Experiment No.4

Roll No.24141014

Title :-

Greedy method to solve problems of Optimal Merge Pattern

Aim :-

To implement the Optimal Merge Pattern using the greedy method and analyze its time and space complexity.

Objective :-

1. To understand the concept of the optimal merge pattern.
2. To apply the greedy method for solving the optimal merge pattern problem.
3. To analyze the efficiency of the greedy approach.
4. To study applications of the optimal merge pattern in real-world problems**.**

Theory :-

The **Optimal Merge Pattern Problem** is a classic example of the greedy method in which we need to determine the minimum total cost of merging a set of sorted files into one single sorted file.

Suppose there are *n* files with different lengths (number of records). When two files are merged, the cost of merging is equal to the sum of their sizes. After merging, the resulting file's size is also the sum of those two. The objective is to find the sequence of merges that leads to the **minimum total merging cost**.

The **greedy method** is used because, at each step, it makes the locally optimal choice — it merges the two smallest files first. This minimizes the intermediate cost at each step and ultimately results in the global optimum.

Steps involved:

1. List the sizes of all files to be merged.
2. Choose the two smallest files and merge them.

3. The cost of merging is the sum of their sizes.
4. Insert the new merged file back into the list.
5. Repeat the process until all files are merged into one.
6. The total cost is the sum of all intermediate merge costs.

APPLICATIONS:

1. Used in **external sorting** (e.g., multiway merge sort in file systems).
2. Useful in **data compression** (e.g., Huffman coding).
3. Used in **database management systems** where files are merged frequently.
4. Used in **information retrieval systems** to merge search results efficiently.
5. Used in **tape drives** and **merge scheduling** operations.

ALGORITHM:

Algorithm: Optimal Merge Pattern (Greedy Method)

Input: n files with sizes s[1], s[2], ..., s[n]

Steps:

1. Sort the file sizes in ascending order.
2. Initialize totalCost = 0.
3. While there is more than one file remaining:
    a. Select the two smallest files, say a and b.
    b. Merge them into a new file of size (a + b).
    c. Add the merge cost (a + b) to totalCost.
    d. Insert the new file size (a + b) back into the list.
    e. Sort the list again.
4. Repeat until only one file remains.
5. Display totalCost as the minimum number of comparisons (optimal merge cost).

Output: Minimum total cost of merging all files.

Program :-

```
#include <stdio.h>

void sort(int arr[], int n) {
    int i, j, temp;
    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

int optimalMerge(int files[], int n) {
    int totalCost = 0, cost, i;

    while (n > 1) {
        sort(files, n); // sort files by size
        cost = files[0] + files[1]; // merge smallest two files
        totalCost += cost;
        files[0] = cost; // replace first element with merged file
        for (i = 1; i < n - 1; i++) {
            files[i] = files[i + 1]; // shift elements left
        }
        n--; // reduce size of array
    }
```

```
    return totalCost;
}

int main() {
    int n, i, files[20], result;
    printf("Enter number of files: ");
    scanf("%d", &n);
    printf("Enter the sizes of the files:\n");
    for (i = 0; i < n; i++)
        scanf("%d", &files[i]);

    result = optimalMerge(files, n);
    printf("Minimum total cost of merging: %d\n", result);
    return 0;
}
```

Output

```
Enter number of files: 3
Enter the sizes of the files:
12
23
25
Minimum total cost of merging: 95


=== Code Execution Successful ===
```

TIME AND SPACE COMPLEXITY:

- Time Complexity: $O(n^2)$ using simple sorting (can be reduced to O(n log n) using a priority queue).

- Space Complexity: O(1) (in-place operations).

Conclusion :-

The Optimal Merge Pattern problem demonstrates how the greedy method can be applied to minimize total merge costs. By repeatedly merging the two smallest files, we ensure that each merge step contributes the least possible cost to the final total. The greedy approach gives an optimal solution and is widely used in file handling, sorting, and data compression applications.

Experiment No.5

Roll No.24141014

Title :-

Minimum Cost Spanning Tree of a given undirected graph using Prim’s algorithm and Kruskal's algorithm and compare

Aim :-

To implement Prim’s and Kruskal’s algorithms to find the Minimum Cost Spanning Tree (MST) of a given undirected weighted graph and compare their performance.

Objective :-

1. To understand the concept of a spanning tree and minimum cost spanning tree.
2. To implement Prim’s algorithm for MST construction.
3. To implement Kruskal’s algorithm for MST construction.
4. To compare the performance and efficiency of both algorithms.
5. To understand the applications of MST in real-world scenarios.

Theory :-

A spanning tree of a connected undirected graph is a subgraph that includes all the vertices with the minimum number of edges (n – 1) and no cycles.
A minimum cost spanning tree (MST) is a spanning tree in which the total weight (sum of edge costs) is minimum.

There can be several spanning trees for a graph, but the MST is the one with the lowest possible total edge weight.

-Prim’s Algorithm:
Prim’s algorithm is a greedy algorithm that grows the spanning tree starting from an initial vertex. It repeatedly adds the smallest edge that connects a vertex in the tree to a vertex outside the tree until all vertices are included.

