**Floyd Warshall Algorithm**

**Introduction**

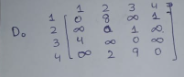
The Floyd-Warshall algorithm is a dynamic programming algorithm used to find the shortest paths between all pairs of vertices in a weighted graph. This algorithm works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative). It follows [Dynamic Programming](https://www.geeksforgeeks.org/introduction-to-dynamic-programming-data-structures-and-algorithm-tutorials/) approach to check every possible path going via every possible node in order to calculate shortest distance between every pair of nodes.

**Working with example**

1. Take a weighted positive graph

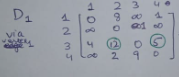


Step 1: Initialize the Distance[][] matrix using the input graph such that Distance[i][j]= weight of edge from i to j, also Distance[i][j] = Infinity if there is no edge from i to j.



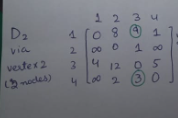
Step 2: Treat node K=1 as an intermediate node and calculate the Distance[][] for every {i,j} node pair by finding the minimum value from already computed distance in earlier steps (K-1)

= Distance[i][j] = minimum (Distance[i][j], (Distance from i to 1) + (Distance from 1 to j ))  
= Distance[i][j] = minimum (Distance[i][j], Distance[i][1] + Distance[1][j])



Step 3: Treat node K=2 as an intermediate node and calculate the Distance[][] for every {i,j} node pair by finding the minimum value from already computed distance in earlier steps (K-2)

= Distance[i][j] = minimum (Distance[i][j], (Distance from i to 2) + (Distance from 2 to j ))  
= Distance[i][j] = minimum (Distance[i][j], Distance[i][2] + Distance[2][j])



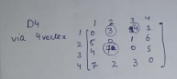
Step 4: Treat node K=3 as an intermediate node and calculate the Distance[][] for every {i,j} node pair by finding the minimum value from already computed distance in earlier steps (K-3)

= Distance[i][j] = minimum (Distance[i][j], (Distance from i to 3) + (Distance from 3 to j ))  
= Distance[i][j] = minimum (Distance[i][j], Distance[i][3] + Distance[3][j])



Step 5: Treat node K=4 as an intermediate node and calculate the Distance[][] for every {i,j} node pair by finding the minimum value from already computed distance in earlier steps (K-4)

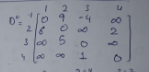
= Distance[i][j] = minimum (Distance[i][j], (Distance from i to 4) + (Distance from 4 to j ))  
= Distance[i][j] = minimum (Distance[i][j], Distance[i][4] + Distance[4][j])



1. Take a weighted positive negative graph.



Step 1: Initialize the Distance[][] matrix using the input graph such that Distance[i][j]= weight of edge from i to j, also Distance[i][j] = Infinity if there is no edge from i to j.



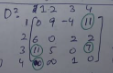
Step 2: Treat node K=1 as an intermediate node and calculate the Distance[][] for every {i,j} node pair by finding the minimum value from already computed distance in earlier steps (K-1)

= Distance[i][j] = minimum (Distance[i][j], (Distance from i to 1) + (Distance from 1 to j ))  
= Distance[i][j] = minimum (Distance[i][j], Distance[i][1] + Distance[1][j])



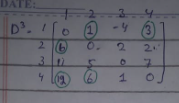
Step 3: Treat node K=2 as an intermediate node and calculate the Distance[][] for every {i,j} node pair by finding the minimum value from already computed distance in earlier steps (K-2)

= Distance[i][j] = minimum (Distance[i][j], (Distance from i to 2) + (Distance from 2 to j ))  
= Distance[i][j] = minimum (Distance[i][j], Distance[i][2] + Distance[2][j])



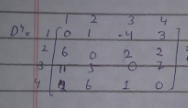
Step 4: Treat node K=3 as an intermediate node and calculate the Distance[][] for every {i,j} node pair by finding the minimum value from already computed distance in earlier steps (K-3)

= Distance[i][j] = minimum (Distance[i][j], (Distance from i to 3) + (Distance from 3 to j ))  
= Distance[i][j] = minimum (Distance[i][j], Distance[i][3] + Distance[3][j])



