FLOYD - WARSHALL ALGORITHM

DIVYANSHU SHRIVASTAVA – 187909

MCA – 2ND YEAR

AGENDA

- INTRODUCTION
- HOW FLOYD-WARSHALL ALGORITHM WORKS?
- RECURRENCE RELATION
- RECURSIVE APPROACH
- TOP DOWN DYNAMIC PROGRAMMING(MEMORIZATION)
- BOTTOM-UP DYNAMIC PROGRAMMING
- IMPLEMENTATION

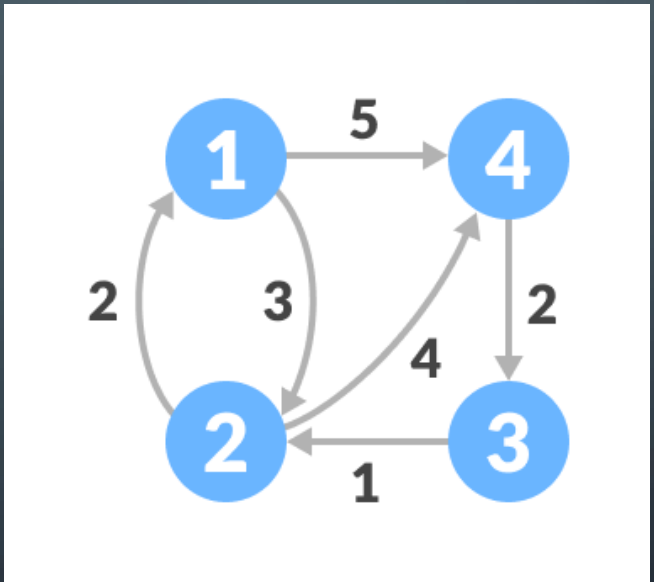
INTRODUCTION

Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. This algorithm works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).

- Floyd - Warshall algorithm is also called as
  - Floyd's algorithm
  - Roy-Floyd algorithm
  - Roy-Warshall algorithm
  - WFI algorithm.
- A weighted graph is a graph in which each edge has a numerical value associated with it.
- This algorithm follows the dynamic programming approach to find the shortest paths.

HOW FLOYD-WARSHALL ALGORITHM WORKS?

Let the given graph be:



STEP 1

Create a matrix A^0 of dimension n*n where n is the number of vertices. The row and the column are indexed as i and j respectively. i and j are the vertices of the graph.

Each cell A[i][j] is filled with the distance from the i^{th} vertex to the j^{th} vertex. If there is no path from i^{th} vertex to j^{th} vertex, the cell is left as infinity.

$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

STEP 2

Now create a matrix A^1 using matrix A^0. The elements in the first column and the first row are left as they are. The remaining cells are filled in the following way.

Let k be the intermediate vertex in the shortest path from source to destination. In this step, k is the first vertex. A[i][j] is filled with (A[i][k] + A[k][j]) if (A[i][j] > A[i][k] + A[k][j]).

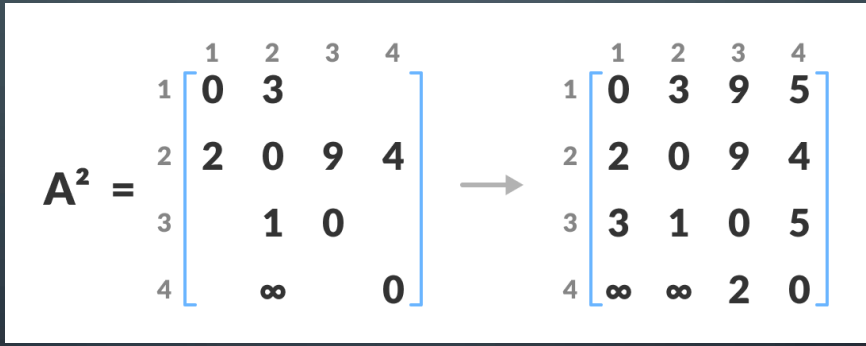
That is, if the direct distance from the source to the destination is greater than the path through the vertex k, then the cell is filled with A[i][k] + A[k][j].

In this step, k is vertex 1. We calculate the distance from source vertex to destination vertex through this vertex k.

$$A^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & & \\ \infty & & 0 & \\ \infty & & & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & 9 & 4 \\ \infty & 1 & 0 & 8 \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

STEP 3

In a similar way, A^2 is created using matrix A^3. The elements in the second column and the second row are left as they are.
In this step, k is vertex (i.e. vertex 2). The remaining steps are same as step 2.



