School of Computing

Laboratory Manual

Course Code: CSE319
Course Name: Algorithm Design Strategies & Analysis Laboratory

List of Experiments

0. **Analyzing Algorithms**
   a. Insertion Sort – Time Complexity Analysis
   b. Merge Sort Algorithms – Time Complexity Analysis
1. **Applications of Heuristic Method**
   a. Travelling Salesman Problem
2. **Applications of Greedy Method**
   a. Fractional Knapsack Problem
   b. Job Sequencing with Deadlines
3. **Applications of Dynamic Programming**
   a. Optimal Binary Search Trees
   b. 0/1 Knapsack Problem
4. **Applications of Branch & Bound**
   a. Travelling Salesman Problem
   b. 0/1 Knapsack Problem
5. **Applications of Backtracking**
   a. 8 Queens Problem
   b. Sum of Subset Problem
6. **Programs on Graphs – Application of DFS**
   a. Topological Sort of Directed Acyclic Graph
7. **Programs on Graphs – Minimum Spanning Tree**
   a. Prim's Algorithm
   b. Kruskals's Algorithm
8. **Programs on Graphs – Shortest Path Algorithms**
   a. Single Source Shortest Paths using Bellman-Ford algorithm
9. **Programs on Graphs – Shortest Path Algorithms**
   a. All-Pairs Shortest Paths using Floyd-Warshall algorithm
10. **Programs on Graphs – Network Flow Problem**
   a. Maximum Flow using Ford Fulkerson Method

Experiment Number: 00

Analyzing Algorithms

Insertion Sort & Merge Sort Algorithms – Time Complexity Analysis

Aim:

To implement insertion sort algorithm and merge sort algorithm. To compare their time complexity by counting the total number of active operations present in the algorithm. To compare the rate of growth of algorithms for various input sizes like n=1000, n=2000 and n=5000. To analyze the complexity for various types of inputs such as, (i) Ordered Elements (ii) Reverse Ordered Elements and (iii) Random Elements.

Algorithm(s):

(a) Insertion Sort

```
Algorithm InsertionSort(A[0..n-1])
    For j ← 1 to n-1 do
        Key ← A[j]
        i ← j-1
        While i>-1 and A[i]>Key do
            A[i+1] ← A[i]
            i ← i-1
        End While
        A[i+1] ← Key
    End For
End InsertionSort
```

(b) Merge Sort

```
Algorithm MergeSort(A[0..n-1], p, r)
    If p>=r then
        Return
    End If
    q ← ⌊( p + r ) / 2⌋
    MergeSort(A, p, q)
    MergeSort(A, q+1, r)
    Merge(A, p, q, r)
End MergeSort
```

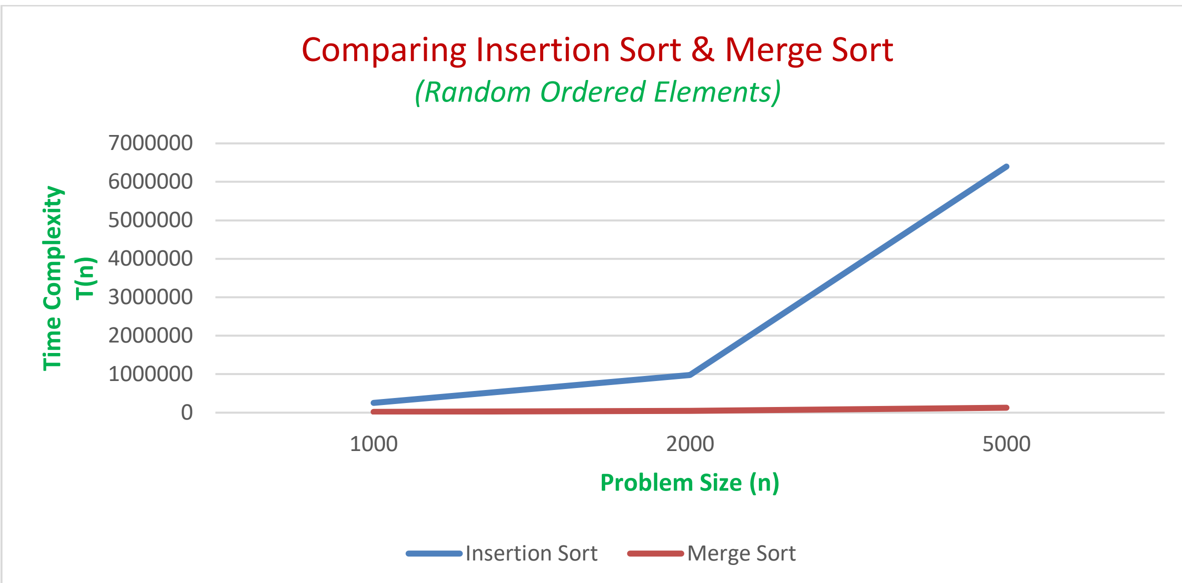
```
Algorithm Merge(A[0..n-1], p, q, r)
        n1 ← q - p + 1
        n2 ← r - q
        Let L[0..n1] and R[0..n2] be new arrays
        i←j←0
        For k←p to r do
                If k ≤ n1 then
                        L[i] ← A[k]
                        i ← i + 1
                Else
                        R[j] ← A[k]
                        j ← j + 1
                End If
        End For
        L[i] ← R[j] ← ∞
        i←j←0
        For k←p to r do
                If L[i]<=R[j] then
                        A[k] ← L[i]
                        i ← i + 1
                Else
                        A[k] ← R[j]
                        j ← j + 1
                End If
        End For
End MergeSort
```

Results & Discussion:

Comparison Table

Size (n)	Number of Active Operations					
	Ordered Elements		Reverse Ordered Elements		Random Ordered Elements	
	Insertion Sort	Merge Sort	Insertion Sort	Merge Sort	Insertion Sort	Merge Sort
1000	999	20951	499969	20951	254383	20951
2000	1999	45903	1999968	45903	978987	45903
5000	4999	128615	12499890	128615	6397319	128615

Comparison Chart



(***Note:*** Includes similar graphs for Ordered Elements and Reverse Ordered Elements)

The following points are inferred from this experiment.

1. For random ordered elements and reverse ordered elements, Merge Sort gives better performance than Insertion Sort.
2. For Ordered elements, Insertion Sort gives better performance than Merge Sort.

