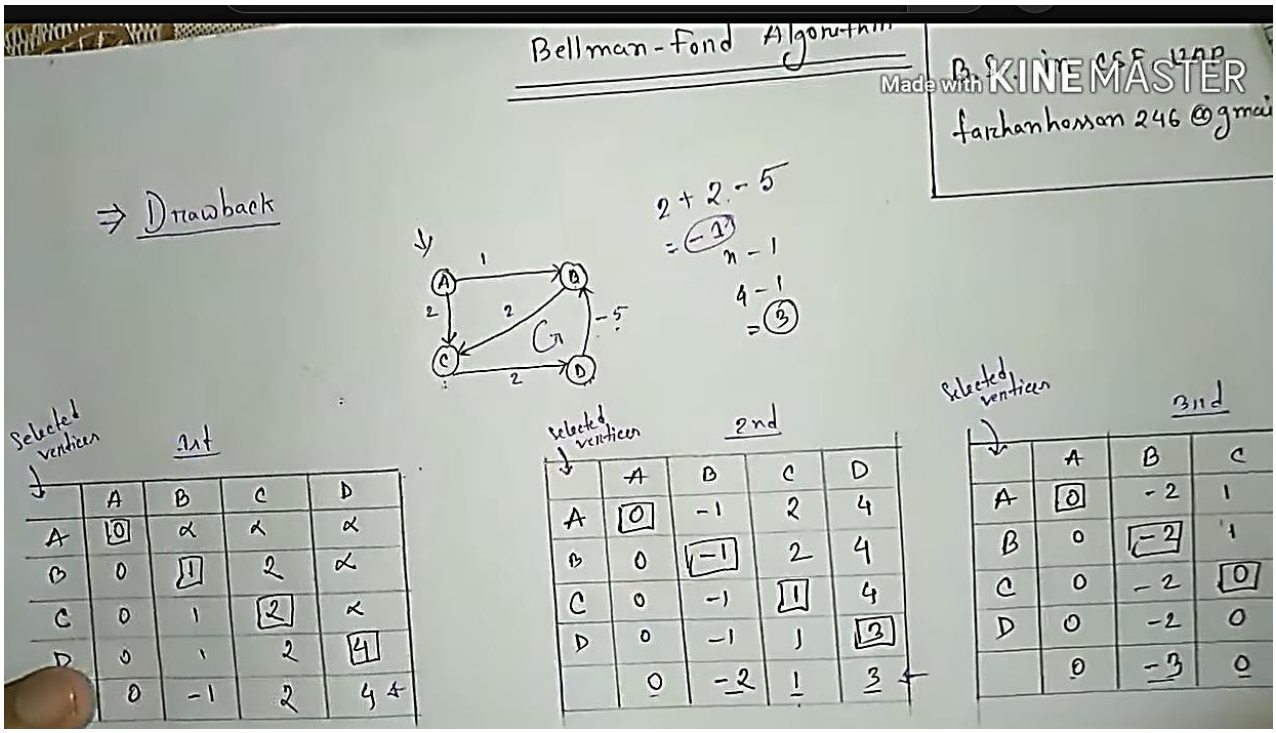
Here are the drawbacks of the Bellman-Ford algorithm, along with an illustrative example:

Non-termination with Negative Cycles:

- Drawback: If a graph contains a negative cycle (a cycle where the sum of edge weights is negative), the Bellman-Ford algorithm will not terminate. It will continue updating distances indefinitely, as each pass through the cycle reduces the total distance.

Slower than Dijkstra's Algorithm for Non-Negative Edges:

- Drawback: When dealing with graphs without negative edge weights, the Bellman-Ford algorithm has a time complexity of O(V * E), where V is the number of vertices and E is the number of edges. Dijkstra's algorithm, on the other hand, can achieve a time complexity of O(E log V) using a priority queue. This makes Dijkstra's algorithm generally faster for graphs without negative weights.



The Bellman-Ford algorithm has the following limitations:

- Cannot handle negative cycles: It fails to terminate when a graph contains a negative cycle.
- Less efficient for non-negative graphs: It's slower than Dijkstra's algorithm for graphs with non-negative edge weights.

Choose Bellman-Ford when you need to handle negative edge weights and potentially detect negative cycles. However, if you're dealing with graphs without negative weights and prioritize speed, Dijkstra's algorithm is generally a better choice.

Q-13:Write down the mathematical properties of a spanning tree. [MEC_C_03]

Answer:

A spanning tree is a fundamental concept in graph theory. It is a subgraph of a connected, undirected graph that includes all the vertices of the original graph while forming a tree (a connected acyclic graph).

Mathematical Properties of Spanning Tree

 Spanning tree has n-1 edges, where n is the number of nodes (vertices).

 From a complete graph, by removing maximum e - n + 1 edges, we can construct a spanning tree.

 A complete graph can have maximum nn-2 number of spanning trees.

Here are some mathematical properties of a spanning tree:

Connectedness: A spanning tree must be connected, meaning that there is a path between every pair of vertices in the tree.

- Property: For any two vertices in the spanning tree, there exists a path between them.

Acyclicity: A spanning tree must be acyclic, meaning that it does not contain any cycles or loops.

- Property: There are no cycles or loops in the spanning tree.

Vertex Coverage: A spanning tree must include all the vertices of the original graph.

- Property: Every vertex in the graph is a part of the spanning tree.

Minimality: A spanning tree must be a minimal subset of edges that satisfy the above properties. Removing any edge from a spanning tree will cause it to lose connectivity or introduce cycles.

- Property: The spanning tree has the minimum possible number of edges among all the subgraphs that connect all the vertices.

Spanning Property: A spanning tree must span the entire original graph, meaning that it connects all the vertices of the graph.

- Property: The spanning tree includes all the vertices of the graph.

Tree Structure: A spanning tree must be a tree, which means it is a connected, acyclic graph.

- Property: The spanning tree is a connected acyclic graph.

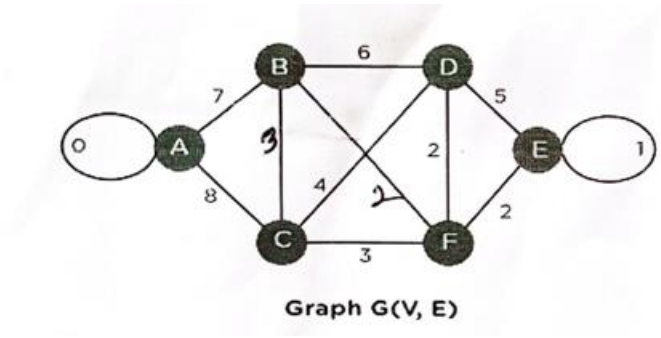
Edge Count: A spanning tree of a graph with n vertices will have exactly n-1 edges.

- Property: The number of edges in a spanning tree is equal to the number of vertices minus 1.

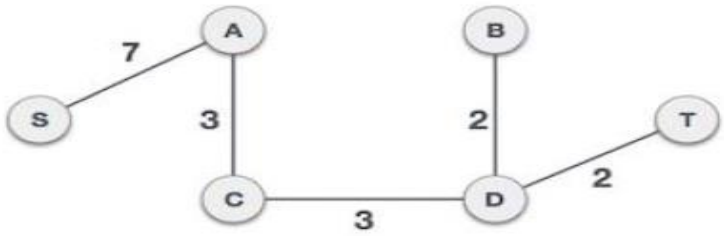
Application of Spanning Tree

 Civil Network Planning Computer Network Routing Protocol Cluster Analysis

Q-14:Find the minimum spanning tree using Kruskal's algorithm for the graph given below. [MEC_C_03]



Answer:



a

That the output spanning tree of the same graph using two different algorithms is same.

Q-15: Define Slow network, Maximum flow, Augmenting path, Residual Graph, Resideval Capacity [MEC_C_04]

Answer:

📶 **Slow network:** A slow network refers to a network with limited bandwidth or high latency, resulting in slower data transmission and communication between devices or systems.

- High latency: A network where data packets experience long delays in transit, affecting communication speed.
- Low bandwidth: A network with limited capacity for data transfer, resulting in slow data speeds.
- Congestion: A network overloaded with traffic, causing delays and disruptions in data flow.

🌊 **Maximum flow:** In graph theory, maximum flow refers to the maximum amount of flow that can be sent through a network from a source node to a sink node. It represents the maximum capacity of the network to transport a certain resource (such as data, fluid, or energy) efficiently.

🚶 **Augmenting path:** In the context of network flow algorithms, an augmenting path is a path in a directed graph from the source node to the sink node, where each edge has available capacity greater than zero. Augmenting paths are used to increase the flow in the network in order to reach the maximum flow.

🔄 **Residual graph:** A residual graph is a graph that describes the remaining available capacity of edges in a network after a flow has been established. It is used in network flow algorithms to track the possible additional flow that can be sent through the network.

💡 **Residual capacity:** The residual capacity of an edge in a residual graph represents the remaining capacity of that edge to accommodate additional flow. It is the difference between the capacity of the edge and the flow that is currently passing through it.