Steps:

1. Select an arbitrary vertex as the starting point.
2. Initialize the MST with this vertex.
3. Find the edge with the smallest weight that connects the MST to a vertex not yet in the MST**.**
4. Add this edge and the vertex to the MST.
5. Repeat until all vertices are included.

-Kruskal's Algorithm:
Kruskal's algorithm is also a greedy algorithm but it builds the MST edge by edge. It starts with all vertices as separate trees and merges them using the smallest edge that connects two different trees, avoiding cycles.

Steps:

1. Sort all edges in non-decreasing order of their weights.
2. Select the smallest edge that does not form a cycle.
3. Add this edge to the MST.
4. Repeat until the MST contains (n – 1) edges.

-APPLICATIONS:

1. Designing communication and computer networks (telephone, electrical, internet).
2. Finding efficient road and rail connection routes.
3. Network routing and cluster analysis.
4. Image segmentation and circuit design.
5. Approximation algorithms for NP-hard problems like the Travelling Salesman Problem (TSP).

-ALGORITHMS:

Algorithm for Prim's:

Input: Weighted connected undirected graph with vertices V and edges E.
Output: Minimum cost spanning tree.

Steps:

1. Choose any vertex as the starting vertex.

2. Initialize visited[] array to mark included vertices.
3. Repeat (V - 1) times:
    a. Find the edge with minimum weight that connects a visited vertex and an unvisited vertex.
    b. Add this edge to the MST.
    c. Mark the new vertex as visited.
4. Print the edges and total cost of MST.

-Algorithm for Kruskal's:

Input: Weighted connected undirected graph with vertices V and edges E.
Output: Minimum cost spanning tree.

Steps:

1. Sort all edges in non-decreasing order of their weight.
2. Initialize each vertex as a separate set (use union-find structure).
3. For each edge (u, v):
    a. If including (u, v) does not form a cycle, add it to MST.
    b. Otherwise, discard it.
4. Continue until (V - 1) edges are added to MST.
5. Print the MST edges and total cost.

Program :-

```
//Prim's Algorithm

#include <stdio.h>

#include <limits.h>


#define MAX 20


int main() {

   int n, cost[MAX][MAX];

   int visited[MAX] = {0};
```

```
int i, j, ne = 1;
int min, a, b, u, v, total = 0;

printf("Enter number of vertices: ");
scanf("%d", &n);
printf("Enter the cost adjacency matrix (use 999 for no edge):\n");
for (i = 1; i <= n; i++)
   for (j = 1; j <= n; j++)
      scanf("%d", &cost[i][j]);

visited[1] = 1;

while (ne < n) {
   min = 999;
   for (i = 1; i <= n; i++)
      for (j = 1; j <= n; j++)
         if (cost[i][j] < min && visited[i] != 0 && visited[j] == 0) {
            min = cost[i][j];
            a = u = i;
            b = v = j;
         }

   if (visited[u] == 0 || visited[v] == 0) {
      printf("Edge %d: (%d -> %d) cost = %d\n", ne++, a, b, min);
      total += min;
      visited[b] = 1;
   }
   cost[a][b] = cost[b][a] = 999;
```

```
    }

    printf("Minimum cost = %d\n", total);
    return 0;
}
```

Output

```
Enter number of vertices: 4
Enter the cost adjacency matrix (use 999 for no edge):
23
14
54
45
67
999
78
66
54
34
54
67
54
69
98
87
Edge 1: (1 -> 2) cost = 14
Edge 2: (1 -> 4) cost = 45
Edge 3: (1 -> 3) cost = 54
Minimum cost = 113
```

//Kruskal's Algorithm

```
#include <stdio.h>
```

```
#define MAX 20

int find(int parent[], int i) {
   while (parent[i])
      i = parent[i];
   return i;
}

int uni(int parent[], int i, int j) {
   if (i != j) {
      parent[j] = i;
      return 1;
   }
   return 0;
}

int main() {
   int n, cost[MAX][MAX];
   int i, j, u, v, a, b, ne = 1, min, mincost = 0;
   int parent[MAX] = {0};

   printf("Enter number of vertices: ");
   scanf("%d", &n);
   printf("Enter the cost adjacency matrix (use 999 for no edge):\n");
   for (i = 1; i <= n; i++)
      for (j = 1; j <= n; j++)
         scanf("%d", &cost[i][j]);
```

```
    while (ne < n) {
        min = 999;
        for (i = 1; i <= n; i++)
            for (j = 1; j <= n; j++)
                if (cost[i][j] < min) {
                    min = cost[i][j];
                    a = u = i;
                    b = v = j;
                }
        u = find(parent, u);
        v = find(parent, v);
        if (uni(parent, u, v)) {
            printf("Edge %d: (%d -> %d) cost = %d\n", ne++, a, b, min);
            mincost += min;
        }
        cost[a][b] = cost[b][a] = 999;
    }

    printf("Minimum cost = %d\n", mincost);
    return 0;
}
```

```
Output
Enter number of vertices: 2
Enter the cost adjacency matrix (use 999 for no edge):
34
23
99
67
Edge 1: (1 -> 2) cost = 23
Minimum cost = 23


=== Code Execution Successful ===6
```