Experiment Number: 01

Applications of Heuristic Method
Travelling Salesman Problem

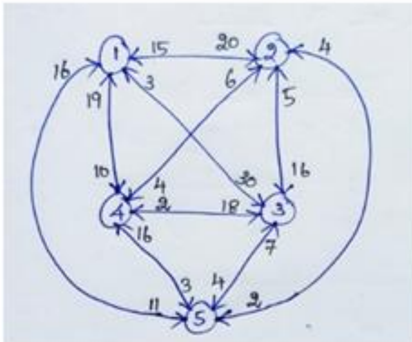
Aim:

(a) Given a set of cities and distance between each pair of cities, the problem is to find the shortest possible trip to visit every city exactly once and returns to the starting city. The cities are mapped with the vertices and the distance between the pairs are mapped with the edge cost of a graph. The aim is to design and demonstrate an algorithm for solving the TSP by using heuristic method.

```
Algorithm TSP(Cost[1..n][1..n], n, S)
    Input:      n – Number of Vertices
                S – Starting Vertex
                Cost[1..n][1..n] - 'n x n' Cost matrix.
    Output:     Shortest Tour and the Cost

    //Store all the vertices except the starting vertex into a list
    For i←1 to n Do
        If i<>S Then
            Vertex[i] ← i
        End If
    End For

    //Let the initial minPath as infinity
    minPath ← ∞

    //Repeat until there is no other permutations
    While NextPermutation(Vertex, n) Do
        currentPathWeight ← 0
        K ← S
        For i←1 to n Do
            currentPathWeight = currentPathWeight + Cost[K][Vertex[i]]
            K ← Vertex[i]
        End For
        currentPathWeight = currentPathWeight + Cost[K][S]
        minPath = Min(minPath, currentPathWeight)
    End While

    Return minPath
End TSP
```

Sample Input & Output:

Input Graph



$$
\begin{array}{c|ccccc}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & \infty & 20 & 30 & 10 & 11 \\
2 & 15 & \infty & 16 & 4 & 2 \\
3 & 3 & 5 & \infty & 2 & 4 \\
4 & 19 & 6 & 18 & \infty & 3 \\
5 & 16 & 4 & 7 & 16 & \infty
\end{array}
$$

Result

Alive Path is: 1 - 4 - 6 - 10 - 11

Tour for Salesperson is: 1 - 4 - 2 - 5 - 3 - 1

Cost of the Tour: 28

Experiment Number: 02

Applications of Greedy Approach

Fractional Knapsack Problem & Job Sequencing Problem

Aim:

To demonstrate the following two applications of dynamic programming approach.

(a) Fractional Knapsack Problem - Given a set of 'n' items, Items[1..n], each item is having weight & profit. In fractional knapsack, the given items can be divisible. i.e. It is possible to select part of an item. Given a knapsack with maximum weight capacity W. The aim is to fill the knapsack with the selected items for the maximum capacity such that the total profit should be maximum.

(b) Job Sequencing with Deadline Problem: Given an array of jobs where every job has a deadline and associated profit if the job is finished before the deadline. It is also given that every job takes a single unit of time, so the minimum possible deadline for any job is 1. The objective is to find a sequence of jobs, which is completed within their deadlines and gives maximum profit.

Algorithm(s):

(a) Fractional Knapsack Problem

Algorithm FractionalKnapsack(Objects[1..n], n, C, SelectedObjects[1..m])

Input: Objects[1..n] – List of 'n' objects each with Profit & Weight
C – Capacity of Knapsack
n – Number of Items

Output: SelectedObjects[1..m] & Maximum Profit

```
//Calculating Profit Per Weight Ratio
For i←1 to n do
        Objects[i].PW ← Objects[i].P / Objects[i].W
End For

//Sort the Objects[1..n] in descending order of its PW
SortDescendingPW(Objects, n)

m ← 0
For i←1 to n do
        If C=0 then
                Return m
        End If
```

```
            If Objects[i].W <= C then
                m ← m + 1
                SelectedObjects[m] ← Objects[i]
                C ← C – Objects[i].W
            Else
                Obj ← Objects[i]
                Obj.W ← C
                Obj.P ← Obj.P * (Obj.W / Objects[i].W)
                m ← m + 1
                SelectedObjects[m] ← Obj
                C ← 0
            End If
        End For
        Return m;
End FractionalKnapsack
```

(b) Job Sequencing with Deadline

```
Algorithm JobSequencingGreedy(Jobs[1..n], n, Slots[1..m])
        Input:      Jobs[1..n] – List of 'n' objects each with Profit & Deadline
                    n – Number of Items
        Output:     Slots[1..m] & the size m

        //Finding Maximum Slots
        m ← Maximum(Jobs[1..n].Deadline)

        //Sorting the Jobs[1..n] in descending order of its Profit
        SortDescendingProfit(Jobs, n)

        Let Slots[1..m] to maintain the order of Jobs to be done.

        For i←1 to m do
            Slots[i] ← 0
        End For

        For i←1 to n do
            For j←Jobs[i].Deadline to 1 downwards do
                If Slots[j]=0 then
                    Slots[j] ← i
                    Break from Loop
                End If
            End For
```

End For
Return m
End JobSequencingGreedy

Results & Discussion:

Sample Input & Output

(a) Fractional Knapsack Problem

Input:

No. of Items: n = 7							
Item	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Weight[1..7]	2	3	5	7	1	4	1
Profit[1..7]	10	5	15	7	6	18	3
Bag Capacity: W = 15							

Output:

```
Bag Capacity: 15
Number of Available Objects: 7

Available Objects:

        Item No Profit  Weight
        1       10      2
        2       5       3
        3       15      5
        4       7       7
        5       6       1
        6       18      4
        7       3       1

Selected Objects:

        Item No Profit  Weight  Profit Per Weight
        5       6       1       6
        1       10      2       5
        6       18      4       4.5
        3       15      5       3
        7       3       1       3
        2       3.33333 2       1.66667

Total Profit: 55.3333
```

(b) Job Sequencing with Deadline

Input:

n=8	Jobs With Profit & Deadlines							
Jobs	1	2	3	4	5	6	7	8
Profits	67	53	42	39	31	24	18	5
Deadlines	4	5	5	3	2	1	3	2

Output:

```
Number of Jobs: 8

Jobs with Deadline:

        Job No. Profit  Deadline
        1       67      4
        2       53      5
        3       42      5
        4       39      3
        5       31      2
        6       24      1
        7       18      3
        8       5       2

Maximum Slots Available: 5

Scheduled Jobs:

        Slot No Job No. Profit
        1       5       31
        2       4       39
        3       3       42
        4       1       67
        5       2       53
```

Experiment Number: 03

Applications of Dynamic Programming

Optimal BST & 0-1 Knapsack Problem

Aim:

To demonstrate the following two applications of dynamic programming approach.

(a) Optimal BST - Given a set of 'n' elements Key[1..n] and the frequency of searching elements includes successful search probability list P[1..n] and unsuccessful search probability list Q[0..n]. Problem is to construct the Optimal BST for the given key elements such that the total search cost is as small as possible.

(b) 0-1 Knapsack Problem - Given a set of 'n' items, Items[1..n], each item is having weight & profit. Given a knapsack with maximum weight capacity W. The aim is to fill the knapsack with the selected items for the maximum capacity such that the total profit should be maximum.

Algorithm(s):

(a) Optimal Binary Search Tree

Algorithm OptimalBST(Keys[1..n], P[1..n], Q[0..n], n)

Input: Keys[1..n] – 'n' numbers of integer key elements
P[1..n] – Probability of Successful Searches
Q[0..n] – Probability of Unsuccessful Searches
n – Number of Key Elements

Output: C[0..n, 0..n] – Cost Matrix
R[0..n, 0..n] – Root Matrix

Let C[0..n, 0..n] be an array – Cost Matrix
Let W[0..n, 0..n] be an array – Weight Matrix
Let R[0..n, 0..n] be an array – Root Matrix

```
For Len ← 1 to n+1 do
    For i ← 0 to (n+1)-Len do
        j ← i + Len – 1
        If i=j then
            W[i, j] ← Q[i]
        Else
            W[i, j] ← W[i, j-1] + P[j] + Q[j]
        End If
    End For
End For
```

```
For Len ← 1 to n+1 do
    For i ← 0 to (n+1)-Len do
        j ← i + Len – 1
        If i=j then
            C[i, j] ← R[i, j] ← 0
        Else
            Min ← ∞
            MinK ← -1
            For k ← i+1 to j do
                Sum ← C[i, k-1] + C[k, j] + W[i, j]
                If Sum<Min then
                    Min ← Sum
                    MinK ← k
                End If
            End For
            C[i, j] ← Min
            R[i, j] ← MinK
        End If
    End For
End For

Return C & R

End OptimalBST
```