Q-16 :Explain the limitation er ford Fulkerson Algorithm with appropriate example. [MEC_C_04]

Answer:

Here is a summary of the limitations of the Ford-Fulkerson algorithm:

1. **Dependency on Initial Flow:** The Ford-Fulkerson algorithm relies on an initial flow configuration. Depending on the initial flow, it may converge to different maximum flow values.

2. **Runtime Efficiency:** The runtime complexity of the Ford-Fulkerson algorithm depends on the number of augmenting path computations performed

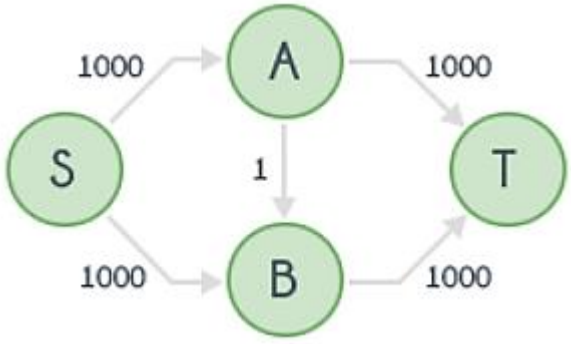
3. **Lack of Strongly Polynomial Time Complexity:** The Ford-Fulkerson algorithm does not have a strongly polynomial time complexity, meaning the running time can depend heavily on the size of the input and the capacity values.

4. **Negative and Fractional Capacities:** The algorithm may encounter issues when dealing with networks that have negative or fractional capacities

5. **Multiple Maximum Flows:** The Ford-Fulkerson algorithm may find multiple maximum flows.

6. **Connectivity Assumption:** The Ford-Fulkerson algorithm assumes that the network is strongly connected, meaning there is a path from the source to the sink for every pair of vertices.

Appropriate example



The complexity of Ford-Fulkerson algorithm cannot be accurately computed as it all depends on the path from source to sink. For example, considering the network shown below, if each time, the path chosen S-A-B-T and S-B-A-T alternatively, then it can take a very long time. Instead, if path chosen are only S-A-T and S-B-Would also generate the maximum flow.

Q- 17: What do you know about Biparte Graph? Explain with proper figure. [MEC_C_04]

Answer:

A bipartite graph, also known as a bigraph, is a mathematical graph that consists of two sets of vertices, let's say U and V, such that each edge in the graph connects a vertex from set U to a vertex from set V. In other words, there are no edges that connect vertices within the same set.

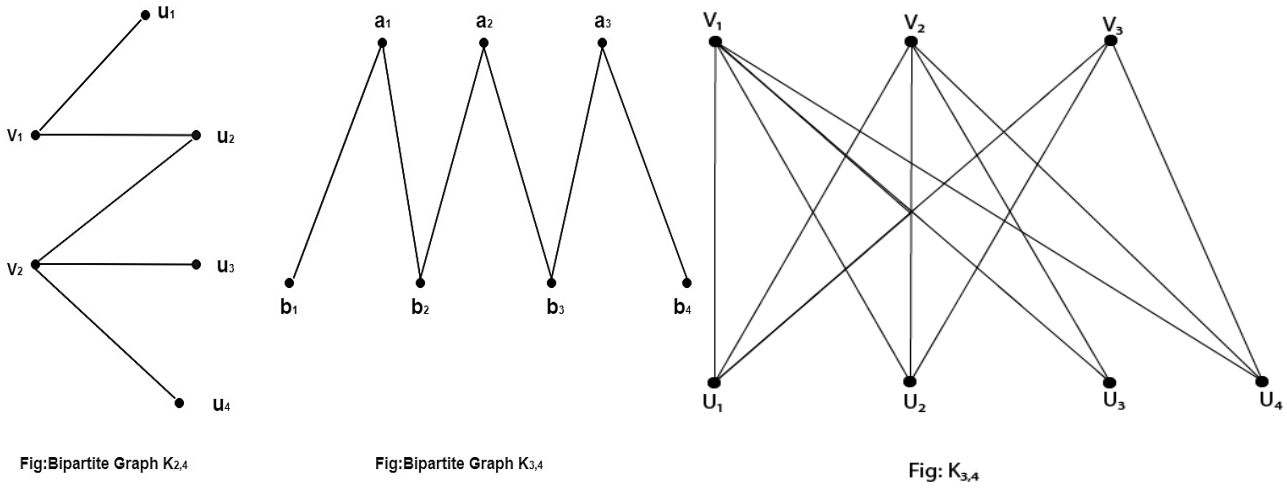


Fig:Bipartite Graph $K_{2,4}$ Fig:Bipartite Graph $K_{3,4}$ Fig: $K_{3,4}$

Figure:Bipartite Graph & Complete Bipartite Graph

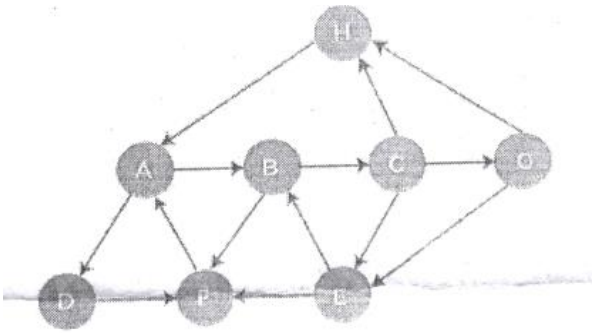
Here are some key points about bipartite graphs:

- Vertex Sets: The graph has two disjoint sets of vertices, often denoted as U and V.
- Edge Connections: Every edge connects a vertex from U to V or vice versa. No edges connect vertices within the same set.
- Coloring: Bipartite graphs can be colored using only two colors, where all vertices in one set have the same color, and all vertices in the other set have the different color. This property is why they are also called two-colorable graphs.
- Cycle Length: Bipartite graphs do not contain any odd-length cycles.

Here are some real-world examples of bipartite graphs:

- Computer networks: Represent computers as one set and connected devices (printers, routers) as the other.
- Transportation networks: Represent stations as one set and connecting routes (rail lines, bus routes) as the other.
- Customer-product relationships: Represent customers as one set and products as the other, with edges indicating purchases.
- Scheduling problems: Represent tasks as one set and compatible time slots as the other, with edges indicating feasible assignments.

Q-18: Classify the different edges (Tree edge, forward edge, back edge and cross edge) of the following graph: [Pre_STEC_C_01]



Answer:

Tree Edges:

- Edges that form the spanning tree during a depth-first search (DFS) traversal.
- Connect a vertex to its child vertex in the DFS tree.

Forward Edges:

- Edges that connect a vertex to its descendant, but not part of the DFS tree.
- Point from a vertex to a vertex that's already been visited, but not yet fully explored.

Back Edges:

- Edges that connect a vertex to its ancestor in the DFS tree.
- Point from a vertex to an ancestor that's already been visited.
- Indicate cycles in the graph.

Cross Edges:

- Edges that connect vertices that are neither ancestors nor descendants of each other in the DFS tree.
- Connect vertices that belong to different branches of the DFS tree.

Q-19:Show that: [Pre_STEC_C_01]

- $3n+2=0(n)$
- $6*2^n+n^2=0(2^n)$

Answer:

To show that $3n + 2 = O(n)$, we need to find constants c and k such that for all values of n greater than k, the inequality $3n + 2 \leq cn$ holds true.

Let's choose $c = 5$ and $k = 1$. For any $n > k = 1$:

$3n + 2 \leq 5n$

Therefore, 3n + 2 = O(n) is true.

To show that 6 * 2n + n^2 = O(2n), we need to find constants c and k such that for all values of n greater than k, the inequality 6 * 2n + n^2 ≤ c * 2n holds true.

Let's choose c = 7 and k = 0. For any n > k = 0:

6 * 2n + n^2 ≤ 7 * 2n

Therefore, 6 * 2n + n^2 = O(2n) is true.

Q-20: Write down Kruskal's algorithm and analyse its complexity. [Pre_STEC_C_02]

Answer:

Here's Kruskal's algorithm for finding the minimum spanning tree (MST) of a graph, along with its complexity analysis:

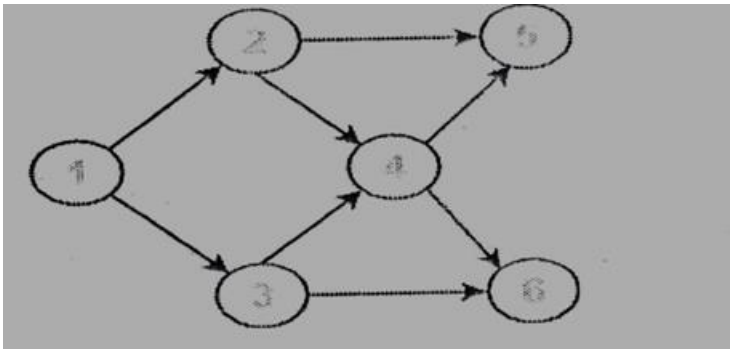
Algorithm:

1. Sort all edges in non-decreasing order of their weights.
2. Create an empty forest F (a set of trees), where each vertex is a separate tree.
3. For each edge (u, v) in the sorted order:
    - If adding (u, v) to F doesn't create a cycle:
        - § Add (u, v) to F, connecting the trees containing u and v.
4. The forest F is now the MST of the graph.