Comparison:-

Algorithm	**Time Complexity**	**Space Complexity**	**Approach**
Prim's	$O(V^2)$ (simple array) or O(E log V) (using heap)	$O(V^2)$	Greedy vertex-based
Kruskal's	O(E log E)	O(E + V)	Greedy edge-based

Feature	**Prim's Algorithm**	**Kruskal's Algorithm**
Type	Vertex-based	Edge-based
Sorting	Not required	Required
Data structure used	Adjacency matrix	Edge list and union-find
Efficiency	Better for dense graphs	Better for sparse graphs
Cycle check	Implicit	Explicit using union-find

Algorithm	Time Complexity	Space Complexity	Approach
Complexity	$O(V^2)$	O(E log E)	

Conclusion :-

Both Prim's and Kruskal's algorithms use the greedy approach to find the Minimum Cost Spanning Tree of a connected undirected graph. Prim's algorithm grows the MST from a starting vertex, while Kruskal's algorithm selects the smallest edges to connect disjoint trees. Prim's is efficient for dense graphs, whereas Kruskal's is more efficient for sparse graphs. Both algorithms produce the same final result — a spanning tree with the minimum possible total edge cost.

Experiment No.6

Roll No.24141014

Title :-

Shortest paths to other vertices using Dijkstra's algorithm from a given vertex in a weighted connected graph

Aim :-

To implement Dijkstra’s algorithm to find the shortest paths from a given source vertex to all other vertices in a weighted connected graph.

Objective :-

- To understand the concept of single-source shortest path problems.
- To apply Dijkstra’s algorithm for finding shortest paths in a weighted graph.
- To analyze the efficiency of the algorithm in terms of time and space complexity.
- To study the applications of Dijkstra’s algorithm in real-world problems.

Theory :-

In graph theory, the **shortest path problem** is to find the path between two vertices such that the total weight (or cost) of its edges is minimized.

Dijkstra’s Algorithm is a **greedy algorithm** used to find the shortest paths from a single source vertex to all other vertices in a weighted, connected graph with **non-negative edge weights**.

The algorithm maintains a set of vertices whose shortest distance from the source is already known. It repeatedly selects the vertex with the **minimum tentative distance** and updates the distances of its neighboring vertices.

Steps involved:

1. Assign a tentative distance value to every vertex: 0 for the source vertex and infinity for all others.
2. Mark all vertices as unvisited. Set the source as the current vertex.

3. For the current vertex, consider all its unvisited neighbors and calculate their tentative distances through the current vertex.
4. Update the shortest known distance if the newly calculated distance is smaller.
5. After examining all neighbors, mark the current vertex as visited.
6. Select the unvisited vertex with the smallest tentative distance and repeat the process.
7. Continue until all vertices are visited or the shortest paths are determined.

-APPLICATIONS:

1. Used in **network routing protocols** (e.g., OSPF and IS-IS).
2. GPS navigation systems for finding the **shortest driving routes**.
3. **Transportation and logistics** for optimal delivery routes.
4. **Internet data packet routing** and bandwidth optimization.
5. **Robotics** and **AI pathfinding** for obstacle avoidance.

-ALGORITHM:

Algorithm: Dijkstra(G, source)

Input: Weighted connected graph G(V, E), source vertex s.
Output: Shortest distances from source s to all other vertices.

Steps:

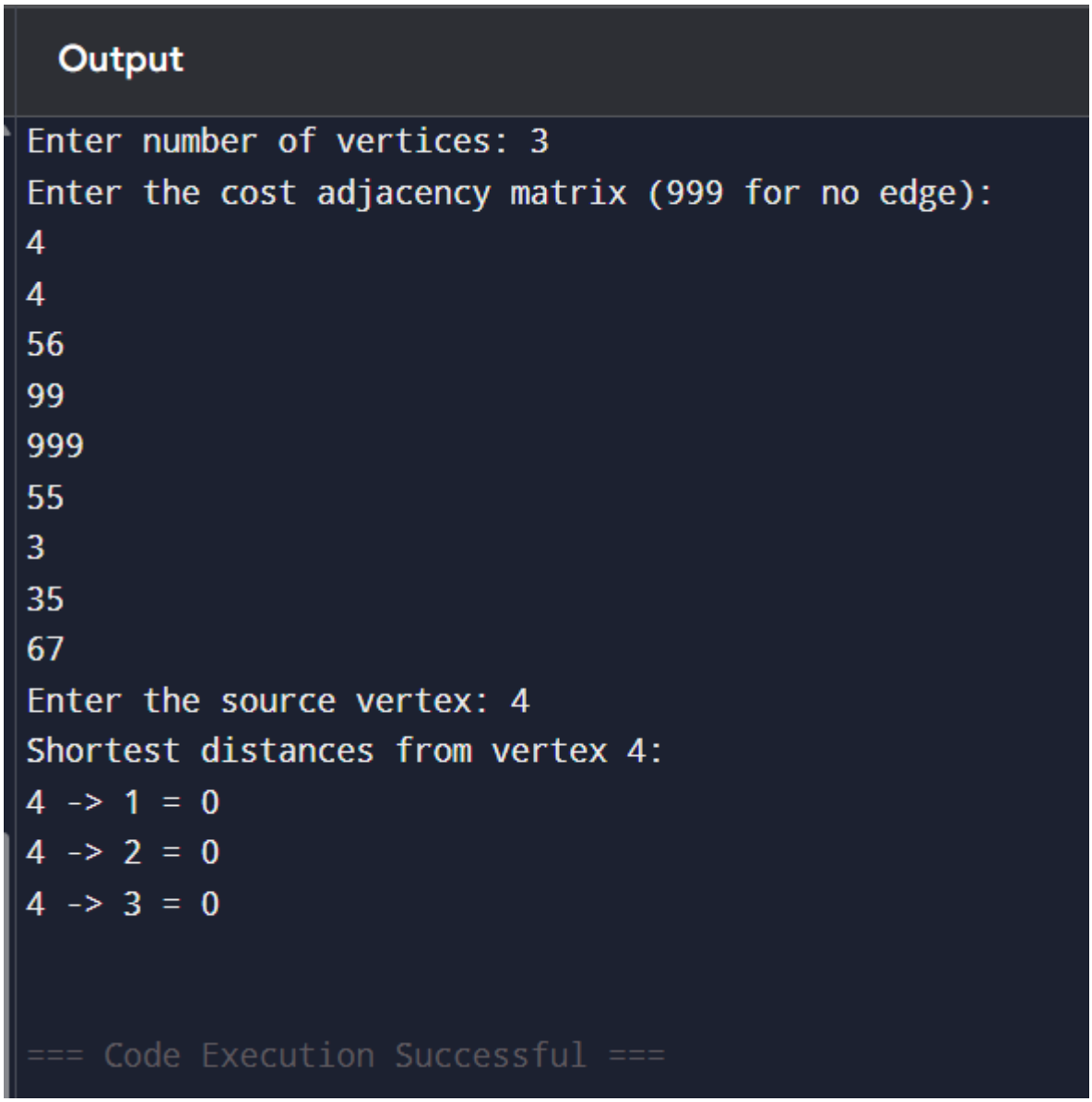
1. For each vertex v in V:
   distance[v] = ∞
   visited[v] = false
2. distance[source] = 0
3. Repeat (V - 1) times:
   a. Select the vertex u with the minimum distance value that is not visited.
   b. Mark u as visited.
   c. For each neighbor v of u:
      If distance[u] + weight(u, v) < distance[v], then
         distance[v] = distance[u] + weight(u, v)
4. Print distance[] array as the shortest distances from source to all vertices.

Program :-

```
#include <stdio.h>
#define INF 999

void dijkstra(int n, int cost[10][10], int source) {
   int distance[10], visited[10], count, min, next, i, j;

   for (i = 1; i <= n; i++) {
      distance[i] = cost[source][i];
      visited[i] = 0;
   }

   distance[source] = 0;
   visited[source] = 1;
   count = 1;

   while (count < n - 1) {
      min = INF;
      for (i = 1; i <= n; i++)
         if (distance[i] < min && !visited[i]) {
            min = distance[i];
            next = i;
         }

      visited[next] = 1;
      for (i = 1; i <= n; i++)
         if (!visited[i])
            if (min + cost[next][i] < distance[i])
```

```
                distance[i] = min + cost[next][i];

        count++;
    }

    printf("Shortest distances from vertex %d:\n", source);
    for (i = 1; i <= n; i++)
        if (i != source)
            printf("%d -> %d = %d\n", source, i, distance[i]);
}

int main() {
    int n, i, j, source;
    int cost[10][10];

    printf("Enter number of vertices: ");
    scanf("%d", &n);

    printf("Enter the cost adjacency matrix (999 for no edge):\n");
    for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
            scanf("%d", &cost[i][j]);

    printf("Enter the source vertex: ");
    scanf("%d", &source);

    dijkstra(n, cost, source);
    return 0;
```

}



TIME AND SPACE COMPLEXITY:

- Time Complexity:
  $O(V^2)$ using adjacency matrix
  $O((V + E) \log V)$ using min-heap/priority queue
- Space Complexity: O(V)

Conclusion :-

Dijkstra's algorithm efficiently finds the shortest paths from a single source vertex to all other vertices in a weighted graph with non-negative edge weights. It uses the greedy

approach by repeatedly selecting the vertex with the smallest tentative distance. The algorithm is widely used in network routing, navigation systems, and optimization applications due to its efficiency and simplicity.

Experiment No.7

Roll no.24141014

Title :-

Optimal binary search trees using Dynamic Programming.

Aim :-

To construct an Optimal Binary Search Tree (OBST) using Dynamic Programming and to find its minimum cost.

Objective :-

- To understand the concept of binary search trees and their average search cost.

- To design and implement an algorithm to construct an Optimal Binary Search Tree using Dynamic Programming.

-To analyze the time and space complexity of the algorithm.

-To study the applications of OBST in searching and information retrieval systems.

Theory :-

A **Binary Search Tree (BST)** is a data structure where each node contains a key, and every key in the left subtree is less than the node's key, while every key in the right subtree is greater.

When performing searches on a BST, the **expected cost** (number of comparisons) depends on the structure of the tree. For a given set of keys with different access probabilities, a normal BST may not be optimal. Therefore, the **Optimal Binary Search Tree (OBST)** is constructed to minimize the **expected search cost**.

