


Experiment No. : 5

Title: Floyd-Warshall Algorithm using Dynamic programming approach



Batch:SY-IT(B3)

Roll No.: 16010423076

Experiment No.: 5

Aim: To Implement All pair shortest path Floyd-Warshall Algorithm using Dynamic programming approach and analyse its time Complexity.

Algorithm of Floyd-Warshall Algorithm:

```

FLOYD-WARSHALL( $W$ )
1   $n = W.rows$ 
2   $D^{(0)} = W$ 
3  for  $k = 1$  to  $n$ 
4      let  $D^{(k)} = (d_{ij}^{(k)})$  be a new  $n \times n$  matrix
5      for  $i = 1$  to  $n$ 
6          for  $j = 1$  to  $n$ 
7               $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ 
8  return  $D^{(n)}$ 

```

Constructing Shortest Path:



We can give a recursive formulation of $\pi_{ij}^{(k)}$. When $k = 0$, a shortest path from i to j has no intermediate vertices at all. Thus,

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty, \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty. \end{cases} \quad (25.6)$$

For $k \geq 1$, if we take the path $i \rightsquigarrow k \rightsquigarrow j$, where $k \neq j$, then the predecessor of j we choose is the same as the predecessor of j we chose on a shortest path from k with all intermediate vertices in the set $\{1, 2, \dots, k-1\}$. Otherwise, we

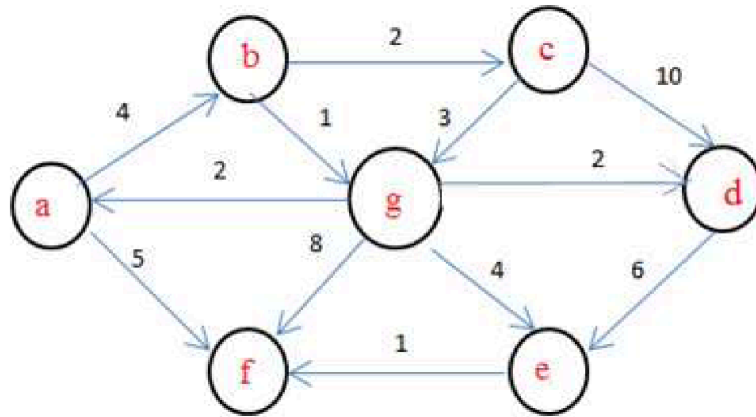
choose the same predecessor of j that we chose on a shortest path from i with all intermediate vertices in the set $\{1, 2, \dots, k-1\}$. Formally, for $k \geq 1$,

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases} \quad (25.7)$$

Working of Floyd-Warshall Algorithm:

Problem Statement

Find Shortest Path for each source to all destinations using Floyd-Warshall Algorithm for the following graph



Solution

Step 1: Represent the Graph as an Adjacency Matrix

The given graph has 7 nodes: a, b, c, d, e, f, g. We represent it using an adjacency matrix, where:

- If there is an edge between two nodes, the corresponding cell contains the weight of the edge.
- If there is no direct edge, the cell contains ∞ (infinity).
- The diagonal elements (distance from a node to itself) are 0.

	a	b	c	d	e	f	g
a	0	4	∞	∞	∞	5	2
b	∞	0	2	∞	∞	∞	1
c	∞	∞	0	10	∞	∞	3
d	∞	∞	∞	0	∞	∞	2
e	∞	∞	∞	6	0	∞	1
f	∞	∞	∞	∞	4	0	8
g	∞	∞	∞	2	1	∞	0

Step 2: Floyd-Warshall Algorithm

The algorithm iterates over all possible intermediate vertices and updates the shortest paths accordingly.