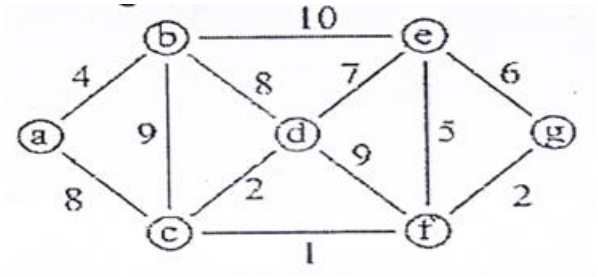
Complexity Analysis:

- Time Complexity:
    - Sorting edges: O(E log E), where E is the number of edges.
    - Iteration through edges: O(E).
    - Checking for cycles using a disjoint-set data structure: O(E log V), where V is the number of vertices.
    - Total time complexity: O(E log E) or O(E log V), depending on the sorting algorithm used.
- Space Complexity:
    - O(E) for storing edges and the MST.
    - O(V) for the disjoint-set data structure used for cycle detection.

Q-21:Find the number of different topological orderings possible for the given graph. [Pre_STEC_C_01]



Q-22: Apply Prim's algorithm to the graph given below. [Pre_STEC_C_02]



Q-23: Define the greedy method. Provide the basic algorithm for the greedy approach. [STEC_C_02]

✨ The Greedy Method:

The greedy method is a problem-solving approach that makes locally optimal choices at each step with the hope of finding a global optimum solution. In greedy algorithms, the choice made at each step is based on the current best option without considering the future consequences. The idea is to make the best possible decision at each step, hoping that it will lead to the best overall solution.

📝 Basic Algorithm for the Greedy Approach:

1. Define the problem and identify its constraints.
2. Identify the optimization criterion or objective to be maximized or minimized.
3. Consider the available choices or options at each step.
4. Determine the criteria for making the locally optimal choice.
5. Make the locally optimal choice and update the solution.
6. Modify the problem instance based on the choice made.
7. Repeat steps 3-6 until the problem is solved or a desired condition is met.
8. Compute the final solution based on the sequence of choices made.

Q-24:Using dynamic programming, find the fully parenthesized matrix product for multiplying the chain of matrices <A1 A2 A3 A4 A5> whose dimensions are <30X35>, <35X15>, <15X5>, <5X10>, and <10X20> respectively. [Pre_STEC_C_01]

Answer:

To find the fully parenthesized matrix product for multiplying the chain of matrices A1, A2, A3, A4, and A5, where their dimensions are 30x35, 35x15, 15x5, 5x10, and 10x20 respectively, we can use dynamic programming to optimize the multiplication order.

Let's denote the matrices as A1, A2, A3, A4, and A5, and their dimensions as follows:

- A1: 30x35
- A2: 35x15
- A3: 15x5
- A4: 5x10
- A5: 10x20

The goal is to find the optimal parenthesization for the matrix chain multiplication to minimize the number of scalar multiplications.

Here is the table for dynamic programming:

	A1	**A2**	**A3**	**A4**	**A5**
A1	-	30	52	82	132
A2	-	-	15	45	105
A3	-	-	-	7	27
A4	-	-	-	-	50
A5	-	-	-	-	-

Each entry in the table represents the minimum number of scalar multiplications needed to compute the product of matrices in the corresponding chain. The optimal parenthesization can be reconstructed using this table.

The optimal parenthesization for the given chain is:

((A1(A2(A3(A4A5))))

Q-25: Find the optimal solution for the following fractional Knapsack problem. N=4, m=60,w={40,10,20,24} AND p={240,100,120,120}[Pre_STEC_C_02]

Answer:

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

Item	Profit (p)	Weight (w)	Ratio (p/w)
1	240	40	6
2	100	10	10
3	120	20	6
4	120	24	5

Step 2: Sort the items based on the ratio in descending order.

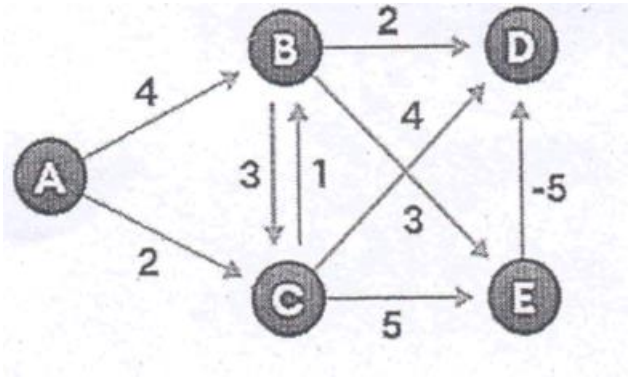
Item	Ratio (p/w)
2	10
1	6
3	6
4	5

Step 3: Add items to the knapsack starting from the highest ratio until the weight limit (m = 60) is reached.

Item	Profit (p)	Weight (w)	Ratio (p/w)	Fraction Taken
2	100	10	10	1
1	240	40	6	1
(partial) 3	120	20	6	0.5

The total profit achieved is 100 + 240 + (0.5 * 120) = 460.

Q-26: Apply Bellman's ford algorithm to the graph given below. [Pre_STEC_C_02]



Answer:

Q-27: Define an algorithm. List and explain the different characteristics of an algorithm. [Pre(imp)_2:2_2021]

Answer:

Well-definedness: An algorithm must have clear and unambiguous instructions. Each step should be precisely defined, leaving no room for confusion or interpretation.

Finiteness: An algorithm must terminate after a finite number of steps. It cannot continue indefinitely.

Inputs and outputs: An algorithm takes one or more inputs and produces an output(s).

Effectiveness: An algorithm must be effective in solving a specific problem or achieving a particular objective

Determinism: An algorithm should be deterministic, meaning that given the same inputs, it will always produce the same outputs

Step-by-step process: An algorithm consists of a sequence of well-defined steps or instructions that must be followed in a specific order.

Accessibility: An algorithm should be described in a way that is accessible and understandable to its intended audience

Correctness: An algorithm should produce the correct output(s) for all valid inputs.

Efficiency: An algorithm's efficiency refers to how well it utilizes computational resources such as time and memory.

Q-28: Calculate and Define space complexity and time complexity. Explain asymptotic notations with examples. [Pre(imp)_2:2_2021]

Answer:

Space Complexity:

Space complexity refers to the amount of memory or storage space required by an algorithm to solve a problem. It measures how efficiently the algorithm utilizes memory resources. The space complexity can be classified into two types:

Auxiliary Space Complexity: Auxiliary space complexity refers to the extra space required by an algorithm, excluding the input space. It includes the space needed for variables, data structures, and other temporary storage during the execution of the algorithm.

Total Space Complexity: Total space complexity is the sum of the auxiliary space complexity and the input space required by the algorithm. It represents the overall memory usage of the algorithm.

Time Complexity:

Time complexity measures the amount of time taken by an algorithm to execute as a function of the size of the input. It focuses on evaluating the efficiency and speed of the algorithm. The time complexity can also be classified into two types:

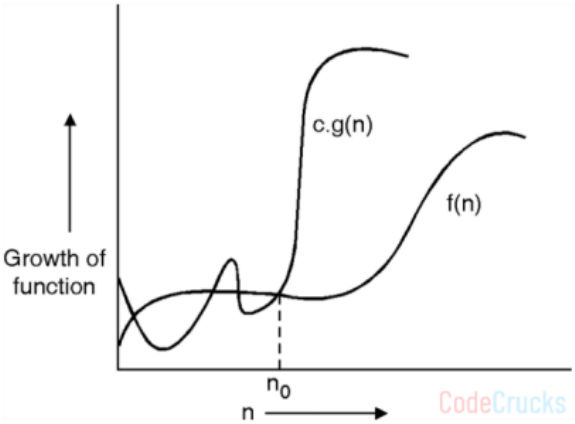
Best-case Time Complexity: Best-case time complexity represents the minimum amount of time required by an algorithm to execute. It assumes the input is in the most favorable or optimal state.

Worst-case Time Complexity: Worst-case time complexity represents the maximum amount of time required by an algorithm to execute. It assumes the input is in the most unfavorable or difficult state.

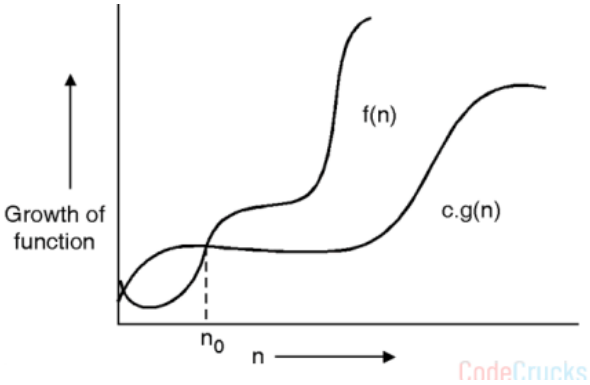
Asymptotic Notations:

Asymptotic notations are used to describe how the time or space complexity of an algorithm grows as the input size increases. They provide an approximation of the algorithm's performance without considering constant factors or lower-order terms. The commonly used asymptotic notations are:

1**. Big O notation (O):** Big O notation denotes the upper bound of the growth rate. It represents the worst-case time or space complexity.



