Floyd Warshall Algorithm

The Floyd Warshall Algorithm is for solving all pairs of shortest-path problems. The problem is to find the shortest distances between every pair of vertices in a given edge-weighted directed Graph.

It is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. This algorithm follows the dynamic programming approach to find the shortest path.

A C-function for a N x N graph is given below. The function stores the all pair shortest path in the matrix cost [N][N]. The cost matrix of the given graph is available in cost Mat [N][N].

Example:

```
Input:  graph[][] = { {0,   5,  INF, 10},
                      {INF,  0,  3,  INF},
                      {INF, INF, 0,   1},
                      {INF, INF, INF, 0} }
which represents the following graph
             10
       (0)------->(3)
        |         /|\
      5 |          |  1
        |          |
       \|/         |
       (1)------->(2)
             3
Output: Shortest distance matrix
   0      5      8      9
 INF      0      3      4
 INF    INF      0      1
 INF    INF    INF      0
```

Recommended Problem
Floyd Warshall

Dynamic Programming

Graph

+2 more

Samsung

Solve Problem

Submission count: 86.7K

Floyd Warshall Algorithm:

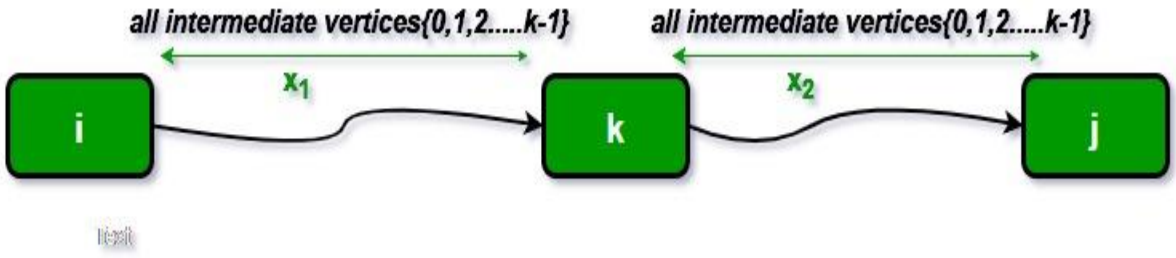
```
# define N 4

void floydwarshall()
{
int cost [N][N];
int i, j, k;
for(i=0; i<N; i++)
for(j=0; j<N; j++)
cost [i][j]= cost Mat [i] [i];
for(k=0; k<N; k++)
{
for(i=0; i<N; i++)
for(j=0; j<N; j++)
if(cost [i][j]> cost [i] [k] + cost [k][j];
cost [i][j]=cost [i] [k]+'cost [k] [i]:
}

//display the matrix cost [N] [N]
}
```

- Initialize the solution matrix same as the input graph matrix as a first step.
- Then 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.
    - k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.
    - 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]

The following figure shows the above optimal substructure property in the all-pairs shortest path problem.



Below is the implementation of the above approach:

```cpp
// C++ Program for Floyd Warshall Algorithm
#include <bits/stdc++.h>
using namespace std;

// 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
// problem using Floyd Warshall algorithm
void floydWarshall(int dist[][V])
{

    int i, j, k;

    /* 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][j] > (dist[i][k] + dist[k][j])
                    && (dist[k][j] != INF
                        && dist[i][k] != INF))
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }

    // Print the shortest distance matrix
    printSolution(dist);
}

/* A utility function to print solution */
void printSolution(int dist[][V])
{
    cout << "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)
                cout << "INF"
                     << " ";
            else
                cout << dist[i][j] << "   ";
        }
        cout << endl;
    }
}

// Driver's code
int main()
{
    /* Let us create the following weighted graph
            10
    (0)------->(3)
        |      /|\
    5  |       |
        |      | 1
    \|/        |
    (1)------->(2)
            3      */
    int graph[V][V] = { { 0, 5, INF, 10 },
                        { INF, 0, 3, INF },
                        { INF, INF, 0, 1 },
                        { INF, INF, INF, 0 } };

    // Function call
    floydWarshall(graph);
    return 0;
}

// This code is contributed by Mythri J L
```

Output

```
The 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: $O(V^3)$
Auxiliary Space: $O(V^2)$

The above program only prints the shortest distances. We can modify the solution to print the shortest paths also by storing the predecessor information in a separate 2D matrix.

Also, the value of INF can be taken as INT_MAX from limits.h to make sure that we handle the maximum possible value. When we take INF as INT_MAX, we need to change the if condition in the above program to avoid arithmetic overflow.