Algorithm Steps:

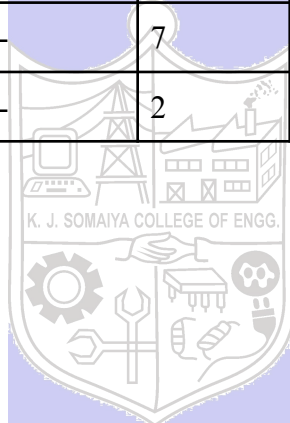
1. Initialize the distance matrix with the adjacency matrix.
2. Iterate through all nodes (k) as an intermediate node.
 - For each pair of nodes (i, j), check if the path $i \rightarrow k \rightarrow j$ is shorter than the direct path $i \rightarrow j$.

- If yes, update $\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j])$.
- 3. Repeat this process for all nodes.

Step 3: Final Shortest Path Matrix (All-Pairs)

The matrix below represents the shortest distances between each pair of nodes:

	a	b	c	d	e	f	g
a	0	4	6	4	3	5	2
b	-	0	2	3	2	-	1
c	-	-	0	5	4	-	3
d	-	-	-	0	3	-	2
e	-	-	-	3	0	-	1
f	-	-	-	7	4	0	5
g	-	-	-	2	1	-	0



Derivation of Floyd-Warshall Algorithm:

Time complexity Analysis

Best Case: $O(V^3)$

- In the best-case scenario, the algorithm completes all iterations of the nested loops without any relaxation steps needed.
- This implies that the shortest paths between all pairs of vertices are already determined and no updates are necessary.
- Because of this, the three nested loops each of which iterates through every vertex—are the only factors influencing the algorithm's time complexity.
- With V vertices, each loop iterates V times, resulting in a time complexity of $O(V^3)$ for the best case.

Average Case: $O(V^3)$

- The average-case time complexity of the Floyd-Warshall algorithm is also $O(V^3)$.
- This complexity holds true across various graph structures and densities, as the algorithm's performance primarily depends on the number of vertices and the number

of iterations needed to compute shortest paths between all pairs of vertices.

- The algorithm's inherent nature of iterating through all pairs of vertices and updating distances contributes to the cubic time complexity, regardless of the specific graph characteristics.

Worst Case: $O(V^3)$

- Conversely, in the worst-case scenario, the algorithm performs relaxation steps for each pair of vertices during all iterations of the nested loops.
- This means that the algorithm needs to update distances between vertices multiple times until the shortest paths are determined.
- Again, the time complexity is determined by the three nested loops, each iterating through all vertices.
- With V vertices, each loop iterates V times, resulting in a time complexity of $O(V^3)$ for the worst case as well.

Program(s) of Floyd-Warshall Algorithm:

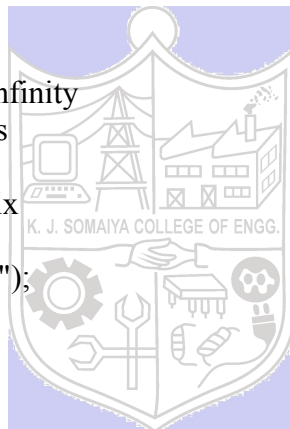
```
#include <stdio.h>
```

```
#define INF 99999 // Representing Infinity
```

```
#define V 4 // Number of vertices
```

```
// Function to print the solution matrix
```

```
void printSolution(int dist[][V]) {
    printf("Shortest distance matrix:\n");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (dist[i][j] == INF)
                printf("INF\t");
            else
                printf("%d\t", dist[i][j]);
        }
        printf("\n");
    }
}
```



```
// Floyd-Warshall Algorithm
```

```
void floydWarshall(int graph[][V]) {
    int dist[V][V], i, j, k;
```

```
    // Copying the input graph to the distance matrix
```

```
    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            dist[i][j] = graph[i][j];
```

```
    // Updating distances using Floyd-Warshall Algorithm
```

```
    for (k = 0; k < V; k++) {
```

```

for (i = 0; i < V; i++) {
    for (j = 0; j < V; j++) {
        if (dist[i][k] != INF && dist[k][j] != INF && dist[i][k] + dist[k][j] < dist[i][j]) {
            dist[i][j] = dist[i][k] + dist[k][j];
        }
    }
}

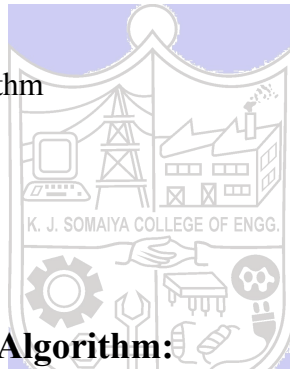
// Printing the shortest path matrix
printSolution(dist);
}

// Main function
int main() {
    int graph[V][V] = {
        {0, 3, INF, 5},
        {2, 0, INF, 4},
        {INF, 1, 0, INF},
        {INF, INF, 2, 0}
    };

    // Running Floyd-Warshall Algorithm
    floydWarshall(graph);

    return 0;
}