(b) 0/1 Knapsack Problem

```
Algorithm ZeroOneKnapsackDP(W[1..n], P[1..n], C, n)
    Input:      W[1..n] – List of Weights for 'n' Items
                P[1..n] – Profit for 'n' Items
                C – Capacity of Knapsack
                n – Number of Items
    Output:     M[0..n, 0..C] – Maximum Profit Matrix

    Let M[0..n, 0..C] be an array – Maximum Profit Matrix

    For j←0 to C do
        M[0, j] ← 0
    End For

    For i←0 to n do
```

```
        M[i, 0] ← 0
    End For

    For i←1 to n do
        For j←1 to W[i]-1 do
            M[i, j] ← M[i-1, j]
        End For

        For j←W[i] to C do
            If M[i-1, j]>(P[i]+ M[i-1, j-W[i]]) then
                M[i, j] ← M[i-1, j]
            Else
                M[i, j] ← P[i] + M[i-1, j-W[i]]
            End If
        End For
    End For

    Return M

End ZeroOneKnapsackDP
```

Results & Discussion:

Sample Input & Output

(a) Optimal Binary Search Tree

Input:

n = 4	0	1	2	3	4
Keys[1..4]		10	20	30	40
P[1..4]		3	3	1	1
Q[0..4]	2	3	1	1	1

Output:

```
Matrix - w
        2       8       12      14      16
        3       7       9       11
        1       3       5
        1       3
        1
```

```
Matrix - c
        0       8       19      25      32
        0       7       12      19
        0       3       8
        0       3
        0
```

```
Matrix - r
        0       1       1       2       2
        0       2       2       2
        0       3       3
        0       4
        0

Optimal Cost: 2
```

(b) 0/1 Knapsack Problem

Input:

No. of Items: n = 4				
Item	**1**	**2**	**3**	**4**
Weight[1..4]	3	4	5	6
Profit[1..4]	2	3	4	1
Bag Capacity: W = 8				

Output:

```
Profit Matrix - m
        0       0       0       0       0       0       0       0       0
        0       0       0       2       2       2       2       2       2
        0       0       0       2       3       3       3       5       5
        0       0       0       2       3       4       4       5       6
        0       0       0       2       3       4       4       5       6
Maximum Profit: 6
```

Experiment Number: 04

Applications of Branch & Bound

Travelling Salesman Problem & 0-1 Knapsack Problem

Aim:

(a) Given a set of cities and distance between each pair of cities, the problem is to find the shortest possible trip to visit every city exactly once and returns to the starting city. The cities are mapped with the vertices and the distance between the pairs are mapped with the edge cost of a graph. The aim is to design and demonstrate an algorithm for solving the TSP by using least cost branch and bound strategy.

(b) 0-1 Knapsack Problem - Given a set of 'n' items, Items[1..n], each item is having weight & profit. Given a knapsack with maximum weight capacity W. The aim is to fill the knapsack with the selected items for the maximum capacity such that the total profit should be maximum

Algorithm(s):

(a) Travelling Salesman Problem

Algorithm TSP(Cost[1..n][1..n], n, Start)

Input: n – Number of Vertices
Start – Starting Vertex
Cost[1..n][1..n] - 'n x n' Cost matrix.

Output: Shortest Tour and the Cost

Let S[1..m] be an array of Nodes.
Each node has the properties: Vertex, Visited[1..n], nVisitied, Path[0..n], CostMatrix[1..n][1..n], Cost and Alive.

Cnt 🡐 0

Cnt 🡐 Cnt + 1
S[Cnt].Alive 🡐 True
S[Cnt].Cost 🡐 ∞
For j←1 to n ***do***
    S[Cnt].Visisted 🡐 False
End For
S[Cnt].nVisited 🡐 0
S[Cnt].Vertex 🡐 Start
S[Cnt].Path[S[Cnt].nVisited] 🡐 Start
S[Cnt].nVisited 🡐 S[Cnt].nVisited + 1
S[Cnt].CostMatrix 🡐 Cost

```
    S[Cnt].Cost ← ReduceMatrix(S[Cnt].CostMatrix, n)

    While (i ← GetLeastCost(S,Cnt)) ≠ -1 do
        S[i].Alive ← False
        nUnVisited ← 0
        For j ← 1 to n do
            If S[i].Visited[j] ≠ True then
                nUnVisited ← nUnVisited + 1
                u ← S[i].Vertex
                v ← j
                Cnt ← Cnt + 1
                S[Cnt] ← S[i]
                S[Cnt].Alive ← True
                S[Cnt].Vertex ← v
                S[Cnt].Path[S[Cnt].nVisited] ← v
                S[Cnt].nVisited ← S[Cnt].nVisited + 1
                S[Cnt].Visited[v] ← True;
                For k ← 1 to n do
                    S[Cnt].CostMatrix[u][k] ← ∞
                    S[Cnt].CostMatrix[k][v] ← ∞
                End For
                S[Cnt].CostMatrix[v][Start] ← ∞
                costReduced ← ReduceMatrix(S[Cnt].CostMatrix, n)
                S[Cnt].Cost ← S[i].Cost+costReduced+S[i].CostMatrix[u][v];
            End If
        End For

        If nUnVisited = 0 then
            // Killing Nodes
            For k ← 1 to Cnt do
                If S[k].Alive and S[k].Cost> S[i].Cost then
                    S[k].Alive ← False
                End If
            End For
            S[i].Path[S[i].nVisited] ← Start
            S[i].nVisited ← S[i].nVisited + 1
            sNode ← i
        End If
    End While
    Return S[sNode]
End TSP
```

```
Algorithm ReduceMatrix(Cost[1..n][1..n], n)
    Input:      n – Number of Vertices
                Cost[1..n][1..n] - ‘n x n’ Cost matrix.
    Output:     Reduced Matrix - Cost[1..n][1..n] and
                The reduced cost
    TotalCost ← 0

    //Row wise reduction
    For i ← 1 to n do
        Min ← Cost[i][1]
        For j ← 2 to n do
            If Cost[i][j] < Min then
                Min ← Cost[i][j]
            End If
        End For
        For j ← 1 to n do
            If Min ≠ ∞ and Cost[i][j] ≠ ∞ then
                Cost[i][j] ← Cost[i][j] - Min
            End If
        End For
        If Min ≠ ∞ then
            TotalCost ← TotalCost + Min
        End If
    End For

    //Column wise reduction
    For j ← 1 to n do
        Min ← Cost[1][j]
        For i ← 2 to n do
            If Cost[i][j] < Min then
                Min ← Cost[i][j]
            End If
        End For
        For i ← 1 to n do
            If Min ≠ ∞ and Cost[i][j] ≠ ∞ then
                Cost[i][j] ← Cost[i][j] - Min
            End If
        End For
        If Min ≠ ∞ then
            TotalCost ← TotalCost + Min
        End If
    End For
```

Return TotalCost
End ReduceMatrix

Algorithm GetLeastCost(Node S[1..m], m)

