Experiment No. : 6

Title: Floyd-Warshall Algorithm using Dynamic programming approach

Batch: A2 **Roll No.: 16010421059** **Experiment No.: 6**

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)

$n = W.rows$

$D^{(0)} = W$

for $k = 1$ **to** n

  let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ be a new $n \times n$ matrix

  for $i = 1$ **to** n

    for $j = 1$ **to** n

      $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$

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,

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

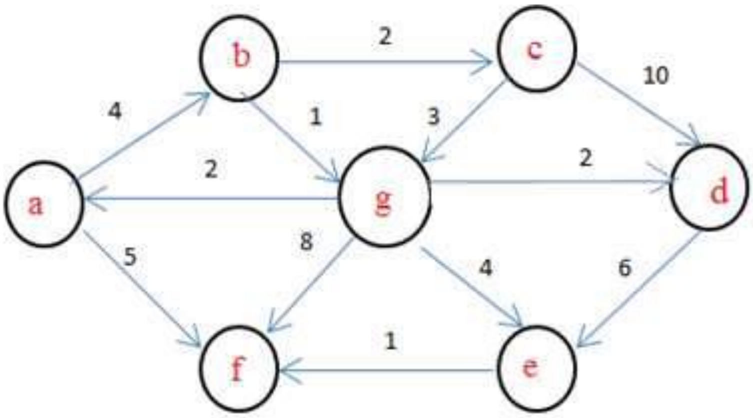
For $k \geq 1$, if we take the path $i \rightsquigarrow k \rightsquigarrow j$, where $k \neq j$, then the predecessor of j we choose is the same as the predecessor of j we chose on a shortest path from k with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. Otherwise, we choose the same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

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

Working of Floyd-Warshall Algorithm:

Problem Statement

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



Derivation of Floyd-Warshall Algorithm:

Time complexity Analysis:

The time complexity of the Floyd-Warshall algorithm is O(n^3), where n is the number of vertices in the graph. This is because the algorithm involves computing the shortest path between all pairs of vertices in the graph, and there are n^2 pairs of vertices to consider. For each pair of vertices, the algorithm considers all possible intermediate vertices, which takes O(n) time. Therefore, the total time complexity of the algorithm is O(n^3).
The space complexity of the algorithm is also O(n^2), as it requires a two-dimensional array to store the distance between each pair of vertices.
Despite its relatively high time complexity, the Floyd-Warshall algorithm is still a practical choice for small to medium-sized graphs. However, for very large graphs, the time complexity can become prohibitive, and other algorithms such as Dijkstra's algorithm or A* algorithm may be more suitable.

Program(s) of Floyd-Warshall Algorithm:

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

void floydWarshall(int **graph, int n)
{
   int i, j, k;
   for (k = 0; k < n; k++)
   {
      for (i = 0; i < n; i++)
      {
         for (j = 0; j < n; j++)
         {
```

```
                if (graph[i][j] > graph[i][k] + graph[k][j])
                    graph[i][j] = graph[i][k] + graph[k][j];
            }
        }
    }
}

int main(void)
{
    int n, i, j;
    printf("Enter the number of vertices: ");
    scanf("%d", &n);
    int **graph = (int **)malloc((long unsigned) n * sizeof(int *));
    for (i = 0; i < n; i++)
    {
        graph[i] = (int *)malloc((long unsigned) n * sizeof(int));
    }
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            if (i == j)
                graph[i][j] = 0;
            else
                graph[i][j] = 100;
        }
    }
    printf("Enter the edges: \n");
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            printf("[%d][%d]: ", i, j);
            scanf("%d", &graph[i][j]);
        }
    }
    printf("The original graph is:\n");
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            printf("%d ", graph[i][j]);
        }
        printf("\n");
    }
    floydWarshall(graph, n);
    printf("The shortest path matrix is:\n");
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
```

```
        {
            printf("%d ", graph[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Output(o) of Floyd-Warshall Algorithm:

```
Enter the number of vertices: 5
Enter the edges:
[0][0]: 12
[0][1]: 23
[0][2]: 43
[0][3]: 54
[0][4]: 1
[1][0]: 21
[1][1]: 6
[1][2]: 0
[1][3]: 43
[1][4]: 4
[2][0]: 5
[2][1]: 62
[2][2]: 3
[2][3]: 34
[2][4]: 76
[3][0]: 98
[3][1]: 1
[3][2]: 34
[3][3]: 0
[3][4]: 0
[4][0]: 7
[4][1]: 43
[4][2]: 6
[4][3]: 54
[4][4]: 3
```

```
The original graph is:
12 23 43 54 1
21 6 0 43 4
5 62 3 34 76
98 1 34 0 0
7 43 6 54 3
The shortest path matrix is:
8 23 7 41 1
5 6 0 34 4
5 28 3 34 6
6 1 1 0 0
7 30 6 40 3
```

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

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.
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.
3. Similarly, 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**.
4. Similarly, A^3 and A^4 is also created.
5. A^4 gives the shortest path between each pair of vertices.
6. There are many applications of the Floyd Warshall algorithm. Some of the most popular applications are finding the shortest path between two vertices in a graph, detecting negative cycles in a graph, and computing the transitive closure of a graph.
   The Floyd Warshall algorithm can also be used for other purposes such as solving the all-pairs shortest path problem in weighted graphs, finding the closest pairs of vertices in a graph, and computing the diameter of a graph.

Conclusion: (Based on the observations):
Successfully implemented Floyd-Warshall Algorithm using Dynamic programming approach.

Outcome:

CO2: Implement Greedy and Dynamic Programming algorithms

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. Coreman ,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.