```



Output(o) of Floyd-Warshall Algorithm:

```

Output
Shortest distance matrix:
0   3   7   5
2   0   6   4
3   1   0   5
5   3   2   0

|
=== Code Execution Successful ===

```

Post Lab Questions:-

Explain the dynamic programming approach for the Floyd-Warshall algorithm and write the various applications of it.

- The Floyd-Warshall algorithm is a dynamic programming approach used to find the shortest paths between all pairs of nodes in a weighted graph. It works by breaking down the problem into smaller subproblems and using previously computed results to build the final solution.
- The algorithm maintains a distance matrix ($\text{dist}[i][j]$), where each element represents the shortest distance from vertex i to vertex j . Initially, this matrix is set to direct edge weights, with INF for no direct edge.
- At each step, the algorithm considers an intermediate vertex k and checks if the path $i \rightarrow k \rightarrow j$ is shorter than the current path $i \rightarrow j$.
- If so, it updates $\text{dist}[i][j]$. This process repeats for all pairs (i, j) , considering each vertex k as an intermediate step one by one.
- The time complexity of the Floyd-Warshall algorithm is $O(V^3)$, where V is the number of vertices, making it suitable for small to medium-sized graphs.

Applications of the Floyd-Warshall Algorithm

1. **Shortest Path Calculation:**
It is widely used to compute the shortest paths between all pairs of nodes in transportation and communication networks.
2. **Network Routing:**
Used in computer networks and routing protocols to find the best paths for data transfer.
3. **Flight and Road Network Planning:**
Helps in determining the shortest routes between cities in air and road transport systems.
4. **Graph Analysis in Social Networks:**
Used to measure distances between users or nodes in social network graphs.
5. **Game Development:**
Used in pathfinding and AI decision-making in games where entities need to find optimal movement paths.
6. **Dependency Analysis in Compilers:**
Helps in optimizing code execution by analyzing dependencies between different components of a program.
7. **Telecommunication Systems:**
Used for optimizing call routing and network connectivity in telecom systems.

Conclusion: (Based on the observations):

From this experiment, I learned how the Floyd-Warshall algorithm efficiently finds the shortest paths between all pairs of nodes using a dynamic programming approach. By implementing and analyzing the algorithm, I understood how it updates the distance matrix step by step and how its time complexity remains $O(V^3)$ in all cases. I also explored its real-world applications in network routing, transportation, social networks, and game development. This experiment helped me strengthen my understanding of advanced algorithms and their practical use in solving shortest path problems.

Outcome: CO3: Implement advanced programming algorithms with their applications.

References:

1. Richard E. Neapolitan, " Foundation of Algorithms ", 5th Edition 2016, Jones & Bartlett Students Edition
2. Harsh Bhasin , " Algorithms : Design & Analysis", 1st Edition 2013, Oxford Higher education, India
3. T.H. Cormen ,C.E. Leiserson,R.L. Rivest, and C. Stein, " Introduction to algorithms", 3rd Edition 2009, Prentice Hall India Publication
4. Jon Kleinberg, Eva Tardos, " Algorithm Design", 10th Edition 2013, Pearson India Education Services Pvt. Ltd.

