Floyd-Warshall Algorithm

https://www.programiz.com/dsa/floyd-warshall-algorithm

Floyd-Warshall Algorithm

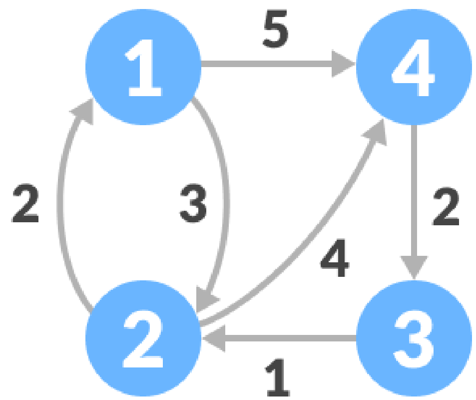
- Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph.
- This algorithm works for both the directed and undirected weighted graphs.
- But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle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

Floyd-Warshall Algorithm

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

How Floyd-Warshall Algorithm Works?

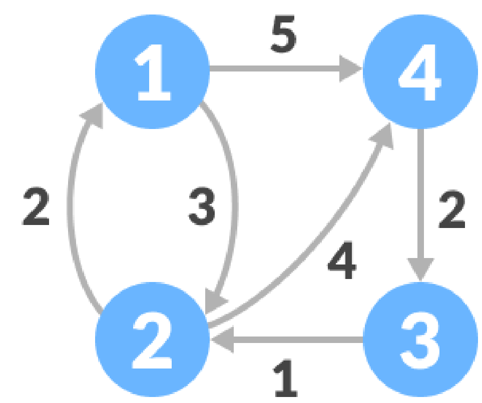
- Let the given graph be:



- Follow the steps below to find the shortest path between all the pairs of vertices.

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.



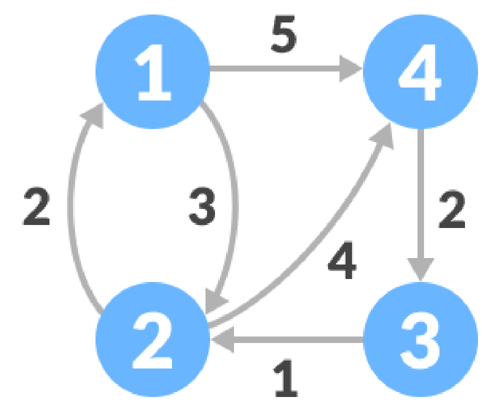
- Each cell A[i][j] is filled with the distance from the i^{th} vertex to the j^{th} vertex.
- If there is no path from i^{th} vertex to j^{th} vertex, the cell is left as infinity.

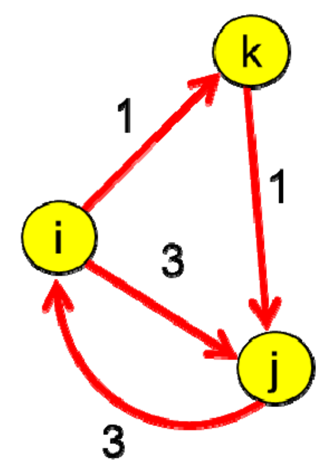
$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

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])

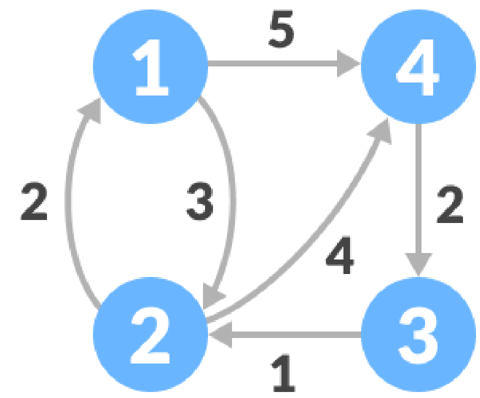
  if (A[i][j] > A[i][k] + A[k][j]).
- That is, if the direct distance from the source to the destination is greater than the path through the vertex k, then the cell is filled with A[i][k] + A[k][j].