Step 5: Treat node K=4 as an intermediate node and calculate the Distance[][] for every {i,j} node pair by finding the minimum value from already computed distance in earlier steps (K-4)

= Distance[i][j] = minimum (Distance[i][j], (Distance from i to 4) + (Distance from 4 to j ))  
= Distance[i][j] = minimum (Distance[i][j], Distance[i][4] + Distance[4][j])



**Pseudocode:**

For(k=1 to 4) for D1 , D2 , D3 , D4

{ for(i=1 to 4)

{for(j=1 to 4)

Distance[i][j] = minimum (Distance[i][j], Distance[i][k] + Distance[k][j])

}

}

**Time Complexity**

Time complexity depends on the number of problems and sub-problems because we have to calculate shortest path of every node (n) . Suppose the size of any metrix is n\*n= n2 (n2 sub-problems) . And we also know every new distance[][] for any i and j depends on the distance calculated in earlier steps e.g., D1 depends on D0 , D2 depends on D1 , D3 depends on D4 . So, we have to work on every metrix n2 \*n e.g., n where n is total number of metrix . So the time complexity becomes O(n3).

For space complexity, we have n matrix and each matrix size is n2 . So its space complexity will be O(n3). But we can also refined this by saying that n2 is the space of matrix but rather than taking all metrixes we take the currently using metrixes , which will be reduce to 2 metrixes at a time. So the space complexity will be O(n2).

**Running running**

The **running time** of the Floyd-Warshall algorithm is O(V3) where V is the number of vertices in the graph.

**Reason for O(V3)**

1. **Outer Loop Over Intermediate Nodes**:
   * The algorithm iterates through all vertices D as potential intermediate nodes.
   * This loop runs V times.
2. **Nested Loops Over Source and Destination Nodes**:
   * For each intermediate node D, the algorithm considers every pair of source (i) and destination (j) nodes.
   * Both i and j iterate over all V vertices, resulting in V2 iterations for these nested loops.
3. **Constant Time Update for Each Pair**:
   * Inside the innermost loop, the algorithm updates the distance Distance[i][j] by comparing two values: Distance[i][j] and Distance[i][D]+Distance[D][j].
   * This operation takes O(1) time.

Thus, the total time complexity is:

**O (V) × O (V2) × O (1) = O (V3)**

**Comparison with other overlapping Algorithm:**

Here’s a comparison of algorithms that are comparable or overlapping with the **Floyd-Warshall Algorithm** for solving shortest path problems in a graph. This includes **Dijkstra’s Algorithm**, **Bellman-Ford Algorithm**, and others like **Johnson’s Algorithm**

**1. Floyd-Warshall Algorithm**

* **Purpose**: Computes the shortest paths between all pairs of vertices in a weighted graph.
* **Approach**: Dynamic Programming.
* **Key Features**:
  + Handles both positive and negative edge weights (but no negative weight cycles).
  + Runs in O(V3) time.
  + Ideal for dense graphs (many edges).

**2. Dijkstra's Algorithm**

* **Purpose**: Finds the shortest path from a single source vertex to all other vertices.
* **Approach**: Greedy Algorithm.
* **Key Features**:
  + Works only with non-negative edge weights.
  + Runs in O (V2) for an adjacency matrix or O ((V+E) ⋅logV) with a priority queue.
  + More efficient for sparse graphs (few edges).
* **Comparison with Floyd-Warshall**:
  + Dijkstra is for **single-source shortest path**, while Floyd-Warshall is for **all-pairs shortest path**.
  + Dijkstra is faster for single-source queries in large sparse graphs.

**3. Bellman-Ford Algorithm**

* **Purpose**: Computes the shortest paths from a single source to all other vertices, even with negative edge weights.
* **Approach**: Dynamic Programming.
* **Key Features**:
  + Handles negative edge weights and detects negative weight cycles.
  + Runs in O(V⋅E)
* **Comparison with Floyd-Warshall**:
  + Bellman-Ford is designed for **single-source shortest path**, while Floyd-Warshall handles **all pairs**.
  + Floyd-Warshall has a fixed O(V3) time complexity, whereas Bellman-Ford scales based on the number of edges (EEE).