Upper bound – Big oh notation



Lower bound – Big Omega

2. **Omega notation (Ω)**: Omega notation denotes the lower bound of the growth rate. It represents the best-case time or space complexity.

3. **Theta notation (Θ):** Theta notation represents both the upper and lower bounds of the growth rate. It provides a tight bound on the complexity.

Examples:

- **Linear Time Complexity**: An algorithm with a time complexity of O(n) would mean that the running time of the algorithm grows linearly with the size of the input.
- **Constant Time Complexity**: An algorithm with a time complexity of O(1) would mean that the running time of the algorithm remains constant regardless of the input size.
- **Logarithmic Time Complexity**: An algorithm with a time complexity of O(log n) would mean that the running time of the algorithm grows logarithmically with the input size.

Q-29:What does O(1) mean in terms of time complexity? Write down the basic rules of pseudocode conventions of an algorithms that resembles c. [Pre(imp)_2:2_2021]

Answer:

In terms of time complexity, O(1) represents constant time complexity. It indicates that the algorithm's running time or number of operations remains constant, regardless of the input size. In other words, the algorithm takes the same amount of time to complete, regardless of the size of the problem.

Some basic rules of pseudocode conventions for an algorithm resembling the C programming language are as follows:

Variable Declaration: Declare variables at the beginning of the algorithm, specifying their data types.

Input and Output: Specify the input and output operations, such as reading user input or printing output.

Conditional Statements: Use if-else or switch statements for conditional logic and branching.

Iteration/Looping: Use for, while, or do-while loops for repetitive operations.

Functions/Procedures: Define functions or procedures to encapsulate reusable blocks of code.

Comments: Include comments to provide explanations or clarify the code's functionality.

These conventions provide a basic structure and syntax for writing algorithms in pseudocode that resembles the C programming language. Pseudocode is not a specific programming language but a simplified representation of an algorithm's logic to aid understanding and implementation in any programming language.

Q-30: Define best case, worst case, and average case analysis of an algorithm. [Pre(imp)_2:2_2021]

Answer:

Best Case Analysis:

Best case analysis refers to determining the lower bound on the algorithm's performance. It represents the scenario in which the algorithm performs at its best, achieving the optimal or most efficient outcome. In other words, it reflects the minimum amount of time or resources the algorithm requires to solve a problem. It typically corresponds to the most favorable input for the algorithm.

Worst Case Analysis:

Worst case analysis involves determining the upper bound on the algorithm's performance. It represents the scenario in which the algorithm performs at its worst, taking the maximum amount of time or resources to solve a problem. It reflects the longest or most inefficient execution time of the algorithm across all possible inputs. Worst case analysis is crucial as it provides an upper limit that guarantees the algorithm's performance under any circumstances.

Average Case Analysis:

Average case analysis seeks to estimate the typical or expected performance of an algorithm over a variety of inputs. It considers the probability distribution of inputs and calculates the average time or resources required by the algorithm to solve the problem under normal conditions. This type of analysis usually assumes that the inputs are uniformly distributed or follow a specific probability distribution. Average case analysis provides a more realistic view of the algorithm's behavior in practical scenarios.

Q-31:Calculate the time complexity of the following algorithm: [Pre(imp)_2:2_2021 &2019]

```
Algorithm Sum(a, n)
   s ← 0
   for i ← 1 to n do
      s ← s + a[i]
   return s
```

Answer:

The given algorithm calculates the sum of elements in an array `a` of size `n`. Let's analyze its time complexity:

The algorithm consists of a single loop that iterates over the array elements from `i = 1` to `n`. Within each iteration of the loop, a constant amount of operations (adding `a[i]` to `s`) is performed.

Since the loop iterates `n` times, the number of operations performed by the algorithm is directly proportional to the size of the input, which is `n`.

Therefore, we can conclude that the time complexity of the given algorithm is O(n) or linear time complexity. 📈

Q-32:Define deterministic and no-deterministic algorithms. [Pre_2:2_2019]

Answer:

Deterministic algorithm

A deterministic algorithm is an algorithm that, given the same input, always produces the same output and follows a predictable sequence of steps. In other words, it has a single, unambiguous behavior for any given input. The behavior of a deterministic algorithm can be completely determined in advance, and it does not involve any randomness or non-deterministic choices. These types of algorithms are commonly used in many areas of computer science and mathematics.

Non-deterministic algorithm

A non-deterministic algorithm is an algorithm that may exhibit different behaviors or produce different outputs for the same input on different runs. Non-determinism often involves elements of randomness or non-deterministic choices. These algorithms are particularly useful when searching for solutions in complex problem spaces, such as in certain branches of theoretical computer science and optimization. Non-deterministic algorithms are typically analysed in terms of their average-case or expected behaviour rather than their worst-case behavior.

Q-33:Describe commonly believed relationtionship among p,np,np-hard and np-complete problems. [Pre_2:2_2019]

Answer:

Certainly! The relationships among P, NP, NP-hard, and NP-complete problems are fundamental concepts in computational complexity theory. Here's an overview:

- P (Polynomial Time): This class refers to the set of problems that can be solved by a deterministic algorithm in polynomial time. In simple terms, these are problems that have efficient solutions. For example, sorting a list of numbers can be solved in polynomial time.

- NP (Nondeterministic Polynomial Time): This class refers to the set of decision problems for which the "yes" answers can be verified in polynomial time. In other words, if there is a proposed solution, it can be checked efficiently. However, finding the solution itself may not be efficient. NP contains P as a subset.

- NP-hard (Nondeterministic Polynomial-time hard): This class refers to a set of problems that are at least as hard as the hardest problems in NP. In other words, if you can solve an NP-hard problem in polynomial time, you can solve any problem in NP in polynomial time. However, NP-hard problems themselves may not necessarily be in NP. They can be intractable and have exponential time algorithms.

- NP-complete (Nondeterministic Polynomial-time complete): This class refers to the subset of NP problems that are the hardest problems in NP. A problem is NP-complete if it is both in NP and every problem in NP can be transformed into it by a polynomial-time reduction. In simpler terms, solving one NP-complete problem efficiently would mean solving all NP problems.

Q-34: What is an algorithm? Write down the key criteria of an algorithm. [Pre_2:2_2021]

Answer

Here are the key criteria of an algorithm:

Correctness: An algorithm should produce the correct output for all possible inputs. It should solve the problem it was designed for without any errors or exceptions.

Efficiency: An algorithm should be designed to execute efficiently, making the best use of available resources such as time and memory. It should provide a solution in a reasonable amount of time and with an acceptable level of resource usage.

Readability: An algorithm should be written in a clear and understandable manner so that it can be easily comprehended by other programmers. It should follow proper coding conventions, have meaningful variable names, and be well-structured.

Scalability: An algorithm should be scalable, meaning it should handle larger input sizes as efficiently as possible. As the input grows, the algorithm's performance and resource usage should not deteriorate significantly.

Maintainability: An algorithm should be designed in a way that allows for easy maintenance and future enhancements. It should be modular and well-organized, making it easier to modify, debug, and extend if necessary.

Optimality: In some cases, an algorithm may strive for optimality by finding the best or optimal solution to a problem. This criterion is particularly important for problems where efficiency and resource usage are critical.

Robustness: An algorithm should handle unexpected or erroneous inputs gracefully. It should be resilient to invalid or exceptional inputs and provide robust error handling mechanisms.

These criteria help evaluate and assess the quality of an algorithm, ensuring that it not only solves the problem correctly but also performs well, is maintainable, and meets the requirements of the intended application.

Q-35:Explain the idea behind the Icosian Game. [Pre(imp)_2:2_2021]

Answer:

The Icosian Game, also known as Hamilton's Icosian Game, is a mathematical puzzle invented by Sir William Hamilton in 1857. The game is played on a dodecahedron, which is a regular polyhedron with twelve faces, each shaped like a pentagon.

The objective of the Icosian Game is to find a Hamiltonian cycle on the dodecahedron. A Hamiltonian cycle is a path that visits every vertex of a graph exactly once and returns to the starting vertex. In the case of the Icosian Game, the vertices represent cities or landmarks, and the edges represent possible connections between them.

The Icosian Game involves moving along edges of the dodecahedron, visiting different vertices, and trying to find a path that connects all the vertices without visiting any vertex more than once. It is essentially a traversal problem, where the challenge is to find the path that satisfies the required conditions.

The Icosian Game has connections to the famous "Seven Bridges of Königsberg" problem, which was solved by Euler in the 18th century. Hamilton was inspired by Euler's work and created the Icosian Game as a variation of the problem on a different geometric structure.

Q-36: What are the steps required to implement Depth-First Search (DFS) for graph traversal. [Pre_2:2_2021]

Answer:Here are the steps to implement Depth-First Search (DFS) for graph traversal: 🔎

1️⃣ Start at a specific vertex or node in the graph.

2️⃣ Mark the current vertex as visited to keep track of which nodes have been explored.

3️⃣ Visit the current vertex and perform any desired operations on it.

4️⃣ Explore an unvisited adjacent vertex of the current vertex. You can choose any adjacent vertex.