Input: S[1..m] – List of 'm State Space Tree Nodes
Output: Index of a Node with Least Cost

```
    LeastCost ← ∞
    LeastCostIndex ← -1
    For i ← 1 to m do
        If S[i].Alive and S[i].Cost<LeastCost then
            LeastCost ← S[i].Cost
            LeastCostIndex ← i
        End If
    End For
    Return LeastCostIndex
End GetLeastCost
```

(b) 0-1 Knapsack Problem

Algorithm:

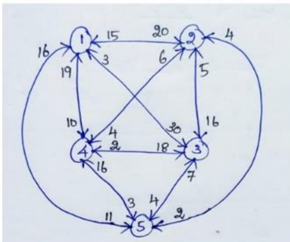
1. Sort all items in decreasing order of ratio of value per unit weight so that an upper bound can be computed using Greedy Approach.
2. Initialize maximum profit, maxProfit = 0, create an empty queue, Q, and create a dummy node of decision tree and enqueue it to Q. Profit and weight of dummy node are 0.
3. Do following while Q is not empty.
    a. Extract an item from Q. Let the extracted item be u.
    b. Compute profit of next level node. If the profit is more than maxProfit, then update maxProfit.
    c. Compute bound of next level node. If bound is more than maxProfit, then add next level node to Q.
    d. Consider the case when next level node is not considered as part of solution and add a node to queue with level as next, but weight and profit without considering next level nodes.

Results & Discussion:

Sample Input & Output

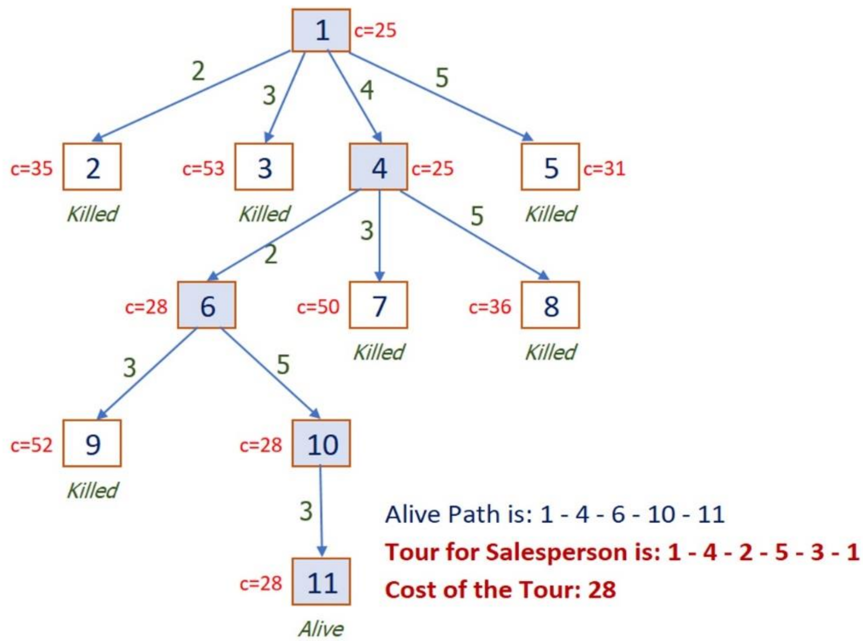
(a) Travelling Sales Person Problem

Input Graph



$$\begin{array}{c|c} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{pmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{pmatrix} \end{array}$$

Result




```
Travelling Salesperson Problem:

Number of Vertices: 5 [1..5]
Starting Vertex: 1

Tour Cost: 28
Tour:    1  4  2  5  3  1
```

```
Travelling Salesperson Problem:

Number of Vertices: 5 [1..5]
Starting Vertex: 3

Tour Cost: 28
Tour:    3  1  4  2  5  3
```

(b) 0-1 Knapsack Problem

Input:

No. of Items: n = 4				
Item	**1**	**2**	**3**	**4**
Weight[1..4]	3	4	5	6
Profit[1..4]	2	3	4	1
Bag Capacity: W = 8				

Output:

Maximum Profit: 6

Experiment Number: 05

Applications of Backtracking

n Queen Problem & Sum of Subset Problem

Aim:

(a) n-Queen Problem - The n Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. The chess queens can attack in any direction as horizontal, vertical, horizontal and diagonal way. Given an input 'n' (number of queens), aim is to find all the possible solutions to place 'n' queens properly by using backtracking approach.

(b) Sum of Subset Problem - Given a set of positive integers, and a value sum, determine that the sum of the subset of a given set is equal to the given sum.

Algorithm(s):

(a) n Queen Problem

Algorithm nQueen(n, Solutions[0..m-1][0..n-1], m)

Input: n – Number of Queens

Output: m – Number of Solutions

Solutions[0..m-1][0..n-1] – Solutions matrix – Each row represents a solution.

True or False – Can be solved or Not

Let Board[0..n-1][0..n-1] be a 'n x n' Boolean matrix (values: 0 or 1)– Represents a chess board.

```
For i←0 to n-1 do
    For j←0 to n-1 do
        Board[i][j] ← 0
    End For
End For

//Try placing a queen from 0th Row
Row ← 0
Return PalaceQueen(Board, n, Row, Solutions, m)
End nQueen
```

Algorithm PlaceQueen(Board[0..n-1][0..n-1], n, Row, Solutions[0..m-1][0..n-1], m)

Input: n – Number of Queens

Board[0..n-1][0..n-1] - 'n x n' Boolean matrix (values: 0 or 1)– Represents a chess board.

Row – Placing a queen at this row.

<u>Output:</u> m – Number of Solutions

Solutions[0..m-1][0..n-1] – Solutions matrix – Each row represents a solution.

True or False – Can be solved or Not

```
//Base Case
If Row = n then
    //Solution Found. Copy it into Solutions[] array
    K ← 0
    For i←0 to n-1 do
        For j←0 to n-1 do
            If Board[i][j]=1 then
                Solutions[m][k] ← j+1
                k ← k + 1
            End If
        End For
    End For
End If
Res ← False
For Col←0 to n-1 do
    If IsSafe(Board, n, Row, Col) then
        // Place this queen in board[r][c]
        Board[Row][Col] ← 1;

        If PlaceQueen(board, n, r+1, Solutions, m) = True then
            Res ← True
        End If

        // If placing queen in board[r][c] doesn't lead to a solution,
        // then remove queen from board[r][c]
        Board[Row][Col] = 0; // BACKTRACK
    End If
End For
Return Res
End PlaceQueen
```

Algorithm IsSafe(Board[0..n-1][0..n-1], n, Row, Col)

<u>Input:</u> n – Number of Queens

Board[0..n-1][0..n-1] - 'n x n' Boolean matrix (values: 0 or 1)– Represents a chess board.

Row & Col – Row and Column to check placement chance.

