Floyd Warshall Algorithm

The **Floyd Warshall Algorithm i**s for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

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
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.

Output:

```
Shortest distance matrix
   0      5      8      9
 INF      0      3      4
 INF    INF      0      1
 INF    INF    INF      0
```

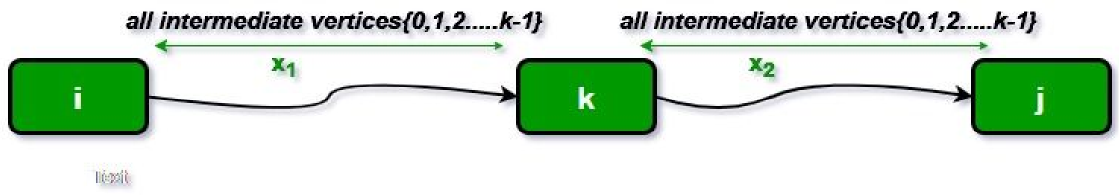
Floyd Warshall Algorithm:

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].

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


all intermediate vertices{0,1,2.....k-1}
all intermediate vertices{0,1,2.....k-1}
x_1
x_2
i
k
j
text

Following is implementations of the Floyd Warshall algorithm.

Code in “C”

```
// A C program for Bellman-Ford's single source
// shortest path algorithm.
#include <bits/stdc++.h>

// a structure to represent a weighted edge in graph
struct Edge
{
    int src, dest, weight;
};
```

```
// a structure to represent a connected, directed and
// weighted graph
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges.
    struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[E];
    return graph;
}

// A utility function used to print the solution
void printArr(int dist[], int n)
{
    printf("Vertex   Distance from Source\n");
    for (int i = 0; i < n; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}

// The main function that finds shortest distances from src to
// all other vertices using Bellman-Ford algorithm.  The function
// also detects negative weight cycle
void BellmanFord(struct Graph* graph, int src)
{
    int V = graph->V;
    int E = graph->E;
    int dist[V];
    // Step 1: Initialize distances from src to all other vertices
    // as INFINITE
    for (int i = 0; i < V; i++)
        dist[i]   = INT_MAX;
    dist[src] = 0;

    // Step 2: Relax all edges |V| - 1 times. A simple shortest
    // path from src to any other vertex can have at-most |V| - 1
    // edges
    for (int i = 1; i <= V-1; i++)
    {
        for (int j = 0; j < E; j++)
        {
            int u = graph->edge[j].src;
            int v = graph->edge[j].dest;
            int weight = graph->edge[j].weight;
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
                dist[v] = dist[u] + weight;
        }
    }

    // Step 3: check for negative-weight cycles.  The above step
    // guarantees shortest distances if graph doesn't contain
    // negative weight cycle.  If we get a shorter path, then there
    // is a cycle.
    for (int i = 0; i < E; i++)
    {
        int u = graph->edge[i].src;
        int v = graph->edge[i].dest;
        int weight = graph->edge[i].weight;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
            printf("Graph contains negative weight cycle");
    }

    printArr(dist, V);

    return;
}

int main()
```

```
{
    /* Let us create the graph given in above example */
    int V = 5;  // Number of vertices in graph
    int E = 8;  // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1 (or A-B in above figure)
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = -1;

    // add edge 0-2 (or A-C in above figure)
    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 4;

    // add edge 1-2 (or B-C in above figure)
    graph->edge[2].src = 1;
    graph->edge[2].dest = 2;
    graph->edge[2].weight = 3;

    // add edge 1-3 (or B-D in above figure)
    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;
    graph->edge[3].weight = 2;

    // add edge 1-4 (or A-E in above figure)
    graph->edge[4].src = 1;
    graph->edge[4].dest = 4;
    graph->edge[4].weight = 2;

    // add edge 3-2 (or D-C in above figure)
    graph->edge[5].src = 3;
    graph->edge[5].dest = 2;
    graph->edge[5].weight = 5;

    // add edge 3-1 (or D-B in above figure)
    graph->edge[6].src = 3;
    graph->edge[6].dest = 1;
    graph->edge[6].weight = 1;
    // add edge 4-3 (or E-D in above figure)
    graph->edge[7].src = 4;
    graph->edge[7].dest = 3;
    graph->edge[7].weight = -3;

    BellmanFord(graph, 0);

    return 0;
}
```