STEP 4

In a similar way, A^3 and A^4 is also created. A4 gives the shortest path between each pair of vertices.

$$A^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & & \infty & \\ & 0 & 9 & \\ \infty & 1 & 0 & 8 \\ & & 2 & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 9 & 5 \\ 2 & 0 & 9 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

$$A^4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & & & 5 \\ & 0 & & 4 \\ & & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 7 & 5 \\ 2 & 0 & 6 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

RECURRENCE RELATION

- Assume we are trying to find a shortest path from *u* to *v* using a set of k vertices $\{w_1, ..., w_k\}$. Then either
- w_k is an intermediate vertex of a shortest path from *u* to *v* and we can break the problem down into the shortest path from *u* to w_k and w_k to *v*
- OR w_k is not an intermediate vertex and we can just consider the shortest path involving $\{w_1, ..., w_{k-1}\}$ Recursive Relation: Let $d_{ij}^{(k)}$ be the shortest path from v_i to v_j using only vertices $\{v_1, ..., v_k\}$. Then:

$$d_{ij}^{(k)} = \begin{cases} w(v_i, v_j) & if\ k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} & if\ k > 0 \end{cases}$$

RECURSIVE APPROACH

```
function floydwarshall(w, n) begin
    var D = 2D array (n * n)
    for i = 1 to n begin
        for j = 1 to n begin
            D[i, j] = fwr(w, n, i, j)
        end
    end
    return D
end

function fwr(w, k, i, j) begin
    if k = 0 then return w[i, j]
    else begin
        var c1 = fwr(w, k-1, i, j)
        var c2 = fwr(w, k-1, i, k) + fwr(w, k-1, k, j)
        return min(c1, c2)
    end
end
```

Time Complexity $O(n \cdot nm) = O(n^2m)$, may be as bad as n^4 if $m=n^2$

TOP-DOWN APPROACH

```
function floydwarshall(w, n) begin
    var D = 2D array (n * n)
    for i = 1 to n begin
        for j = 1 to n begin
            D[i, j] = fwr(w, n, i, j)
        end
    end
    return D
end

var m = 3D array (n*n*n) initialized to infinity
function fwr(w, k, i, j) begin
    if m[k, i, j] = infinity then begin
        if k = 0 then m[k, i, j] = w[i, j]
        else begin
            var c1 = fwr(w, k-1, i, j)
            var c2 = fwr(w, k-1, i, k) + fwr(w, k-1, k, j)
            m[k, i, j] = min(c1, c2)
        end
    end
    return m[k, i, j]
end
```

Time Complexity $O(n^3)$

BOTTOM-UP APPROACH

```
function floydwarshall(w, n) begin
    var D = 3D array (n*n*n) initialized to infinity
    for i = 1 to n begin
        for j = 1 to n begin
            D[0, i, j] = w[i, j]
        end
    end
    for k = 1 to n begin
        for i = 1 to n begin
            for j = 1 to n begin
                var c1 = D[k-1, i, j]
                var c2 = D[k-1, i, k] + D[k-1, k, j]
                D[k, i, j] = min(c1, c2)
            end
        end
    end
    return D[n]
end
```

Time Complexity $O(n^3)$

IMPLEMENTATION - 1

```
# Python Program for Floyd Warshall Algorithm
# Number of vertices in the graph
V = 4
# Define infinity as the large enough value. This value will be
# used for vertices not connected to each other
INF  = 99999

# Solves all pair shortest path via Floyd Warshall Algorithm
def floydWarshall(graph):

    """ dist[][] will be the output matrix that will finally
        have the shortest distances between every pair of vertices """
    """ initializing the solution matrix same as input graph matrix
    OR we can say that the initial values of shortest distances
    are based on shortest paths considering no
    intermediate vertices """
    dist = map(lambda i : map(lambda j : j , i) , graph)

    """ 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 the set
    {0, 1, 2, .. k-1} as intermediate vertices.
      ----> After the end of a iteration, vertex no. k is
     added to the set of intermediate vertices and the
    set becomes {0, 1, 2, .. k}
    """
```

IMPLEMENTATION - 2

```
 for k in range(V):
        # pick all vertices as source one by one
        for i in range(V):
            # Pick all vertices as destination for the
            # above picked source
            for j in range(V):
                # If vertex k is on the shortest path from
                # i to j, then update the value of dist[i][j]
                dist[i][j] = min(dist[i][j] ,
                                 dist[i][k]+ dist[k][j])
    printSolution(dist)
# A utility function to print the solution
def printSolution(dist):
    print "Following matrix shows the shortest distances\
 between every pair of vertices"
    for i in range(V):
        for j in range(V):
            if(dist[i][j] == INF):
                print "%7s" %("INF"),
            else:
                print "%7d\t" %(dist[i][j]),
            if j == V-1:
                print ""
# Driver program to test the above program
graph = [[0,5,INF,10],
             [INF,0,3,INF],
             [INF, INF, 0,   1],
             [INF, INF, INF, 0]
        ]
# Print the solution
floydWarshall(graph);
```

THANKYOU

DIVYANSHU SHRIVASTAVA

MCA 2ND YEAR

ROLL NO. – 187909

REG. NO. – MC18111