**4. Johnson’s Algorithm**

* **Purpose**: Computes the shortest paths between all pairs of vertices in a sparse graph.
* **Approach**: Combines Bellman-Ford and Dijkstra.
* **Key Features**:
  + Handles negative edge weights.
  + Uses reweighting to eliminate negative weights and then applies Dijkstra for all-pairs shortest paths.
  + Runs in O(V2⋅logV+V⋅E), which is more efficient than Floyd-Warshall for sparse graphs.
* **Comparison with Floyd-Warshall**:
  + Johnson’s Algorithm is faster for **sparse graphs**.
  + Floyd-Warshall is simpler and more straightforward for **dense graphs**.

**Comparison Table**

| **Algorithm** | **Purpose** | **Handles Negative Weights?** | **Time Complexity** | **Best for** |
| --- | --- | --- | --- | --- |
| **Floyd-Warshall** | All-pairs shortest paths | Yes | O(V3) | Dense graphs |
| **Dijkstra** | Single-source shortest path | No | O(V2), or O((V+E)logV)) with a priority queue | Sparse graphs (non-negative weights) |
| **Bellman-Ford** | Single-source shortest path | Yes | O(V⋅E) | Graphs with negative weights |
| **Johnson’s Algorithm** | All-pairs shortest paths | Yes | O(V2⋅logV+V⋅E) | Sparse graphs with negative weights |

**Real time Applications of floyd Warshall Algorithm**

The Floyd-Warshall algorithm can calculate the shortest paths for any pairings of vertices, which makes it useful in a variety of real-world situations. Here are a few notable use cases:

### 1. Network routing and navigation systems

By taking into account all potential courses and figuring out which ones are the shortest, the algorithm helps contemporary navigation systems choose the best routes for automobiles or pedestrians. In complicated road networks, it helps in directional optimization.

### 2. Urban transportation planning

City planners use the algorithm to examine traffic patterns and create effective public transportation systems. They can choose the best station locations and routes by finding the quickest paths to all destinations.

### 3. Game development

The Floyd-Warshall algorithm is used by game designers to construct realistic pathfinding for characters and objects in video games. It guarantees that characters may easily move across different environments, dodging hazards and getting where they need to go.

### 4. Traffic optimization

The algorithm is used by traffic engineers to improve traffic flow and lessen congestion. The quickest routes between crossings or junctions help planners create plans for managing traffic lights and the entire road system.

### 5. Telecommunication network management

The Floyd-Warshall algorithm helps telecommunication networks control data routing. It helps identify the most effective data transmission routes, promoting dependable and quick communication.

Thus, the adaptability of the Floyd-Warshall algorithm in these above sectors highlights its practical significance and applicability in resolving challenging optimization and routing issues.

**Code implantation for finding the shortest path from different locations of city.**

INF = float('inf')

vertices = 4  # Number of cities

def floyd\_warshall(graph):

    # Initialize the distance matrix

    distance\_between\_vertices = [[INF] \* vertices for \_ in range(vertices)]

    # Copy the input graph to the distance matrix

    for i in range(vertices):

        for j in range(vertices):

            distance\_between\_vertices[i][j] = graph[i][j]

    # Set the distance to self as 0

    for i in range(vertices):

        distance\_between\_vertices[i][i] = 0

    # Main algorithm

    for k in range(vertices):

        for i in range(vertices):

            for j in range(vertices):

                distance\_between\_vertices[i][j] = min(

                    distance\_between\_vertices[i][j],

                    distance\_between\_vertices[i][k] +

distance\_between\_vertices[k][j]

                )

    print\_shortest\_paths(distance\_between\_vertices)

def print\_shortest\_paths(distance):

    print("Following shows the shortest path between all the pairs of cities:")

    for i in range(vertices):

        for j in range(vertices):

            print(f"The shortest distance between city {i + 1} and city {j + 1} is", end=" ")

            if distance[i][j] == INF:

                print("INF")

            else:

                print(distance[i][j])

        print(" ")

# Example graph representing distances between cities

# Here, the graph is represented as an adjacency matrix

graph = [

    [0, 3, INF, 5],  # Distances from city 1

    [2, 0, INF, 4],  # Distances from city 2

    [INF, 1, 0, INF], # Distances from city 3

    [INF, INF, 2, 0]  # Distances from city 4

]

# Run the Floyd-Warshall algorithm

floyd\_warshall(graph)

### ****Step-by-Step Breakdown****

#### **1. Initialization**

INF = float('inf')

vertices = 4 # Number of cities

* **INF**: Represents infinity, used for vertices that are not directly connected.
* **vertices**: Total number of cities (or nodes) in the graph.