Step 2:

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



For example:

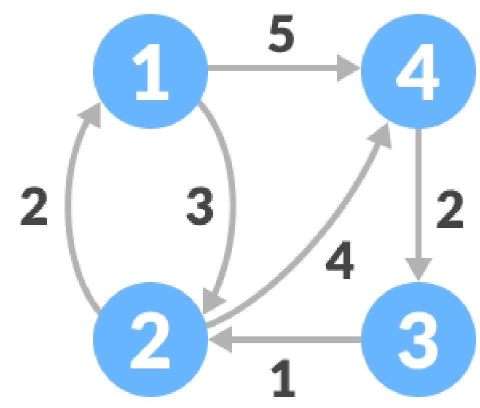
- For $A^1[2, 4]$, the direct distance from vertex 2 to 4 is 4 and the sum of the distance from vertex 2 to 4 through vertex (i.e. from vertex 2 to 1 and from vertex 1 to 4) is 7.
- Since $4 < 7$, $A^0[2, 4]$ is filled with 4.

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

Step 3:

- In a similar way, A^2 is created using A^1.
- The elements in the second column and the second row are left as they are.

- In this step, k is the second vertex (i.e. vertex 2).
- The remaining steps are the same as in step 2.

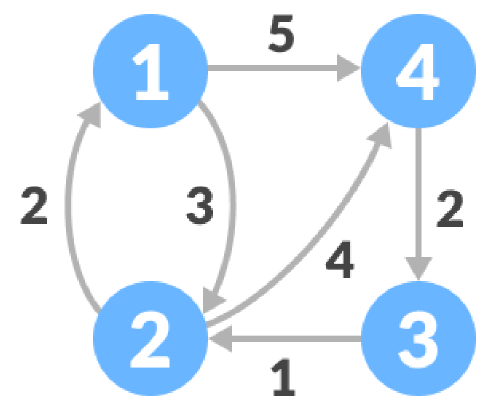


$$A^2 = \begin{array}{cc} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 3 & & \\ 2 & 0 & 9 & 4 \\ & 1 & 0 & \\ & \infty & & 0 \end{bmatrix} \end{array} \longrightarrow \begin{array}{cc} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array}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

Step 4:

- Similarly, A^3 and A^4 is also created.

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



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

A^4 gives the shortest path between each pair of vertices.

Floyd-Warshall Algorithm

```
n = no of vertices
A = matrix of dimension n*n
for k = 1 to n
   for i = 1 to n
      for j = 1 to n
         A^k[i, j] = min (A^(k-1)[i, j], A^(k-1)[i, k] + A^(k-1)[k, j])
return A
```

Floyd Warshall Algorithm Complexity

Time Complexity

- There are three loops.
- Each loop has constant complexities.
- So, the time complexity of the Floyd-Warshall algorithm is $O(n^3)$.

Space Complexity

- The space complexity of the Floyd-Warshall algorithm is $O(n^2)$.

Floyd Warshall Algorithm Applications

- To find the shortest path is a directed graph
- To find the transitive closure of directed graphs
- To find the Inversion of real matrices
- For testing whether an undirected graph is bipartite

```c
// Floyd-Warshall Algorithm in C
#include <stdio.h>
// defining the number of vertices
#define nV 4
#define INF 999
void printMatrix(int A[][nV]);

void floydWarshall(int graph[][nV]){
  int A[nV][nV], i, j, k;
  for (i = 0; i < nV; i++)
    for (j = 0; j < nV; j++)
      A[i][j] = graph[i][j];
  for (k = 0; k < nV; k++)  {
    for (i = 0; i < nV; i++)    {
      for (j = 0; j < nV; j++)      {
        if (A[i][k] + A[k][j] < A[i][j])
          A[i][j] = A[i][k] + A[k][j];
      }
    }
  }
  printMatrix(A);
}
```

```c
void printMatrix(int A[][nV]){
  for (int i = 0; i < nV; i++)  {
    for (int j = 0; j < nV; j++)    {
      if (A[i][j] == INF)
        printf("%4s", "INF");
      else
        printf("%4d", A[i][j]);
    }
    printf("\n");
  }
}

int main(){
  int graph[nV][nV] = {{0, 3, INF, 5},
           {2, 0, INF, 4},
           {INF, 1, 0, INF},
           {INF, INF, 2, 0}};
  floydWarshall(graph);
}
```