Code in “JAVA”

```
    int V, E;
    Edge edge[];

    // Creates a graph with V vertices and E edges
    Graph(int v, int e)
    {
         V = v;
        E = e;
        edge = new Edge[e];
        for (int i=0; i<e; ++i)
            edge[i] = new Edge();
    }

    // The main function that finds shortest distances from src
    // to all other vertices using Bellman-Ford algorithm.  The
    // function also detects negative weight cycle
    void BellmanFord(Graph graph,int src)
    {
        int V = graph.V, E = graph.E;
        int dist[] = new int[V];

        // Step 1: Initialize distances from src to all other
        // vertices as INFINITE
        for (int i=0; i<V; ++i)
            dist[i] = Integer.MAX_VALUE;
        dist[src] = 0;

        // Step 2: Relax all edges |V| - 1 times. A simple
        // shortest path from src to any other vertex can
        // have at-most |V| - 1 edges
        for (int i=1; i<V; ++i)
        {
            for (int j=0; j<E; ++j)
            {
                int u = graph.edge[j].src;
                int v = graph.edge[j].dest;
                int weight = graph.edge[j].weight;
                if (dist[u]!=Integer.MAX_VALUE &&
                    dist[u]+weight<dist[v])
                    dist[v]=dist[u]+weight;
            }
        }

        // Step 3: check for negative-weight cycles.  The above
        // step guarantees shortest distances if graph doesn't
        // contain negative weight cycle. If we get a shorter
        //  path, then there is a cycle.
        for (int j=0; j<E; ++j)
        {
            int u = graph.edge[j].src;
            int v = graph.edge[j].dest;
            int weight = graph.edge[j].weight;
            if (dist[u] != Integer.MAX_VALUE &&
                dist[u]+weight < dist[v])
              System.out.println("Graph contains negative weight cycle");
        }
        printArr(dist, V);
    }

    // A utility function used to print the solution
    void printArr(int dist[], int V)
    {
        System.out.println("Vertex   Distance from Source");
        for (int i=0; i<V; ++i)
            System.out.println(i+"\t\t"+dist[i]);
    }

    // Driver method to test above function
    public static void main(String[] args)
    {
        int V = 5;  // Number of vertices in graph
        int E = 8;  // Number of edges in graph

        Graph graph = new Graph(V, E);

        // add edge 0-1 (or A-B in above figure)
        graph.edge[0].src = 0;
        graph.edge[0].dest = 1;
        graph.edge[0].weight = -1;

        // add edge 0-2 (or A-C in above figure)
        graph.edge[1].src = 0;
        graph.edge[1].dest = 2;
        graph.edge[1].weight = 4;

        // add edge 1-2 (or B-C in above figure)
        graph.edge[2].src = 1;
        graph.edge[2].dest = 2;
        graph.edge[2].weight = 3;

        // add edge 1-3 (or B-D in above figure)
        graph.edge[3].src = 1;
        graph.edge[3].dest = 3;
        graph.edge[3].weight = 2;

        // add edge 1-4 (or A-E in above figure)
        graph.edge[4].src = 1;
        graph.edge[4].dest = 4;
        graph.edge[4].weight = 2;

        // add edge 3-2 (or D-C in above figure)
        graph.edge[5].src = 3;
        graph.edge[5].dest = 2;
        graph.edge[5].weight = 5;

        // add edge 3-1 (or D-B in above figure)
        graph.edge[6].src = 3;
        graph.edge[6].dest = 1;
        graph.edge[6].weight = 1;

        // add edge 4-3 (or E-D in above figure)
        graph.edge[7].src = 4;
        graph.edge[7].dest = 3;
        graph.edge[7].weight = -3;

        graph.BellmanFord(graph, 0);
    }
}
```