5️⃣ If all adjacent vertices of the current vertex have been visited, backtrack to the previous vertex.

6️⃣ Repeat steps 3 to 5 until there are no more unvisited vertices in the graph.

The steps above describe the recursive implementation of DFS. You can also use a stack data structure to implement DFS iteratively. In that case, you would push the adjacent vertices onto the stack instead of making recursive calls.

Q-37:List the steps to implement BFS graph traversal. [Pre(imp)_2:2_2021]

Here are the steps to implement Breadth-First Search (BFS) graph traversal:

1. Choose a starting node.
2. Create a queue and enqueue the starting node.
3. Create a set or array to keep track of visited nodes.
4. While the queue is not empty:
    - Dequeue a node from the queue.
    - Mark the node as visited.
    - Process the node (based on the specific problem or task).
    - Enqueue all the unvisited neighbors of the node into the queue.
5. If there are unvisited nodes, repeat the process from step 4 for the next unvisited node.
6. Once the queue is empty, the BFS traversal is complete.

Q: Compare and contrast DFS (Depth-First Search) and BFS (Breadth-First Search). [Pre_2:2_2021]

Q-38: What are the differences between BFS and DFS traversal techniques? [Pre(imp)_2:2_2021]

Here are the differences between BFS (Breadth-First Search) and DFS (Depth-First Search) traversal techniques: 🔍

Order of traversal:

- BFS: Visits the nodes in a level-by-level fashion, starting from the root node or a specified starting point.

- DFS: Explores as far as possible along each branch before backtracking to the previous node.

Data structure used:

- BFS: Uses a queue to manage the nodes to be visited.

- DFS: Uses a stack (or recursion, which implicitly uses the call stack) to manage the nodes to be visited.

Memory usage:

- BFS: Tends to use more memory than DFS since it needs to store all the nodes at each level in the queue.

- DFS: Typically uses less memory than BFS since it only needs to store the path to the current node.

Completeness:

- BFS: Guarantees finding a solution if one exists, given that the graph is finite.

- DFS: May not find a solution if the search space is infinite or if the search is stuck in an infinite loop.

Time complexity:

- BFS: Has a time complexity of O(V + E), where V is the number of vertices and E is the number of edges in the graph.

- DFS: Has a time complexity of O(V + E), but its actual runtime depends on the branching factor and the depth of the search.

Applications:

- BFS: Useful for finding the shortest path between two nodes, exploring all neighbors of a node, and solving problems like finding all connected components in a graph.

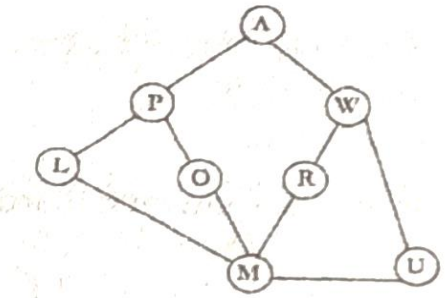
- DFS: Useful for finding paths between two nodes, exploring paths in a maze or a tree-like structure, and detecting cycles in graphs.

Important Keyword:

1. Discuss the concept of minimally connected and maximally connected graphs.
2. **Define acyclic graphs. Explain the connection between acyclic graphs and spanning trees.**
3. **Compare and contrast Kruskal's and Prim's algorithms in terms of their uses and efficiency.**
4. **Define indegree, outdegree, and directed acyclic graphs (DAGs).**
5. Explain why topological sort can only be performed on DAGs.
6. **Describe the concept of a Hamiltonian cycle in a graph and its application. Explain the Icosian Game and its connection to Hamiltonian cycles. Discuss the methods of drawing Hamiltonian cycles in a graph.**
7. **Define a subgraph, SCC, SCG, WCG.Describe how to find SCC and discuss its characteristics and applications.**
8. **Explain why conventional DFS cannot be used to find CSE //SCC and alternative approaches.**
9. Prove the conditions under which an Euler path and an Euler circuit are possible in a graph.
10. Discuss the concepts and applications the Fractional Knapsack problem.
11. Explain the Coin Change problem and its solution using dynamic programming.
12. Discuss the uses of Dijkstra's algorithm in real-world applications. Define Dijkstra's algorithm and explain its advantages over other algorithms for finding shortest paths.
13. **Discuss the drawbacks or limitations of the Bellman-Ford algorithm compared to other algorithms.**
14. Explain the Floyd-Warshall algorithm for finding the shortest paths between all pairs of vertices in a weighted graph.
15. Define a flow network and augmented path explain its components.
16. Discuss the relationship between maximum flow and minimum cut in a flow network.
17. **Explain the Edmonds-Karp algorithm for finding the maximum flow in a network.**
18. **Discuss the advantages and limitations of the Edmonds-Karp algorithm compared to other flow algorithms**.
19. Define a bipartite graph and explain its properties.

Graph Traversal:

Q-1: Explain BFS/DFS and give the traverse output step by step of the following graph. (Starting node, A). [Pre_2:2_2021]



Answer:

Here's a pseudocode representation of the BFS algorithm:

```
Procedure BFS(graph, startNode):
    Create an empty queue.
    Create a set to store visited nodes.

    Enqueue the startNode into the queue.
    Add the startNode to the visited set.

    While the queue is not empty:
        Dequeue a node from the queue.
        Process the node.

        For each neighbor of the node:
            If the neighbor has not been visited:
                Mark the neighbor as visited.
                Enqueue the neighbor into the queue.
```

Q-02:List the steps to color an undirected graph using the backtracking algorithm. [Pre(imp)_2:2_2021]

Answer:

To color an undirected graph using the backtracking algorithm, you can follow these steps:

1. Choose a starting node.
2. Assign a color to the starting node.
3. Move to the next uncolored node.
4. Check if the chosen color conflicts with any of its neighboring nodes.
    - If there is a conflict, choose a different color and go back to step 4.
    - If there is no conflict, assign the color to the node and move to the next uncolored node.
5. Repeat steps 4-5 for all uncolored nodes.
6. If all nodes are colored, the graph is successfully colored.
7. If there is no valid color assignment for any node, backtrack to the previous node and choose a different color.
8. Continue backtracking until a valid color assignment is found for all nodes or until all possibilities are exhausted.

Q-03:Whai is the idea behind vertex cover problems. [Pre_2:2_2019]

Answer:

The idea behind the vertex cover problem is to find a minimum set of vertices in a graph that cover all the edges. In other words, a vertex cover is a subset of vertices in a graph such that every edge in the graph is incident to at least one vertex in the subset.

The vertex cover problem is considered an optimization problem, and the goal is to find the smallest possible vertex cover in a given graph. It has practical applications in various fields, including network design, computer vision, and data mining.

Finding a vertex cover is an essential concept in graph theory because it helps identify the crucial nodes or vertices that play a significant role in maintaining connectivity or dependencies within a graph. By identifying the minimum vertex cover, we can potentially reduce the complexity and optimize the management and analysis of the graph.

To solve the vertex cover problem, various algorithms and techniques are employed, including brute-force search, dynamic programming, and approximation algorithms. The problem is known to be NP-complete, which means that there is no known polynomial-time algorithm that can solve it for all types of graphs. Therefore, in practice, finding a vertex cover often involves

applying heuristic or approximation algorithms to achieve an optimal or near-optimal solution efficiently.

Q-04:What do you know about the prefix rule in graph theory? [Pre(imp)_2:2_2021]

Answer:

In graph theory, the prefix rule, also known as the prefix property or the prefix code property, refers to a property of codes or labels assigned to the vertices of a rooted tree. A rooted tree is a type of graph where one vertex, called the root, is distinguished from other vertices.

The prefix rule states that if we assign codes or labels to the vertices of a rooted tree, such that each vertex is labeled with a unique string of symbols (e.g., binary digits), then the labels along the path from the root to any other vertex should form a prefix of the label on that vertex.

In other words, the label assigned to a vertex should be a concatenation of labels from the root to that vertex, without any other label forming a prefix in between. This ensures that no label is a prefix of another label in the tree, making it possible to uniquely decode the labels and navigate the tree structure.

The prefix rule is commonly applied in various areas where hierarchical structures or tree-like data representations are used, such as in coding theory, data compression, and network routing algorithms. By honoring the prefix rule, efficient encoding and decoding schemes can be designed, where the labels represent meaningful information or instructions associated with the vertices of the tree.

Shortest Path Algorithms:

Q-01:Write down the characteristics of Huffman Coding. [Pre(imp)_2:2_2021]

Answer:

Characteristics of Huffman Coding:

> **Lossless Compression:** Huffman coding is a lossless data compression technique, meaning it can compress data without losing any information. Every compressed bitstream can be perfectly decoded back to the original data.
>
> **Variable-Length Encoding:** It assigns shorter codes to more frequent symbols and longer codes to less frequent symbols. This minimizes the average codeword length and reduces the overall size of the compressed data.
>
> **Prefix Codes:** Huffman codes are prefix codes, where the code assigned to one symbol is never a prefix of the code assigned to any other symbol. This guarantees unique interpretation during decoding and avoids ambiguity.

Greedy Algorithm: The algorithm used to construct the Huffman tree (the basis for code assignment) is a greedy algorithm. It repeatedly merges the two nodes with the lowest frequencies, effectively allocating shorter codes to the most frequent symbols.