Output: True or False – Can be placed at Board[Row][Col] or Not

```
//Check this row on left side
    For i←0 to row-1 do
    If board[i][Col] = True then
        Return False;
    End If
End For

//Check upper diagonal on left side
i ← Row
j ← Col
While i ≥ 0 and j ≥ 0 do
    If Board[i][j] = True then
        Return False
    End If
    i ← i – 1
    j ← j – 1
End While

//Check upper diagonal on right side
i ← Row
j ← Col
While i ≥ 0 and j < n do
    If Board[i][j] = True then
        Return False
    End If
    i ← i – 1
    j ← j + 1
End While
Return True
End IsSafe
```

(b) Sum of Subset Problem

```
Algorithm SumOfSub(s, k, r)
    x[k] ← 1
    If s+w[k] = m  Then
        Print x[1:k]
    Else If s+w[k]+w[k+ 1] ≤ m Then
        SumOfSub(s+w[k],k+1,r-w[k]);
    End If
```

If ((s+r- w[k] ≥ m) ***and*** (s+w[k+1] ≤ m)) ***Then***

x[k] 🡐 0;

SumOfSub(s,k + l, r-w[k])

End If

End SumOfSub

Results & Discussion:

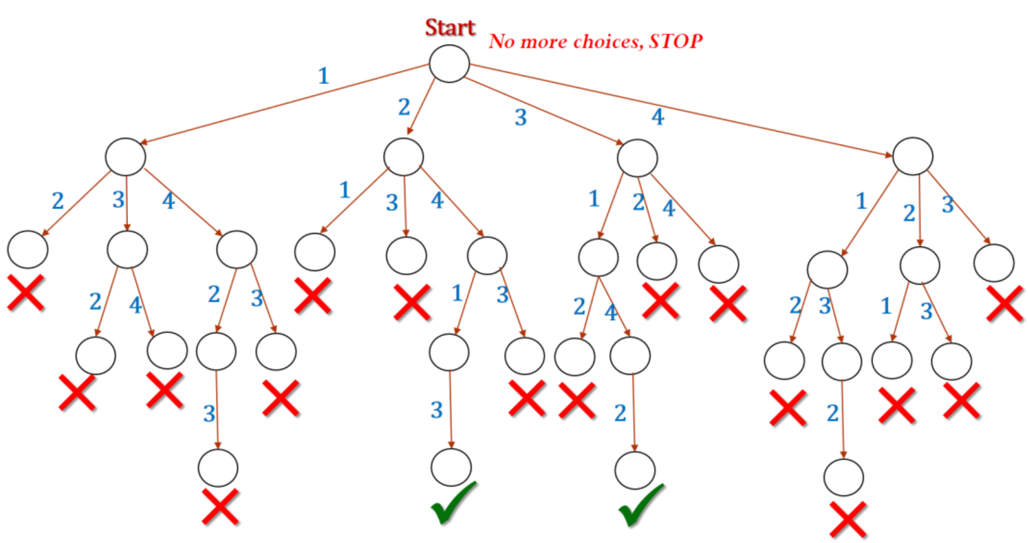
Sample Input & Output:

(a) n-Queen Problem

```
Number of Queens: 8
Number of Solutions Found: 92
1. [1 5 8 6 3 7 2 4 ]
2. [1 6 8 3 7 4 2 5 ]
3. [1 7 4 6 8 2 5 3 ]
4. [1 7 5 8 2 4 6 3 ]
5. [2 4 6 8 3 1 7 5 ]
6. [2 5 7 1 3 8 6 4 ]
7. [2 5 7 4 1 8 6 3 ]
8. [2 6 1 7 4 8 3 5 ]
9. [2 6 8 3 1 4 7 5 ]
10. [2 7 3 6 8 5 1 4 ]
11. [2 7 5 8 1 4 6 3 ]
12. [2 8 6 1 3 5 7 4 ]
13. [3 1 7 5 8 2 4 6 ]
```

```
Number of Queens: 5
Number of Solutions Found: 10
1. [1 3 5 2 4 ]
2. [1 4 2 5 3 ]
3. [2 4 1 3 5 ]
4. [2 5 3 1 4 ]
5. [3 1 4 2 5 ]
6. [3 5 2 4 1 ]
7. [4 1 3 5 2 ]
8. [4 2 5 3 1 ]
9. [5 2 4 1 3 ]
10. [5 3 1 4 2 ]
```

```
Number of Queens: 4
Number of Solutions Found: 2
1. [2 4 1 3 ]
2. [3 1 4 2 ]
```



Solutions
[2, 4, 1, 3]
[3, 1, 4, 2]

	1	2	3	4
1		Q		
2				Q
3	Q			
4			Q	

	1	2	3	4
1			Q	
2	Q			
3				Q
4		Q		

(b) Sum of Subset Problem Problem

Input: *set[] = {1,2,1}, sum = 3*
Output: *[1,2],[2,1]*
Explanation: *There are subsets [1,2],[2,1] with sum 3.*

Input: *set[] = {3, 34, 4, 12, 5, 2}, sum = 30*
Output: *[]*
Explanation: *There is no subset that add up to 30.*

Experiment Number: 06

Programs on Graphs – Application of DFS

Topological Sort of Directed Acyclic Graph

Aim:

Given a directed acyclic graph G, aim is to find the topological ordering of vertices by applying DFS traversal on graph.

Algorithm(s):

(a) Topological Sort of Directed Acyclic Graph

Algorithm TopologicalOrder(G)

Input: G[0..n-1] – A Graph with list of Vertices & Edges.

Output: Sequence of Vertices - Sorted in topological order.

1. Call "DFS(G)" to compute the finishing time for every vertex v ϵ G.V
2. As each vertex is finished, Insert it onto the front of the Linked List.
3. Return the Linked List.

End TopologicalOrder

Algorithm DFS(G)

Input: G[0..n-1] – A Graph with list of Vertices & Edges. Each vertex has the following attributes: Value, Parent, Color, Starting Time and Finishing Time

Output: Sequence of Vertices - Sorted in topological order.

```
For each vertex v ϵ G.V do
        v.Parent ← NIL
        v.Color ← WHITE
End For
Time ← 0
For each vertex u ϵ G.V do
        If u.Color = WHITE then
                DFS_VISIT(G, u, Time)
        End If
End For
End DFS
```

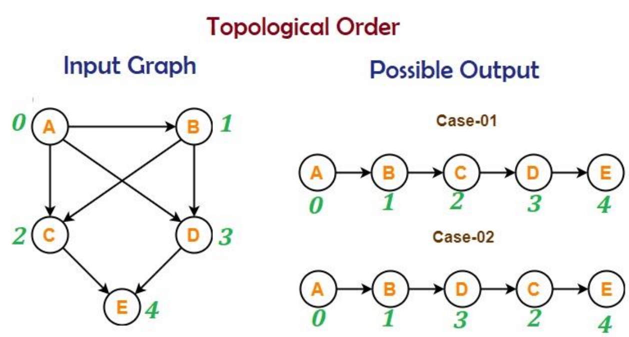
Algorithm DFS_VISIT(G, u, Time)

Input: G[0..n-1] – A Graph with list of Vertices & Edges. Each vertex has the following attributes: Value, Parent, Color, Starting Time and Finishing Time.
u – Starting Vertex
Time – Step Number – Order of visiting vertices.

Output: Display the vertices in DFS order and compute starting time and finishing time.

u.Color 🡠 GRAY
Time 🡠 Time + 1
u.Start 🡠 Time
For each vertex v ϵ G.V ***do***
    If v.Color = WHITE ***then***
        v.Parent 🡠 u
        DFS_VISIT(G, v, Time)
    End If
End For
u.Color 🡠 BLACK
Time 🡠 Time + 1
u.Finish 🡠 Time
End DFS_VISIT

Sample Input & Output:




```
Graph After DFS:
****************************************************
Parent | Vertex Value | Color | Start | Finish
****************************************************
  NULL |            0 |     2 |     1 |     10
     0 |            1 |     2 |     2 |      9
     1 |            2 |     2 |     3 |      6
     1 |            3 |     2 |     7 |      8
     2 |            4 |     2 |     4 |      5
****************************************************

Topological Order of Vertices:    0    1    3    2    4
```

Experiment Number: 07

Programs on Graphs – Minimum Spanning Tree

Prim's and Kruskal's Algorithms

Aim:

Given a weighted undirected graph G, aim is to find the Minimum Spanning Tree by applying the Prim's algorithm and Kruskal's algorithm.