Let there be *n* distinct sorted keys, K1, K2, K3, …, Kn, each having a probability of being searched P1, P2, …, Pn.
Additionally, there are *n+1* dummy keys D0, D1, …, Dn representing unsuccessful searches, with probabilities Q0, Q1, …, Qn.

The goal is to arrange the keys into a binary search tree such that the **expected cost of all searches (successful and unsuccessful)** is minimized.

Cost Formula:
Let C[i][j] represent the cost of the optimal BST that contains keys Ki to Kj.
Then,

C[i][j] = W[i][j] + min (C[i][r-1] + C[r+1][j]) for all r = i to j

where
W[i][j] = sum(P[i] to P[j]) + sum(Q[i-1] to Q[j])

The dynamic programming approach fills the cost and weight matrices in increasing order of chain length to find the optimal root and minimum cost.

-APPLICATIONS:

1. Used in **compilers** for optimal syntax parsing.
2. Efficient **dictionary lookup** and word suggestion systems.
3. **Search optimization** in databases.
4. Used in **data compression** and **predictive text** applications.
5. Ideal for **information retrieval systems** where some items are accessed more frequently than others.

-ALGORITHM:

Algorithm: Optimal Binary Search Tree (Dynamic Programming)

Input:
n keys K1 < K2 < ... < Kn
Probabilities of successful search: P1, P2, ..., Pn
Probabilities of unsuccessful search: Q0, Q1, ..., Qn

Output:
Minimum cost of Optimal Binary Search Tree.

Steps:

1. For i = 1 to n + 1
   C[i][i-1] = 0
   W[i][i-1] = Q[i-1]
2. For l = 1 to n (length of chain)
   For i = 1 to n - l + 1
   j = i + l - 1
   W[i][j] = W[i][j-1] + P[j] + Q[j]
   C[i][j] = ∞
   For r = i to j
   temp = C[i][r-1] + C[r+1][j] + W[i][j]

If temp < C[i][j], then
C[i][j] = temp
Root[i][j] = r

3. The minimum cost of OBST is C[1][n].

Program :-

```
#include <stdio.h>
#include <limits.h>

#define MAX 10

int main() {
   int n, i, j, k, l, r;
   float p[MAX], q[MAX], w[MAX][MAX], c[MAX][MAX];
   int root[MAX][MAX];

   printf("Enter number of keys: ");
   scanf("%d", &n);

   printf("Enter probabilities of successful search (p1 to pn):\n");
   for (i = 1; i <= n; i++)
      scanf("%f", &p[i]);

   printf("Enter probabilities of unsuccessful search (q0 to qn):\n");
   for (i = 0; i <= n; i++)
      scanf("%f", &q[i]);

   for (i = 1; i <= n + 1; i++) {
      c[i][i - 1] = 0;
```

```
    w[i][i - 1] = q[i - 1];
}

for (l = 1; l <= n; l++) {
    for (i = 1; i <= n - l + 1; i++) {
        j = i + l - 1;
        w[i][j] = w[i][j - 1] + p[j] + q[j];
        c[i][j] = INT_MAX;

        for (r = i; r <= j; r++) {
            float t = c[i][r - 1] + c[r + 1][j] + w[i][j];
            if (t < c[i][j]) {
                c[i][j] = t;
                root[i][j] = r;
            }
        }
    }
}

printf("Minimum cost of Optimal BST: %.2f\n", c[1][n]);
printf("Root matrix:\n");
for (i = 1; i <= n; i++) {
    for (j = i; j <= n; j++)
        printf("%d ", root[i][j]);
    printf("\n");
}

return 0;
```

}

Output

```
Enter number of keys: 2
Enter probabilities of successful search (p1 to pn):
4
6
Enter probabilities of unsuccessful search (q0 to qn):
6
7
3
Minimum cost of Optimal BST: 42.00
Root matrix:
1 1
2
```

TIME AND SPACE COMPLEXITY:

- Time Complexity: $O(n^3)$
- Space Complexity: $O(n^2)$

Conclusion :-

The Optimal Binary Search Tree problem demonstrates the power of Dynamic Programming in optimization problems. By systematically evaluating all possible roots and subtrees, the OBST minimizes the average search cost based on access probabilities. It is particularly useful in applications where some elements are accessed more frequently than others, such as compilers, dictionaries, and database indexing systems.

Experiment No.8

Roll no.24141014

Title :-

All-Pairs Shortest Paths Problem using Floyd's algorithm.

Aim :-

To implement Floyd’s algorithm to find the shortest paths between all pairs of vertices in a given weighted graph.

Objective :-

- To understand the concept of finding the shortest paths between every pair of vertices in a weighted graph.
- To apply Floyd’s algorithm for solving the all-pairs shortest path problem.
- To analyze the efficiency of Floyd’s algorithm in terms of time and space complexity.
- To study the applications of this algorithm in real-world systems such as network routing and transportation planning.

Theory :-

The **All-Pairs Shortest Path (APSP)** problem aims to find the shortest possible path between every pair of vertices in a weighted, directed graph.

Floyd’s Algorithm, also known as **Floyd–Warshall Algorithm**, is a **Dynamic Programming** approach used to solve this problem. It is efficient and works for both directed and undirected graphs, provided there are no negative cycles.