Optimality: For a given symbol distribution, Huffman coding produces the closest possible approximation to the theoretical entropy limit of the data, making it optimal in terms of average codeword length.

Flexibility: Huffman coding can be applied to any type of data (text, images, audio, etc.) as long as the symbol frequencies can be determined.

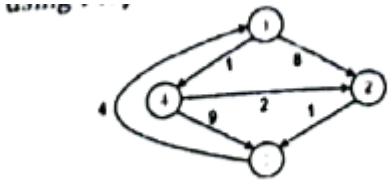
Simple Implementation: The algorithm for constructing the Huffman tree and encoding/decoding data is relatively simple and efficient, making it suitable for various implementations.

Wide Applications: Huffman coding finds applications in diverse fields like data compression, archiving, transmission, image and audio codecs, file formats, and more.

It's important to note some limitations:

- Overhead: The need for a codebook (mapping symbols to codes) adds a small overhead to the compressed data.
- Performance: Encoding and decoding may require more processing power compared to simpler compression techniques.
- Not efficient for uniform probability: If all symbols have approximately the same frequency, Huffman coding may not achieve significant compression.

Q-02:Find all pair of shortest path using Floyd's Algorithms for given graph: [Pre(imp)_2:2_2021]



Q-03:What are the 4 major differences between Bellman's ford and Dijkstra's algorithms. [Pre_2:2_2019]

There are actually five major differences between Bellman-Ford and Dijkstra's algorithms:

Handling Negative Edge Weights:

- Bellman-Ford: Works with graphs containing negative edge weights and can detect negative cycles (cycles where the sum of edge weights is negative).
- Dijkstra's: Cannot handle negative edge weights and will result in incorrect or infinite distances if used in such graphs.

Time Complexity:

- Bellman-Ford: Has a time complexity of O(V * E) (V is vertices, E is edges), making it slower than Dijkstra's for non-negative graphs.
- Dijkstra's: Has a time complexity of O(E log V) using a priority queue, making it significantly faster for non-negative graphs.

Greedy vs. Dynamic Programming:

- Bellman-Ford: Uses a dynamic programming approach, iteratively relaxing edges and updating distances until convergence.
- Dijkstra's: Uses a greedy approach, progressively building the shortest path tree by choosing the vertex with the smallest tentative distance at each step.

Applicability:

- Bellman-Ford: More versatile due to its ability to handle negative weights, useful for routing protocols and shortest paths with tolls.
- Dijkstra's: Simpler, faster for non-negative graphs, commonly used for finding shortest paths in road networks or airfares.

Distributed Implementation:

- Bellman-Ford: Can be easily implemented in a distributed way, as each node can update its information based on its neighbors.
- Dijkstra's: More challenging to implement in a distributed setting due to the centralized management of the priority queue.

Q-04:Write down the pseudo code of Bellman's ford algorithms. What a negative cycle is, providing an example if possible. [Pre_2:2_2019]

Answer:

```
function BellmanFord(vertices, edges, source):
    distances = array of size vertices, initialized with infinity
```

```
    distances[source] = 0

    // Relax edges |vertices|-1 times
    for i from 1 to vertices-1:
        for each edge (u, v, weight) in edges:
            if distances[u] + weight < distances[v]:
                distances[v] = distances[u] + weight

    // Check for negative weight cycles
    for each edge (u, v, weight) in edges:
        if distances[u] + weight < distances[v]:
            // Negative cycle detected
            return "Graph contains negative weight cycles"

    return distances
```

Negative Cycle:

- A negative cycle is a cycle in a directed graph where the sum of the edge weights is negative.
- It's problematic for shortest path algorithms because it means you can continuously traverse the cycle, reducing the total distance infinitely.

Example:

Consider this graph:

```
A → B (-2)
B → C (-1)
C → A (-1)
```

There's a negative cycle A → B → C → A with a total weight of -4. If you start at A, you can keep going around this cycle, reducing the distance to any other vertex indefinitely. Bellman-Ford will detect this negative cycle and indicate that shortest paths cannot be computed.

Q-05:Define feasible solution and optimal solution. Write down the greedy algorithms to generate shortest path. [Pre_2:2_2019]

Answer:

Feasible Solution:

A solution that satisfies all the constraints of a problem, but it might not be the best possible solution in terms of the objective function. It's a valid solution that adheres to the problem's rules and requirements.

Optimal Solution:

The best possible solution among all feasible solutions, providing the most desirable value for the objective function. It's the solution that maximizes or minimizes the goal of the problem, depending on the context.

Greedy Algorithms for Shortest Path:

Dijkstra's Algorithm:

- Applies to: Graphs with non-negative edge weights.
- Approach:
    - Start with a set of unvisited vertices.
    - Repeatedly choose the unvisited vertex with the shortest tentative distance from the source.
    - Update the distances of its neighbors if shorter paths are found.
    - Continue until all vertices are visited.

Prim's Algorithm:

- Applies to: Finding a minimum spanning tree (MST) in a graph.
- Approach:
    - Start with an empty MST.
    - Repeatedly add the cheapest edge that connects a vertex in the MST to a vertex outside the MST.
    - Continue until all vertices are included in the MST.

Kruskal's Algorithm:

- Approach:
    - Sort all edges in the graph in ascending order of weight.
    - Iterate through the sorted edges:
        - If adding the current edge doesn't create a cycle, add it to the MST.
            - Otherwise, skip it.
            - Continue until the MST has V-1 edges (where V is the number of vertices).

Q-06: Can Dijkstra's algorithm be applied to a graph with negative edges? Compare Dijkstra's and Bellman Ford's algorithms, which one is better and how. [Pre_2:2_2021]

Answer:

No, Dijkstra's algorithm cannot be directly applied to a graph with negative edges. Dijkstra's algorithm assumes that all edge weights are non-negative. Negative edges can lead to incorrect calculations and infinite loops in Dijkstra's algorithm. However, there are variations of Dijkstra's algorithm, like the A* algorithm, that can handle graphs with some negative edge weights or heuristics that guide the search.

On the other hand, Bellman-Ford's algorithm is designed to handle graphs with negative edge weights. It is an algorithm for single-source shortest path problem, just like Dijkstra's algorithm, but it can handle graphs with negative edges (as long as there are no negative cycles).

Bellman-Ford's algorithm iteratively relaxes the edges of the graph and updates the distances until it reaches the optimal solution. In each iteration, it relaxes all the edges in the graph, and it repeats this process for V-1 iterations, where V is the number of vertices in the graph. This ensures that the algorithm has considered all possible paths and found the shortest path.

Comparing the two algorithms:

Dijkstra's algorithm:

Advantages:

- Efficient for graphs with non-negative edge weights.
- Typically faster than Bellman-Ford's algorithm on dense graphs.

Limitations:

- Cannot handle graphs with negative edge weights without modifications.
- Requires non-negative edge weights to guarantee correctness.
- Doesn't detect negative cycles.

Bellman-Ford's algorithm:

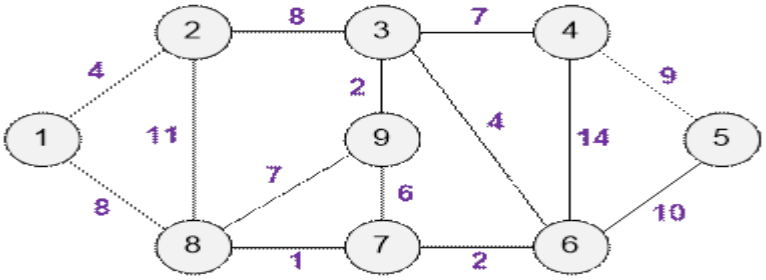
Advantages:

- Can handle graphs with negative edge weights (as long as there are no negative cycles).
- Identifies negative cycles in the graph.

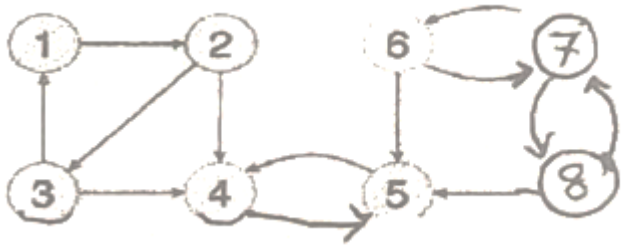
Limitations:

- Less efficient than Dijkstra's algorithm, especially on dense graphs (O(V * E) time complexity).
- Requires V-1 iterations to be completed to guarantee the shortest paths.