Algorithm(s):

(a)Prim's Algorithm for MST

Algorithm Prims(G, w, u)

Input: G[1..n] – A Graph with list of Vertices. Each vertex has properties: Value, Parent, Color (W or B) and Cost.
w – A 'n x n' matrix – Contains edge cost between the edges. (n is the number of vertices)
u – Starting Vertex

Output: Minimum Spanning Tree - Updated vertices – Selected Edges for spanning tree.

```
For each vertex v ∈ G.V do
    v.Parent ← NIL
    v.Color ← WHITE
    v.Cost ← ∞
End For
u.Cost ← 0
Let Q be a Min-Priority Queue
Q ← G.V
While Q ≠ Φ do
    s ← ExtractMin(Q)
    For each v ∈ G.Adj[s] do
        If v.Color = WHITE then
            If w(s,v) < v.Cost then
                v.Cost ← w(s,v)
                v.Parent ← s
            End If
        End If
    End For
    s.Color ← BLACK
```

End While
End Prims

(b) Kruskal's Algorithm for MST

Algorithm Kruskals(G, w)

Input: G[1..n] – A Graph with list of Vertices & Edges.
w – A 'n x n' matrix – Contains edge cost between the edges. (n is the number of vertices)
u – Starting Vertex

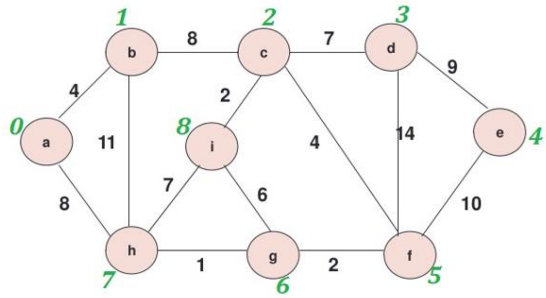
Output: A set of selected Edges.

```
A ← Φ
For each vertex v ϵ G.V do
        MAKE_SET(v)
End For

//Sort the edges in ascending order based on the cost
SortAscendingCost(G.E)

For each edge (u,v) ϵ G.E do
        If FIND_SET(u) ≠ FIND_SET(v) then
                A ← A U { (u,v) }
                UNION(u,v)
        End If
End For
Return A
End Kruskals
```

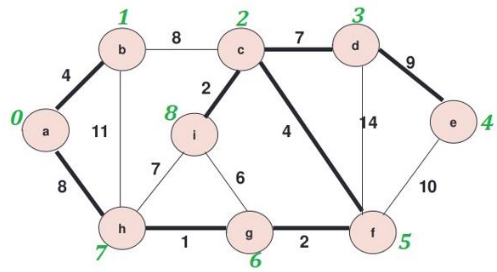
Sample Input & Output:

Prim's & Kruskal's Algorithms

Input Graph



Output - Minimum Spanning Tree Edges



Minimum Cost: *37*

```
Prim's Result:

Minimum Spanning Tree Edges:
0--1 : Cost-4
1--2 : Cost-8
2--3 : Cost-7
3--4 : Cost-9
2--5 : Cost-4
5--6 : Cost-2
6--7 : Cost-1
2--8 : Cost-2

Minimum Cost: 37
```

```
Kruskal's Result:

Minimum Spanning Tree Edges:
7--6 : Cost-1
6--5 : Cost-2
8--2 : Cost-2
5--2 : Cost-4
1--0 : Cost-4
3--2 : Cost-7
2--1 : Cost-8
4--3 : Cost-9

Minimum Cost: 37
```

Experiment Number: 08

Programs on Graphs – Shortest Path Algorithms

Single Source Shortest Paths using Bellman-Ford Algorithm

Aim:

Given a weighed graph G, aim is to find the shortest path from a vertex to all other vertices by applying Bellman-Ford algorithm.

Algorithm(s):

(a) Bellman-Ford Algorithm

Algorithm BellmanFord(G, w, s)

Input: G[0..n-1] – A Graph with list of Vertices. Each vertex has properties: Value, Parent, and Distance.
w – A 'n x n' matrix – Contains edge cost between the edges. (n is the number of vertices)
s – Starting Vertex

Output: Shortest path from the vertex 's' to all other vertices.

```
For each vertex v ϵ G.V do
    v.Parent ← NIL
    v.Distance ← ∞
End For

s.Distance ← 0

For i ← 1 to |G.V|-1 do
    For each u ϵ G.V do
        For each v ϵ G.Adj[u] do
            Relax(u, v, w)
        End For
    End For
End For

For each u ϵ G.V do
    For each v ϵ G.Adj[u] do
        If v.Distance > u.Distance + w(u,v) then
            Return False
```

End For

End For

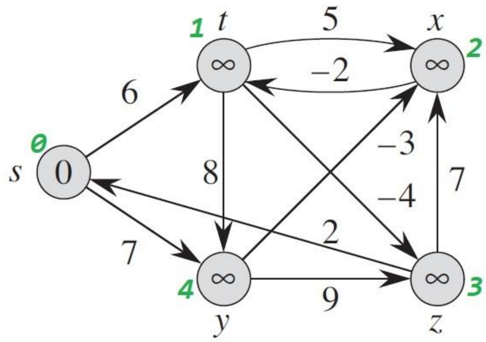
Return True

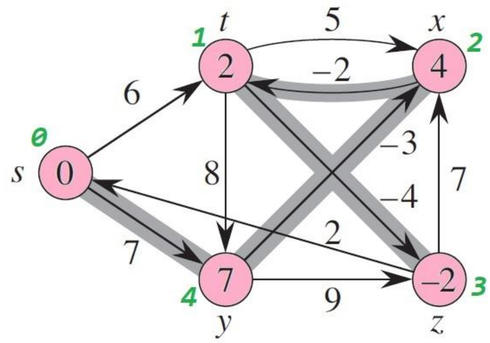
End BellmanFord

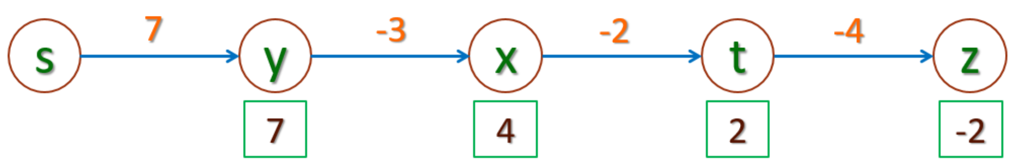
Sample Input & Output:

Bellman-Ford Single Source Shortest Path Algorithm

Input Graph **Output Graph**








```
Single Source Shortest Path (Bellman-Ford Algorithm) Result:

Vertex | Parent | Distance |
*******************************************
     0 | Source |        0 |
     1 |      2 |        2 |
     2 |      4 |        4 |
     3 |      1 |       -2 |
     4 |      0 |        7 |
*******************************************
```

Experiment Number: 09

Programs on Graphs – Shortest Path Algorithms

All-Pairs Shortest Paths using Floyd-Warshall algorithm

Aim:

Given a weighed graph G, aim is to find the shortest path from a vertex to all other vertices by applying Bellman-Ford algorithm.