The algorithm systematically improves the estimate of the shortest path between any two vertices by considering all possible intermediate vertices one by one.

If D[i][j] represents the shortest distance from vertex i to vertex j, and there are n vertices, the algorithm follows this recursive relation:

D[i][j] = min(D[i][j], D[i][k] + D[k][j])

where k is an intermediate vertex being considered.
Initially, D[i][j] is initialized with the direct edge weight between i and j (or ∞ if no direct edge exists).

The final matrix D gives the shortest path distances between all pairs of vertices.

-APPLICATIONS:

1. Used in **network routing** for finding the shortest communication paths.
2. **Transportation and logistics planning** for minimizing travel distance or time.
3. **Geographical mapping systems** (like GPS navigation).
4. **Social network analysis** to determine shortest connections between individuals.
5. **Urban traffic management** and **airline route optimization**.

-ALGORITHM:

Algorithm: Floyd's All-Pairs Shortest Path Algorithm

Input: Weighted directed graph with vertices V = {1, 2, …, n} and edge weights W[i][j].
Output: Matrix D containing shortest path distances between all pairs of vertices.

Steps:

1. Initialize D[i][j] = W[i][j] for all i, j.
2. For k = 1 to n do
   For i = 1 to n do
   For j = 1 to n do
   If D[i][k] + D[k][j] < D[i][j], then
   D[i][j] = D[i][k] + D[k][j]
3. Print the final matrix D as the shortest distances between every pair of vertices.

Program :-

```
#include <stdio.h>

#define INF 999

int main() {
   int n, i, j, k;
   int D[10][10]
   printf("Enter number of vertices: ");
```

```
    scanf("%d", &n);
    printf("Enter the cost adjacency matrix (use 999 for infinity):\n");
    for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
            scanf("%d", &D[i][j]);
    for (k = 1; k <= n; k++) {
        for (i = 1; i <= n; i++) {
            for (j = 1; j <= n; j++) {
                if (D[i][k] + D[k][j] < D[i][j])
                    D[i][j] = D[i][k] + D[k][j];
            }
        }
    }
    printf("\nAll-Pairs Shortest Path Matrix:\n");
    for (i = 1; i <= n; i++) {
        for (j = 1; j <= n; j++) {
            if (D[i][j] == INF)
                printf("INF\t");
            else
                printf("%d\t", D[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

```
Output

Enter number of vertices: 3
Enter the cost adjacency matrix (use 999 for infinity):
2
5
999
6
8
9
4
999
4