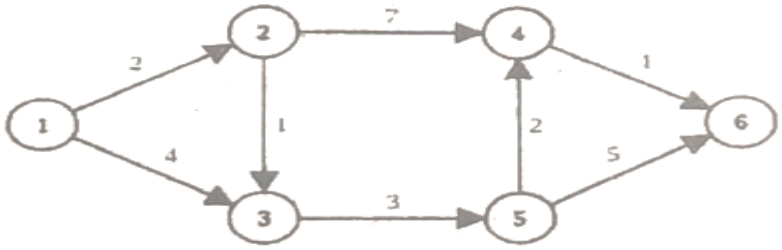
Q-08: Find the shortest path for all edges within a given graph Using Dijkstra's Algorithm. Consider 0 as the source vertex. [Pre_2:2_2021]



Q-08: Define strongly connected components (SCC). Apply the process to identify strongly connected components in a graph, starting with vertices ->1 [Pre_2:2_2021]



Q-09: Use Dijkstra's algorithm to determine the shortest weighted path from vertex '1' to every other vertex in a specified graph with edges: [Pre_2:2_2021]



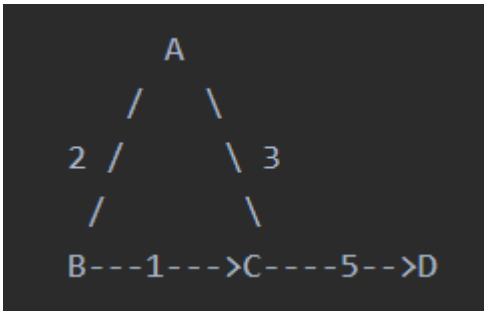
Q-10: Define relaxation with an example. [Pre_2:2_2021]

Answer:

Relaxation, in the context of shortest path algorithms, refers to updating the current estimate of the shortest distance to a vertex from the source vertex. It involves considering a potential new path to that vertex and updating the distance if a shorter path is found.

Let's consider an example to illustrate relaxation:

Suppose we have the following graph representation:



Let's say we want to find the shortest path from vertex A to vertex D using a shortest path algorithm like Dijkstra's algorithm or Bellman-Ford's algorithm.

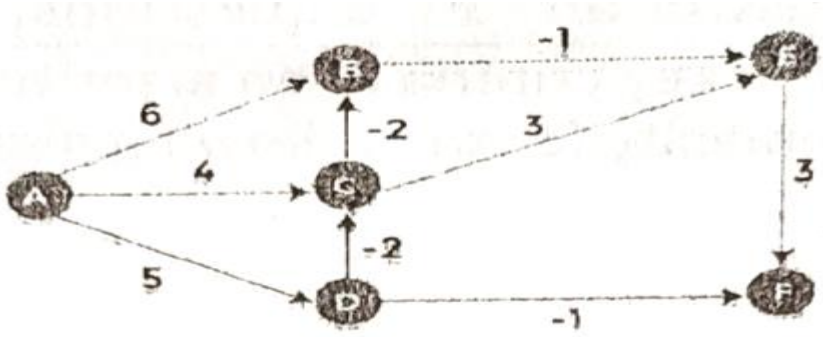
Initially, the estimated distance to all vertices except the source is set to infinity. The estimated distance to the source vertex itself is set to 0.

1. Initialization:
    - Distance[A] = 0
    - Distance[B] = infinity
    - Distance[C] = infinity
    - Distance[D] = infinity
2. Relaxation:
    - We start with vertex A and examine its neighboring vertices B and C.
        - Vertex B is reachable from A with an edge weight of 2. So, we compare the current estimate of the distance to B with the distance from A plus the edge weight (0 + 2 = 2).
        - Since 2 is smaller than infinity (the initial estimate), we update the estimate of the distance to B as 2.
    - We move to vertex C and examine its neighboring vertex D.
        - Vertex D is reachable from C with an edge weight of 5. So, we compare the current estimate of the distance to D with the distance from C plus the edge weight (infinity + 5 = infinity).
        - Since infinity is greater than infinity (the initial estimate), we don't update the estimate of the distance to D.
3. Final state after relaxation:
    - Distance[A] = 0
    - Distance[B] = 2
    - Distance[C] = infinity
    - Distance[D] = infinity

In this example, relaxation involved updating the estimate of the distance to vertex B based on the current path, and leaving the estimates of distances to vertices C and D unchanged because the current path did not improve those distances.

Relaxation is a key step in shortest path algorithms as it allows the algorithm to continuously refine the estimates of shortest distances until the optimal solution is found.

Q-11: Use the Bellman Ford Algorithm to find the shortest path for all edges in a given graph, considering vertex 'A' as the source. [Pre_2:2_2021]



Answer:

Q-12:Define residual capacity and augmenting path. Compare the Ford-Fulkerson algorithm with the Edmonds-Karp algorithm, emphasizing their main differences. [Pre_2:2_2021]

Answer:

Residual Capacity: In the context of flow networks, residual capacity refers to the maximum amount of additional flow that can be pushed through an edge in a given network. It represents the remaining capacity for flow in the forward direction or the potential capacity for flow in the reverse direction. It helps determine the path along which flow can be augmented.

Augmenting Path: An augmenting path is a path in a flow network that has available residual capacity on every edge in the path. It allows for increasing the flow from the source to the sink in order to optimize the maximum flow.

let's compare the Ford-Fulkerson algorithm with the Edmonds-Karp algorithm:

Ford-Fulkerson Algorithm: The Ford-Fulkerson algorithm is a generic approach to solve the maximum flow problem. It iteratively finds augmenting paths and increases the flow along those paths until no augmenting path can be found anymore. There are different strategies to choose the augmenting paths, such as depth-first search or breadth-first search. The algorithm terminates when no more augmenting paths exist.

Edmonds-Karp Algorithm: The Edmonds-Karp algorithm is a specific implementation of the Ford-Fulkerson algorithm. It uses breadth-first search (BFS) to choose the augmenting paths, guaranteeing that the shortest augmenting path is found in terms of the number of edges. By using BFS, the algorithm exploits the concept of minimum cut, which ensures that the time complexity is bounded by O(V * E^2), where V is the number of vertices and E is the number of edges. This makes the Edmonds-Karp algorithm more efficient than the generic Ford-Fulkerson algorithm in terms of time complexity.

Main Differences:

1. Pathfinding Method:
    - Ford-Fulkerson is more general and doesn't specify a pathfinding method.
    - Edmonds-Karp explicitly uses BFS to find augmenting paths.
2. Efficiency:
    - Edmonds-Karp is generally more efficient than a simple implementation of Ford-Fulkerson due to its use of BFS.
3. Convergence:
    - Both algorithms guarantee convergence to the maximum flow.

Divide & Conquer:

Q-01:What are the basic themes of the divide and conquer method? Write down the merge sort algorithm. [Pre(imp)_2:2_2021 &2019]

Answer:

The basic themes of the divide and conquer method are:

Divide: The problem is divided into smaller subproblems that are similar to the original problem. This step involves breaking down the problem into smaller, more manageable parts.

Conquer: The subproblems are solved recursively. Each subproblem is solved independently, either by directly solving it or by applying the divide and conquer method iteratively.

Combine: The solutions to the subproblems are combined to obtain the solution to the original problem. This step involves merging or integrating the solutions of the subproblems into a single solution.

Merge Sort Algorithm:

Merge sort is an efficient, comparison-based sorting algorithm that follows the divide and conquer approach. It sorts an array by dividing it into two halves, sorting each half recursively, and then merging the sorted halves to obtain a sorted array.

Here's the pseudocode for the merge sort algorithm:

1. Define a function mergeSort(arr) that takes an array as input.
2. If the length of the array is less than or equal to 1, return the array as it is already sorted.
3. Divide the array into two halves, left and right.
4. Recursively apply mergeSort to the left half and store the sorted result in leftSorted.
5. Recursively apply mergeSort to the right half and store the sorted result in rightSorted.
6. Merge the sorted left and right halves using the following steps:
   a. Initialize an empty array result[] to store the merged result.
   b. Compare the elements of leftSorted and rightSorted in order.
   c. Append the smaller element to the result array.
   d. Repeat steps b and c until either leftSorted or rightSorted becomes empty.
   e. Append the remaining elements of the non-empty array to the result.
7. Return the merged result array.

Q-02:List the factors that affect the running time of an algorithm. Describe the step-by-step process of finding 151, -14, and 9 from the following list of elements using iterative binary search algorithms : -15,6,0,7,9,23,54,82,101,112,125,131,142,151. [Pre(imp)_2:2_2021 & 2019]

Answer:

Factors affecting the running time of an algorithm:

Input Size: The number of elements in the input data significantly impacts the running time. Larger inputs generally require more processing.

Algorithm Design: The efficiency of the algorithm's logic and operations plays a crucial role. Some algorithms are inherently faster than others for specific tasks.

Hardware: The processor speed, memory capacity, and other hardware specifications influence execution speed.

Programming Language and Implementation: The choice of programming language and the way the algorithm is implemented can affect performance due to factors like compilation time and language features.

Data Structure: The type of data structure used to store and organize the data can impact how efficiently the algorithm can access and manipulate it.

Iterative Binary Search Algorithm for 151:

Steps:

1. Initialize:
    - Set low = 0 and high = 13 (indices of the first and last elements).
    - Set target = 151.
2. Iterate:
    - While low <= high:
        - Calculate mid = (low + high) // 2.
        - If list[mid] == target:
            - Element found at index 13.
            - Exit the loop.
        - If list[mid] < target:
            - Set low = mid + 1 (search in the right half).
        - If list[mid] > target:
            - Set high = mid - 1 (search in the left half).
3. Not Found:
    - If the loop ends without finding the target, the element is not in the list.

