**EXPERIMENT 7 : ALL PAIR SHORTEST PATH FLOYD WARSHALL ALGORITHM**

**AIM:** Write a program to determine shortest distances between every pair of vertices in a given edge weighted directed Graph.

graph[][] = { {0, 5, INF, 10},

{INF, 0, 3, INF},

{INF, INF, 0, 1},

{INF, INF, INF, 0} }

Note that the value of graph[i][j] is 0 if i is equal to j

And graph[i][j] is INF (infinite) if there is no edge from vertex i to j.

**THEORY:** Given data represents the following graph

10

(0)------->(3)

| /|\

5 | |

| | 1

\|/ |

(1)------->(2)

3

**ALGORITHM STEPS: Refer to notes given during offline class**

We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices. For every pair (i, j) of the source and destination vertices respectively, there are two possible cases. 

1. k is not an intermediate vertex in shortest path from i to j. We keep the value of

dist[i][j] as it is.   
**2)** k is an intermediate vertex in shortest path from i to j. We update the value of

dist[i][j] as ( dist[i][k] + dist[k][j] ) if (dist[i][j] ) > ( dist[i][k] + dist[k][j])

**PROG:**

#include<stdio.h>

  // Number of vertices in the graph

#define V 4

/\* Define Infinite as a large enough  value. This value will be used for vertices not connected to each other \*/

#define INF 99999

  // A function to print the solution matrix

**void** printSolution(**int** dist[][V]);

// Solves the all-pairs shortest path

**void** floydWarshall (**int** graph[][V])

{

/\* dist[][] will be the output matrix that will finally have the shortest distances between every pair of vertices \*/

**int** dist[V][V], i, j, k;

/\* Initialize the solution matrix same as input graph matrix. Or we can say the initial values of shortest distances are based on shortest paths considering no

        intermediate vertex. \*/

**for** (i = 0; i < V; i++)

**for** (j = 0; j < V; j++)

            dist[i][j] = graph[i][j];

/\* Add all vertices one by one to the set of intermediate vertices.

---> Before start of an iteration, we have shortest distances between all pairs of vertices such that the shortest distances consider only the vertices in set {0, 1, 2, .. k-1} as intermediate vertices.

----> After the end of an iteration, vertex no. k is added to the set Of intermediate vertices and the set

       becomes {0, 1, 2, .. k} \*/

**for** (k = 0; k < V; k++)

    {

         // Pick all vertices as source one by one

**for** (i = 0; i < V; i++)

        {

          // Pick all vertices as destination for the above picked source

**for** (j = 0; j < V; j++)

            {

                // If vertex k is on the shortest path from i to j, then

//update the value of dist[i][j]

**if** (dist[i][k] + dist[k][j] < dist[i][j])

                    dist[i][j] = dist[i][k] + dist[k][j];

            }

        }

    }

     // Print the shortest distance matrix

    printSolution(dist);

}

/\* A function to print solution \*/

**void** printSolution(**int** dist[][V])

{

**printf** ("The following matrix shows the shortest distances"

            " between every pair of vertices \n");

**for** (**int** i = 0; i < V; i++)

    {

**for** (**int** j = 0; j < V; j++)

        {

**if** (dist[i][j] == INF)

**printf**("%7s", "INF");

**else**

**printf** ("%7d", dist[i][j]);

        }

**printf**("\n");

    }

}

**int** main()

{

     /\* create the weighted graph\*/

**int** graph[V][V] = { {0,   5,  INF, 10},

                        {INF, 0,   3, INF},

                        {INF, INF, 0,   1},

                        {INF, INF, INF, 0}

                      };

     // Print the solution

    floydWarshall(graph);

**return** 0;

}

**Output:**

Following matrix shows the shortest distances between every pair of vertices

0 5 8 9

INF 0 3 4

INF INF 0 1

INF INF INF 0

**TIME COMPLEXITY: O(V^3) where V=number of vertices**

**EXPERIMENT 8 :LONGEST COMMON SUBSEQUENCE**

**AIM:** Write a program to obtain longest common subsequence of the two sequences

**A= stone and B = longest**

**THEORY:** Refer notes from pdf shared in classroom also solve

problem as given in pdf

**PSEUDOCODE:**

**int** max (**int** a, **int** b);

/\* Returns length of LCS for A[] and B[] \*/

int lcs( char \*A, char \*B, int i, int j )

{

    if (i == 0 || j == 0)

        return 0;

    if (A[i] == B[j])

        return 1 + lcs(A, B, i-1, j-1);

    else

        return max(lcs(A, B, i, j-1), lcs(A, B, i-1, j));

}

/\* function to get max of 2 integers \*/

**int** max(**int** a, **int** b)

{

**return** (a > b)? a : b;

}

/\* main program to test above function \*/

int main()

{

    char A[] = “longest";

    char B[] = “stone";

    int i = strlen(A);

    int j = strlen(B);

    cout<<"Length of LCS is "<< lcs( A, B, i, j ) ;

    return 0;

}

**PROG: Attach Print**

**Output:  Length of LCS is 3**

**Attach Print**

**TIME COMPLEXITY:** If m 🡺 length of A and

n 🡺 length of B then

Time Complexity of LCS will be **O(m n)**