All-Pairs Shortest Path Matrix:
2   5   14
6   8   9
4   9   4


=== Code Execution Successful ===
```

TIME AND SPACE COMPLEXITY:

- Time Complexity: $O(n^3)$
- Space Complexity: $O(n^2)$

Conclusion :-

Floyd's algorithm efficiently finds the shortest paths between all pairs of vertices in a weighted graph. It uses the principle of dynamic programming by considering all possible intermediate vertices. The algorithm is simple and widely used in applications like network routing, path optimization, and transportation systems. However, its cubic time complexity makes it better suited for graphs with a smaller number of vertices.

Experiment No.9

Roll No.24141014

Title :-

Single Source Shortest Path Problem

Aim :-

To find the shortest paths from a single source vertex to all other vertices in a weighted connected graph using Dijkstra's algorithm.

Objective :-

1. To understand the concept of shortest path computation in a weighted graph.
2. To implement Dijkstra's algorithm for finding the shortest path from a given source vertex.
3. To analyze the efficiency of the algorithm in terms of time and space complexity.
4. To study real-world applications of shortest path algorithms in computer science and engineering.

Theory :-

The **Single Source Shortest Path (SSSP)** problem determines the shortest paths from a given source vertex to all other vertices in a graph such that the total weight of the path is minimized.

There are several algorithms to solve this problem, such as **Dijkstra's algorithm**, **Bellman-Ford algorithm**, and **Breadth-First Search (for unweighted graphs)**.

Dijkstra's Algorithm is a **greedy method** that works for graphs with non-negative edge weights. It maintains two sets of vertices:

- Visited vertices (whose shortest path is known).
- Unvisited vertices (whose shortest path is yet to be determined).

At each step, the algorithm picks the vertex with the **minimum tentative distance** and updates the distances to its neighbors.

Working Principle:

1. Initialize all vertex distances as infinity, except the source vertex, which is 0.
2. Mark all vertices as unvisited.
3. Pick the vertex with the smallest tentative distance.
4. Update the distance to each unvisited neighbor if a shorter path is found.
5. Mark the current vertex as visited.
6. Repeat until all vertices are visited.

APPLICATIONS:

1. **Network routing protocols** (e.g., OSPF and IS-IS).
2. **GPS and mapping systems** to find the shortest driving route.
3. **Telecommunication networks** to minimize transmission delay.
4. **Transportation and logistics** route optimization.
5. **Artificial Intelligence** for pathfinding in games and robotics.

ALGORITHM:

Algorithm: Dijkstra's Single Source Shortest Path

Input: Weighted graph G(V, E) and source vertex S.
Output: Shortest distance from S to every other vertex.

Steps:

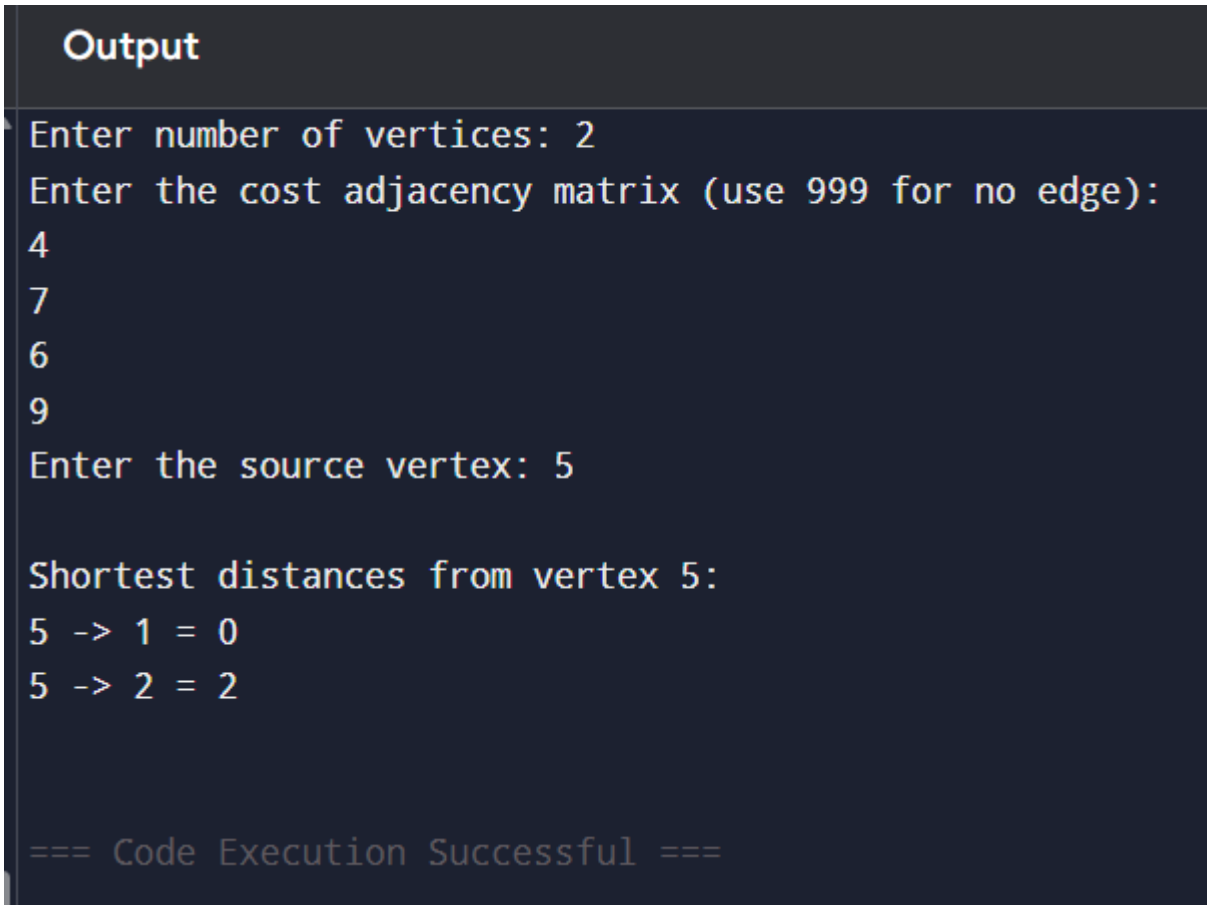
1. For each vertex v in V:
   distance[v] = ∞
   visited[v] = false
2. distance[source] = 0
3. Repeat (V - 1) times:
   a. Select the vertex u with the smallest distance that is unvisited.
   b. Mark u as visited.
   c. For each neighbor v of u:
      If distance[u] + weight(u, v) < distance[v], then
         distance[v] = distance[u] + weight(u, v)
4. Print the distance array containing shortest distances from the source.

Program :-

```
#include <stdio.h>
#define INF 999

void dijkstra(int n, int cost[10][10], int source) {
   int distance[10], visited[10], count, min, next, i, j;

   for (i = 1; i <= n; i++) {
      distance[i] = cost[source][i];
      visited[i] = 0;
   }

   distance[source] = 0;
   visited[source] = 1;
   count = 1;

   while (count < n - 1) {
      min = INF;
      for (i = 1; i <= n; i++)
         if (distance[i] < min && !visited[i]) {
            min = distance[i];
            next = i;
         }

      visited[next] = 1;
      for (i = 1; i <= n; i++)
         if (!visited[i])
            if (min + cost[next][i] < distance[i])
```