#### **2. Function Definition**

def floyd\_warshall(graph):

* **Purpose**: Implements the Floyd-Warshall algorithm to compute the shortest path between all pairs of vertices.
* **Input**: graph – a 2D list (adjacency matrix) representing the distances between cities.

#### **3. Initialize the Distance Matrix**

distance\_between\_vertices = [[INF] \* vertices for \_ in range(vertices)]

* Creates a 2D list initialized with INF values for all vertex pairs.
* This will store the shortest distances between all pairs of vertices.

#### **4. Copy Input Graph**

for i in range(vertices):

for j in range(vertices):

distance\_between\_vertices[i][j] = graph[i][j]

* Copies the input adjacency matrix (graph) to the distance matrix (distance\_between\_vertices) to allow updates.

#### **5. Set Distance to Self**

for i in range(vertices):

distance\_between\_vertices[i][i] = 0

* Sets the distance from a vertex to itself (distance[i][i]) as 0.

#### **6. Main Algorithm**

for k in range(vertices):

for i in range(vertices):

for j in range(vertices):

distance\_between\_vertices[i][j] = min(

distance\_between\_vertices[i][j],

distance\_between\_vertices[i][k] + distance\_between\_vertices[k][j]

)

* **Outer Loop (k)**: Iterates over all vertices as intermediate nodes.
* **Middle Loop (i)**: Iterates over all starting vertices.
* **Inner Loop (j)**: Iterates over all ending vertices.
* **Update Rule**: Updates the shortest distance from i to j:
  + distance[i][j]: Current shortest path between i and j.
  + distance[i][k] + distance[k][j]: Path through intermediate vertex k.

#### **7. Print Results**

print\_shortest\_paths(distance\_between\_vertices)

* Calls a helper function to print the shortest paths between all cities.

#### **8. Helper Function: Print Shortest Paths**

def print\_shortest\_paths(distance):

print("Following shows the shortest path between all the pairs of cities:")

for i in range(vertices):

for j in range(vertices):

print(f"The shortest distance between city {i + 1} and city {j + 1} is", end=" ")

if distance[i][j] == INF:

print("INF")

else:

print(distance[i][j])

print(" ")

* **Input**: distance – the distance matrix.
* **Output**: Prints the shortest distances between all pairs of cities in a readable format.
* **Logic**:
  + If distance[i][j] == INF, there is no path between the cities.
  + Otherwise, it prints the shortest distance.

### ****Example Graph****

graph = [

[0, 3, INF, 5], # Distances from city 1

[2, 0, INF, 4], # Distances from city 2

[INF, 1, 0, INF], # Distances from city 3

[INF, INF, 2, 0] # Distances from city 4

]

* The graph is represented as an adjacency matrix:
  + graph[i][j]: Distance from city i to city j.
  + INF: No direct path between two cities.

### ****Program Execution****

1. The function floyd\_warshall(graph) computes the shortest distances for all pairs of cities.
2. The helper function print\_shortest\_paths(distance) prints the results.

### ****Expected Output****

For the given example graph, the program will output:

Output:

Following shows the shortest path between all the pairs of cities:

The shortest distance between city 1 and city 1 is 0

The shortest distance between city 1 and city 2 is 3

The shortest distance between city 1 and city 3 is 7

The shortest distance between city 1 and city 4 is 5

The shortest distance between city 2 and city 1 is 2

The shortest distance between city 2 and city 2 is 0

The shortest distance between city 2 and city 3 is 6

The shortest distance between city 2 and city 4 is 4

The shortest distance between city 3 and city 1 is INF

The shortest distance between city 3 and city 2 is 1

The shortest distance between city 3 and city 3 is 0

The shortest distance between city 3 and city 4 is 5

The shortest distance between city 4 and city 1 is INF

The shortest distance between city 4 and city 2 is INF

The shortest distance between city 4 and city 3 is 2

The shortest distance between city 4 and city 4 is 0