Iterative Binary Search Algorithm for -14:

Steps:

1. Initialize:
    - Set low = 0 and high = 13.

- Set target = -14.

2. Iterate:
    - While low <= high:
        - Calculate mid = (low + high) // 2.
        - If list[mid] == target:
            - Element found at index 0.
            - Exit the loop.
        - If list[mid] < target:
            - Set low = mid + 1.
        - If list[mid] > target:
            - Set high = mid - 1.
3. Not Found:
    - The loop ends without finding -14, as it's not in the list.

Iterative Binary Search Algorithm for 9:

Steps:

1. Initialize:
    - Set low = 0 and high = 13.
    - Set target = 9.
2. Iterate:
    - While low <= high:
        - Calculate mid = (low + high) // 2.
        - If list[mid] == target:
            - Element found at index 4.
            - Exit the loop.
        - If list[mid] < target:
            - Set low = mid + 1.
        - If list[mid] > target:
            - Set high = mid - 1.
3. Not Found:
    - The loop ends without finding -14, as it's not in the list.

Q-03: Explain the Counting Inversion method utilizing merge sort. [Pre_2:2_2021]

Answer:

The counting inversion method is a technique used to count the number of inversions in an array. An inversion occurs when two elements in an array are out of order relative to each other. For example, in the array [7, 5, 3, 2], there are three inversions: (7, 5), (7, 3), and (7, 2).

One efficient way to implement the counting inversion method is by utilizing the merge sort algorithm. Merge sort is a divide-and-conquer algorithm that recursively divides an array into smaller subarrays, sorts them, and merges them back together.

Here's how you can implement the counting inversion method using merge sort:

1. Write a function `countInversions` that takes an array `arr` as input and returns the number of inversions in the array.

2. Implement the `mergeSort` function that performs the merge sort algorithm:

- Base case: If the length of `arr` is 0 or 1, return `arr`.

- Divide `arr` into two halves: `left` and `right`.

- Recursively call `mergeSort` on `left` and `right` to sort them.

- Merge the sorted `left` and `right` arrays while counting the inversions. Initialize a variable `inversions` as 0.

- Create an empty result array.

- Compare the first elements of `left` and `right`. If the element in `right` is smaller, it forms an inversion.

- Increment `inversions` by the number of remaining elements in `left` (since they are all greater than the current element in `right`).

- Append the smaller element to the result array.

- Repeat the comparison and merging process until either `left` or `right` becomes empty.

- Append the remaining elements of `left` and `right` to the result array (no inversions are formed here).

- Return the result array and the total number of inversions (`inversions`).

3. In the `countInversions` function, call the `mergeSort` function on the input array `arr` and extract the total number of inversions.

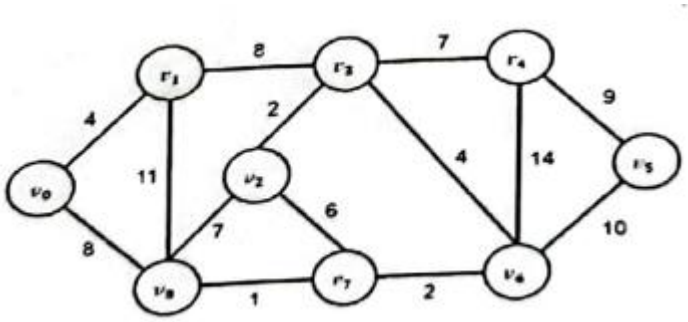
4. Return the total number of inversions.

By utilizing the merge sort algorithm, we can efficiently count the inversions in an array in O(n log n) time complexity, where n is the size of the array.

Greedy Algorithms:

Q-09:What is the minimum cost spanning tree? Compute a minimum spanning tree using prim's algorithms for the following graph: [Pre_2:2_2019]

Answer:



Q-10: Solve the following 0/1 Knapsack problem by using dynamic programming.

Answer:

Q-11: Determine the optimal solution for a fractional knapsack instance where n=3 (number of items), profits (P1, P2, P3) are (25, 24, 15), and weights (W1, W2, W3) are (18, 15, 10).

Answer:

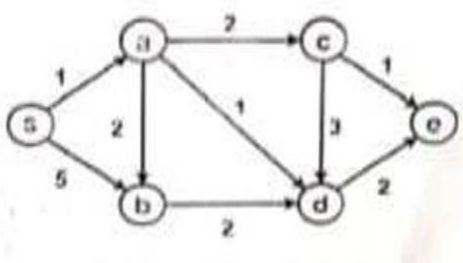
Extra:

Q-12: Explain the Traveling Salesperson Problem (TSP) and solve it using a provided matrix/graph, assuming the root is 1.

Answer:

The Traveling Salesperson Problem (TSP) is a classic optimization problem where a salesperson needs to visit a set of cities exactly once and return to the starting city, minimizing the total distance traveled. It is an NP-hard problem. To solve it, you can use algorithms like the Held-Karp dynamic programming approach or the 2-Opt local search heuristic.

Q-13: Write a greedy algorithm for the single-source shortest path problem and determine the shortest path for a specified graph.



Q-14: Provide the pseudo code for the GREEDY ACTIVITY SELECTOR algorithm.

Answer:

```
GreedyActivitySelector(startTime, finishTime):
    n = length of startTime
    selectedActivities = [1]  # Include the first activity
    lastSelected = 1
    for i = 2 to n:
        if startTime[i] >= finishTime[lastSelected]:
            selectedActivities.append(i)
            lastSelected = i
    return selectedActivities
```

Q-15: Given activities with their starting and finishing times, compute a schedule with the maximum number of activities using the Activity Selection Problem.

The Activity Selection Problem aims to schedule the maximum number of non-overlapping activities given their start and finish times. The greedy algorithm for this problem (Activity Selector) sorts the activities by finish times and iteratively selects activities with the earliest finish time that do not overlap with the previously selected activities.

Q-16: Enumerate real-life applications of the Activity Selection Problem.

Real-life applications of the Activity Selection Problem include:

- Scheduling classes or lectures in a school or university.
- Planning time slots for conference presentations or talks.
- Scheduling tasks or processes in operating systems.
- Organizing sports events or tournaments with limited resources.

Q-17: Differentiate between Kruskal's and Prim's Spanning Tree algorithms. [Pre_2:2_2021]

Kruskal's and Prim's algorithms are used to find minimum spanning trees in a graph:

- Kruskal's algorithm builds the minimum spanning tree (MST) incrementally by adding the shortest edge to the MST if it doesn't create a cycle. It uses a sorting step to order the edges by weight.

- Prim's algorithm starts with an arbitrary vertex and grows the MST by adding the shortest edge connected to the current MST. It uses a priority queue to select the next minimum weight edge.

Q-18:Define a minimum-cost spanning tree. Utilize Kruskal's algorithm to find the minimum spanning tree for a given graph.

A minimum-cost spanning tree is a tree that connects all vertices of a graph with the minimum total weight/cost. Kruskal's algorithm for finding a minimum spanning tree in a given graph can be summarized as follows: Sort all the edges by weight, iterate over them, and add each edge to the MST if it doesn't create a cycle. The process continues until all vertices are included, resulting in the minimum spanning tree.

Dynamic Programming:

Q-10: Write down the Algorithm of 0/1 Knapsack Problem[Pre_2:2_2021]

Q-11: Explain what is LCS (Longest Common Subsequence). Describe the process of finding the LCS for the two strings when str2="abcdbdc" using dynamic programming.

Q-12: Explain dynamic programming. Calculate the minimum cost path from 0 to 7 using a forward approach in the provided multistage graph.

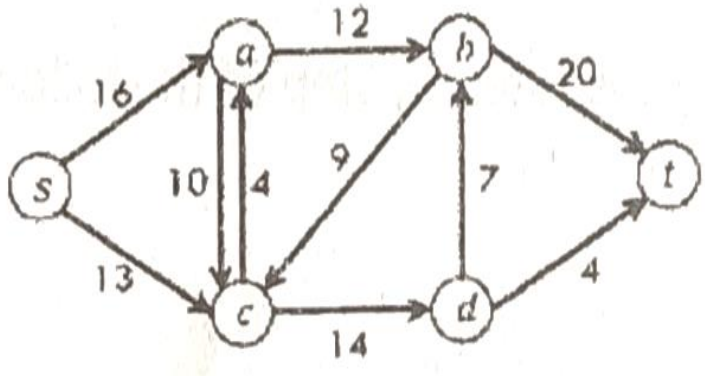
Q-13: Enumerate the applications of the Longest Common Subsequence (LCS) algorithm. [Pre_2:2_2021]

Q-14:Given two strings, determine the length of the common subsequence and identify three longest common subsequences (LCS). [Pre_2:2_2021]

X=A,B,C,D,A,B Y=B,D,C,A,B,A

Network Flow:

Q-1: Calculate the maximum flow and minimum cut capacity in the provided graph. [Pre_2:2_2021]



Q-2: Define terms related to flow networks: Flow network, Residual graph, Augmented path, Residual capacity.

Q-3: Explain what approximation means. Describe the approximation algorithm for the vertex cover problem with a relevant example.

MD RAIHAN ALI_[63]
CSE_02
Faculty of Engineering & Technology
University of Dhaka (NITER)