Algorithm(s):

(a) Floyd-Warshall Algorithm

Algorithm FloydWarshall(Weight[1..n][1..n], Distance[1..n][1..n], Path[1..n][1..n])

Input: Weight[1..n][1..n] – A 'n x n' weight matrix – Contains edge cost between the vertices. (n is the number of vertices)

Output: Distance[1..n][1..n] – A 'n x n' distance matrix – Contains shortest distance between each and every pairs of vertices.
Parent[1..n][1..n] – Path matrix – Contains list of parents in the path for each every pair of vertices.

Let $D^{(0)}$ be a 'n x n' matrix
$D^{(0)}$ 🡐 Weight

For i ← 1 to n ***do***
    For j ← 1 to n ***do***
        If D_{ij}^{0} = 0 or D_{ij}^{0} = ∞ ***then***
            P_{ij}^{0} 🡐 NIL
        Else
            P_{ij}^{0} 🡐 i
        End If
    End For
End For

For k ← 1 to n ***do***
    Let $D^{(k)}$ and $P^{(k)}$ - 'n x n' matrices
    For i ← 1 to n ***do***
        For j ← 1 to n ***do***
            If $D_{ik}^{(k-1)} + D_{kj}^{(k-1)} < D_{ij}^{(k-1)}$ ***then***
                $D_{ij}^{(k)}$ 🡐 $D_{ik}^{(k-1)} + D_{kj}^{(k-1)}$

$P_{ij}^{(k)}$ 🡐 $P_{kj}^{(k-1)}$

Else

$D_{ij}^{(k)}$ 🡐 $D_{ij}^{(k-1)}$

$P_{ij}^{(k)}$ 🡐 $P_{ij}^{(k-1)}$

End If

End For

End For

End For

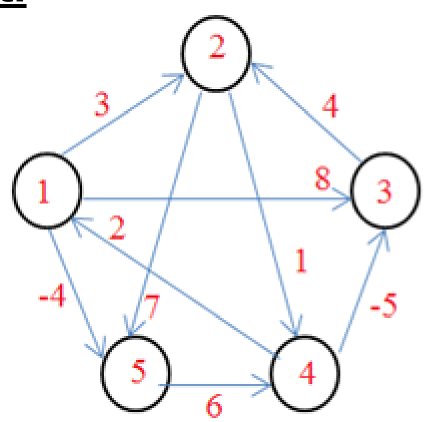
Distance 🡐 $D^{(n)}$

Parent 🡐 $P^{(n)}$

End FloydWarshall

Sample Input & Output:

Input:



Weight Matrix					
	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	3	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0

Output:

```
Distance Matrix:
0       1       -3      2       -4
3       0       -4      1       -1
7       4       0       5       3
2       -1      -5      0       -2
8       5       1       6       0

Parent Matrix:
NIL     3       4       5       1
4       NIL     4       2       1
4       3       NIL     2       1
4       3       4       NIL     1
4       3       4       5       NIL
```

Experiment Number: 10

Programs on Graphs – Network Flow Problem
Maximum Flow using Ford Fulkerson Algorithm

Aim:

Given a weighed graph G, which represents a water flow network. The vertices represent the tanks and the edges represent the pipes connected between the tanks. Edge cost maps to the capacity of the pipe. The aim is to find how much stuff can be pushed from the source to the sink by applying Ford Fulkerson algorithm.

Algorithm(s):

(a) Ford Fulkerson Algorithm

Algorithm MaxFlowFordFulkerson(Cost[0..n-1][0..n-1], Source, Sink)

Input: Cost[0..n-1][0..n-1] – A 'n x n' Cost matrix – Contains edge cost between the vertices. (n is the number of vertices)
Source – Source Vertex
Sink – Sink Vertex

Output: Possible Maximum Flow Capacity From Source to Sink.

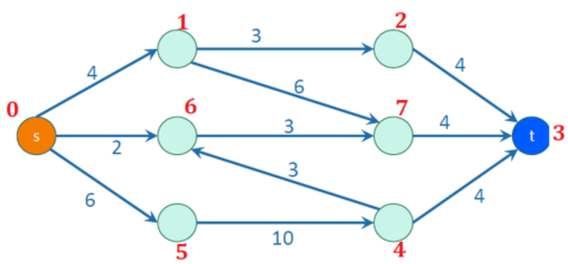
```
Let Path[1..n] be an array represents a path in flow network.

maxFlow ← 0

//Find the best path from source to sink by using BFS
Cnt = PathBFS(Cost, Source, Sink, Path)

//Check for a path is exist between Source & Sink
While Cnt>1 do
        minCapacity ← ∞
        For i ← 0 to Cnt-1 do
                u ← Path[i]
                v ← Path[i+1]
                If Cost[u][v]<minCapacity then
                        minCapacity ← Cost[u][v]
                End If
        End For
        maxFlow ← maxFlow + minCapacity
```

```
        For i ← 0 to Cnt-1 do
            u ← Path[i]
            v ← Path[i+1]
            b[u][v] = b[u][v] - minCapacity
            b[v][u] = b[v][u] + minCapacity
        End For
        Cnt ← PathBFS(Cost, Source, Sink, Path)
    End While
    Return MaxFlow
End MaxFlowFordFulkerson

Algorithm PathBFS(Cost[0..n-1][0..n-1], Source, Sink, Path[1..n])
    Input:      Cost[0..n-1][0..n-1] – A 'n x n' Cost matrix – Contains edge cost
                between the vertices. (n is the number of vertices)
                Source – Source Vertex
                Sink – Sink Vertex
    Output:     Path[] – Sequence of Vertices – A best path between Source to Sink.
                Cnt – Number of Edges in the path

    Let V[1..n] be an array of vertices.

    For each vertex v ϵ V do
        v.Parent ← NIL
        v.Color ← WHITE
        v.Distance ← ∞
    End For

    For k ← 1 to n do
        For i ← 1 to n do
            V[i].AdjList.Add(j)
        End For
    End For

    V[Source].Color ← GRAY
    V[Source].Distance ← 0
    V[Source].Parent ← NIL

    Let Q be an Empty Queue

    Q.EnQ(Source)

    While Q ≠ Φ do
```

```
            u ← Q.DeQ()
            For each v ϵ G.Adj[u] do
                If v.Color = WHITE then
                    v.Color ← GRAY
                    v.Distance ← u.Distance + 1
                    v.Parent ← u
                    Q.EnQ(v)
                End If
            End For
            u.Color ← BLACK
        End While

        // Optaining Path (in reverse: Sink to Source)
        Cnt ← 0
        Path[Cnt] ← Sink
        Cnt ← Cnt + 1

        t ← Sink
        While V[t].Parent <> NIL do
            Path[Cnt] ← V[t].Parent
            Cnt ← Cnt + 1
            t ← V[t].Parent
        End While

        //Reversing path to obtain original path: Source to Sink
        i ← 0
        j ← Cnt-1
        While i<j do
            Swap(Path[i],Path[j])
            i ← i + 1
            j ← j – 1
        End For

        Return Cnt
    End PathBFS
```

Sample Input & Output:
Input:

Input Graph



Weight Matrix								
	0	1	2	3	4	5	6	7
0	0	4	0	0	0	6	2	0
1	0	0	3	0	0	0	0	6
2	0	0	0	4	0	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	4	0	0	3	0
5	0	0	0	0	10	0	0	0
6	0	0	0	0	0	0	0	3
7	0	0	0	4	0	0	0	0

Output:

```
Best Path by BFS: 0 --> 1 --> 2 --> 3
MinCap: 3

Best Path by BFS: 0 --> 1 --> 7 --> 3
MinCap: 1

Best Path by BFS: 0 --> 5 --> 4 --> 3
MinCap: 4

Best Path by BFS: 0 --> 6 --> 7 --> 3
MinCap: 2

Best Path by BFS: 0 --> 5 --> 4 --> 6 --> 7 --> 3
MinCap: 1

Maximum Flow From 0 to 3: 11
```

Additional Experiment:

Approximation Algorithm
Bin-Packing Problem

Aim:

Given 'n' numbers of items with weight and some numbers of bins with fixed capacity. The aim is to assign each item to a bin such that number of total used bins is minimized. Assumption is "All items have weights smaller than bin capacity". Implement all the four approximation algorithms for solving bin-packing problem.