```
                distance[i] = min + cost[next][i];

        count++;
    }

    printf("\nShortest distances from vertex %d:\n", source);
    for (i = 1; i <= n; i++)
        if (i != source)
            printf("%d -> %d = %d\n", source, i, distance[i]);
}

int main() {
    int n, i, j, source;
    int cost[10][10];

    printf("Enter number of vertices: ");
    scanf("%d", &n);

    printf("Enter the cost adjacency matrix (use 999 for no edge):\n");
    for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
            scanf("%d", &cost[i][j]);

    printf("Enter the source vertex: ");
    scanf("%d", &source);

    dijkstra(n, cost, source);
    return 0;
```

}



TIME AND SPACE COMPLEXITY:

- Time Complexity:
  $O(V^2)$ for adjacency matrix representation
  $O((V + E) \log V)$ if implemented with a min-heap or priority queue
- Space Complexity: O(V)

Conclusion :-

Dijkstra's algorithm provides an efficient way to find the shortest paths from a single source vertex to all other vertices in a weighted graph with non-negative edge weights. It follows a greedy approach to progressively build the shortest paths. The algorithm is widely used in applications such as network routing, map navigation, and resource optimization systems.

Experiment No.10

Roll No.24141014

Title :-

8-Queen's problem using Back Tracking

Aim :-

To place 8 queens on a chessboard such that no two queens attack each other using the Backtracking method.

Objective :-

- To understand the concept of the N-Queens problem as a classic combinatorial problem.
- To implement a backtracking algorithm to find all possible solutions for placing 8 queens on a chessboard.
- To analyze the efficiency of the backtracking method in solving constraint satisfaction problems.
- To explore applications of backtracking in problem-solving and artificial intelligence.

Theory :-

The **8-Queens Problem** is a special case of the N-Queens problem, where the goal is to place 8 queens on an 8×8 chessboard such that no two queens threaten each other. A queen can attack any piece located on the same **row, column, or diagonal**.

The **backtracking approach** systematically searches for solutions by placing a queen in a row and then recursively attempting to place queens in subsequent rows. If a placement leads to a conflict, the algorithm **backtracks** to the previous row to try a different position. This eliminates invalid solutions early and reduces computation time.

Key Conditions for Safety:

- No two queens share the same row.
- No two queens share the same column.
- No two queens share the same diagonal.

Backtracking is ideal for problems with constraints because it explores all possibilities while pruning invalid choices.

APPLICATIONS:

1. Solving **constraint satisfaction problems** in AI.
2. **Puzzle solving** such as Sudoku, crosswords, and magic squares.
3. **Combinatorial optimization** in scheduling and resource allocation.
4. **Decision-making algorithms** in robotics and games.
5. **Graph coloring** and other mathematical problems.

ALGORITHM:

Algorithm: 8-Queens Problem using Backtracking

Input: 8×8 chessboard
Output: Positions of 8 queens such that no two queens attack each other

Steps:

1. Start in the first row.
2. Place a queen in the first column of the current row.
3. Check if the queen is safe (no other queens in the same column, left diagonal, right diagonal).
    a. If safe, move to the next row and repeat step 2.
    b. If not safe, try the next column in the current row.
4. If all columns are tried and no safe position is found, backtrack to the previous row.
5. Repeat until all rows are filled.
6. Print the solution.
7. Continue to find all possible solutions.

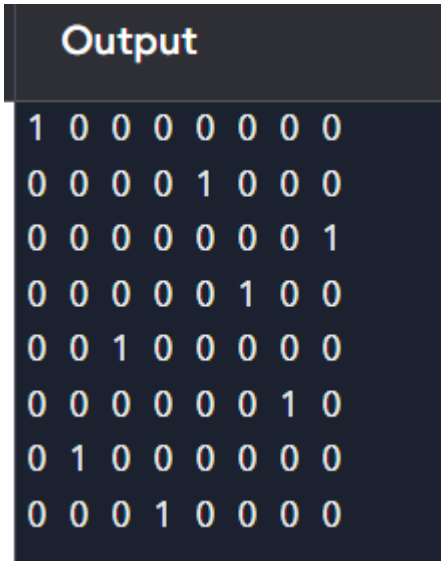
Program :-

```
#include <stdio.h>

#define N 8


int board[N][N];
```

```
int isSafe(int row, int col) {
    int i, j;

    // Check column above
    for (i = 0; i < row; i++)
        if (board[i][col])
            return 0;

    // Check upper-left diagonal
    for (i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return 0;

    // Check upper-right diagonal
    for (i = row - 1, j = col + 1; i >= 0 && j < N; i--, j++)
        if (board[i][j])
            return 0;

    return 1;
}

void printBoard() {
    int i, j;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++)
            printf("%d ", board[i][j]);
        printf("\n");
```

```
    }
    printf("\n");
}

int solve(int row) {
    if (row >= N) {
        printBoard();
        return 1; // solution found
    }
    int count = 0;
    for (int col = 0; col < N; col++) {
        if (isSafe(row, col)) {
            board[row][col] = 1;
            count += solve(row + 1);
            board[row][col] = 0; // backtrack
        }
    }
    return count;
}

int main() {
    int totalSolutions;
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            board[i][j] = 0;

    totalSolutions = solve(0);
    printf("Total solutions: %d\n", totalSolutions);
```

```
    return 0;
}
```



TIME AND SPACE COMPLEXITY:

- Time Complexity: O(N!) (exponential in the number of queens)
- Space Complexity: $O(N^2)$ for the board

Conclusion :-

The 8-Queens problem illustrates the power of the backtracking technique for constraint satisfaction problems. Backtracking efficiently explores possible placements, pruning invalid solutions early. This approach is widely applicable in puzzles, AI, and combinatorial optimization problems.