Algorithm(s):

(a) First Fit Algorithm

Algorithm BinPackingFirstFit(Objects[1..n], n, Bins[])

Input: n – Number of Objects
Objects[1..n] – 'n' objects each with object number and weight.
Bins[] – List of empty bins. Each bin has properties: Objects[1..n], nObjects, C (Capacity of bin), uc (Used Capacity), bc (Balance Capacity)

Output: Bins[1..m] – 'm' number of used bins.
m – Number of Used Bins.

```
m ← 0
For i ← 1 to n do
    j ← ChooseFirstFitBin(B, m,  Objects[i].weight)
    B[j].Objects[B[j].nObjects] ← Objects[i].weight
    B[j].nObjects ← B[j].nObjects + 1
    B[j].uc ← B[j].uc + Objects[i].weight
    B[j].bc ← B[j].bc - Objects[i].weight;
End For
Return m
End BinPackingFirstFit
```

```
Algorithm ChooseFirstFitBin(Bins[1..m], m, weight)
    For i ← 1 to m do
        If Bins[j].bc >= weight then
            Return j
        End If
    End For
    m ← m + 1
```

```
        Return m
End ChooseFirstFitBin
```

(b) Best Fit Algorithm

```
Algorithm BinPackingBestFit(Objects[1..n], n, Bins[])
     Input:   n – Number of Objects
              Objects[1..n] – 'n' objects each with object number and weight.
              Bins[] – List of empty bins. Each bin has properties: Objects[1..n],
              nObjects, C (Capacity of bin), uc (Used Capacity), bc (Balance Capacity)
     Output:  Bins[1..m] – 'm' number of used bins.
              m – Number of Used Bins.

     m ← 0
     For i ← 1 to n do
          j ← ChooseBestFitBin(B, m,  Objects[i].weight)
          B[j].Objects[B[j].nObjects] ← Objects[i].weight
          B[j].nObjects ← B[j].nObjects + 1
          B[j].uc ← B[j].uc + Objects[i].weight
          B[j].bc ← B[j].bc - Objects[i].weight;
     End For

     Return m
End BinPackingFirstFit

Algorithm ChooseBestFitBin(Bins[1..m], m, weight)
     s ← 0
     minbc ← 99999;
     For i ← 1 to m do
          If Bins[j].bc >= weight then
               If Bins[j].bc < minbc then
                    s  ← j
                    minbc ← B[j].bc
               End If
          End If
     End For
     If s > 0 then
          Return s;
     End If
     m ← m + 1
     Return m
End ChooseBestFitBin
```

(c) First Fit Decreasing Algorithm

Algorithm BinPackingFirstFitDecreasing(Objects[1..n], n, Bins[])

Input: n – Number of Objects
Objects[1..n] – 'n' objects each with object number and weight.
Bins[] – List of empty bins. Each bin has properties: Objects[1..n], nObjects, C (Capacity of bin), uc (Used Capacity), bc (Balance Capacity)

Output: Bins[1..m] – 'm' number of used bins.
m – Number of Used Bins.

```
Let DecreasingObjects[1..n] be an array of objects.

Copy Objects[1..n] to DecreasingObjects[1..n]

//Sorting DecreasingObjects[1..n] in descending order of its weight
For i ← 1 to n do
    For j ← i+1 to n do
        If DecreasingObjects[j].weight > DecreasingObjects[i].weight then
            Swap(DecreasingObjects[j], DecreasingObjects[i])
        End If
    End For
End For
Return BinPackingFirstFit(DecreasingObjects, n, B)
End BinPackingFirstFitDecreasing
```

(d) Best Fit Decreasing Algorithm

Algorithm BinPackingBestFitDecreasing(Objects[1..n], n, Bins[])

Input: n – Number of Objects
Objects[1..n] – 'n' objects each with object number and weight.
Bins[] – List of empty bins. Each bin has properties: Objects[1..n], nObjects, C (Capacity of bin), uc (Used Capacity), bc (Balance Capacity)

Output: Bins[1..m] – 'm' number of used bins.
m – Number of Used Bins.

```
Let DecreasingObjects[1..n] be an array of objects.

Copy Objects[1..n] to DecreasingObjects[1..n]

//Sorting DecreasingObjects[1..n] in descending order of its weight
For i ← 1 to n do
```

```
            For j ← i+1 to n do
                If DecreasingObjects[j].weight > DecreasingObjects[i].weight then
                    Swap(DecreasingObjects[j], DecreasingObjects[i])
                End If
            End For
        End For
        Return BinPackingBestFit(DecreasingObjects, n, B)
End BinPackingBesttFitDecreasing
```

Sample Input & Output:

n = 6 *No. of Objects*

c = 10 *Bin Capacity*

w[1..6] = {5, 6, 3, 7, 5, 4 } *Weights of objects*

1	2	3	4	5	6	*Object Number*
5	6	3	7	5	4	*Object's Weight*

```
Optimum number of Bins Required: 3
```

```
Approximation - First Fit:
***************************
Number of Bins Required: 4
Objects Packed in different Bins:
        Bin-1: {1, 3}
        Bin-2: {2, 6}
        Bin-3: {4}
        Bin-4: {5}

Approximation - Best Fit:
***************************
Number of Bins Required: 4
Objects Packed in different Bins:
        Bin-1: {1, 5}
        Bin-2: {2, 3}
        Bin-3: {4}
        Bin-4: {6}
```

```
Approximation - First Fit Decreasing:
****************************************
Number of Bins Required: 3
Objects Packed in different Bins:
        Bin-1: {4, 3}
        Bin-2: {2, 6}
        Bin-3: {1, 5}

Approximation - Best Fit Decreasing:
****************************************
Number of Bins Required: 3
Objects Packed in different Bins:
        Bin-1: {4, 3}
        Bin-2: {2, 6}
        Bin-3: {1, 5}
```

Additional Experiments

1. Hamiltonian Cycles – Backtracking Strategy
2. Knapsack Problem – Backtracking Strategy
3. Matrix Chain Ordering – Dynamic Programming
4. String Editing Problem – Dynamic Programming
5. Travelling Salesman Problem – Dynamic Programming
6. Single Source Shortest Path – Dijkstra's Algorithm
7. Scheduling Independent Tasks – Approximation Algorithm
8. Interval Partitioning – Approximation Algorithm