Project 2

Analyzing the Floyd - Warshall algorithm

CSE 5211: Analysis of Algorithms

Dr. William Shoaff

Zubin Kadva

Table of Contents

1. Problem Description

The main purpose of this project is to analyse the Floyd-Warshall algorithm used to find the shortest path in a weighted graph with positive or negative edges, but no negative cycles. A shortest path problem is the problem of finding the path between two nodes in a graph such that the sum of the weights of its edges is minimized.

2. Known algorithms

The most important algorithms for solving this problem are:

- Dijkstra's algorithm
- Bellman – Ford algorithm
- A* search algorithm
- Floyd – Warshall algorithm
- Johnson's algorithm
- Viterbi algorithm

3. The shortest path problem

Two vertices are adjacent when they are both incident to a common edge. A path in an undirected graph is a sequence of vertices

$$P = (v_1, v_2, \dots, v_n) \in V \times V \times \dots \times V$$

such that v_i is adjacent to v_{i+1} for$1 \leq i < n$. Such a path P is called a path of length $n - 1$ from v_1 to v_n.

Let $e_{i,j}$ be the edge incident to both v_i and v_j. Given a real-valued weight function $: E \to \mathbb{R}$, and an undirected (simple) graph G, the shortest path from v to v' is the path $P = (v_1, v_2, \dots, v_n)$ (where $v_1 = v$ and $v_n = v'$ that over all possible n minimizes the sum

$$\sum_{i=1}^{n-1} f(e_{i,i+1}).$$

When each edge in the graph has unit weight or $f: E \to \{1\}$, this is equivalent to finding the path with fewest edges. [1]

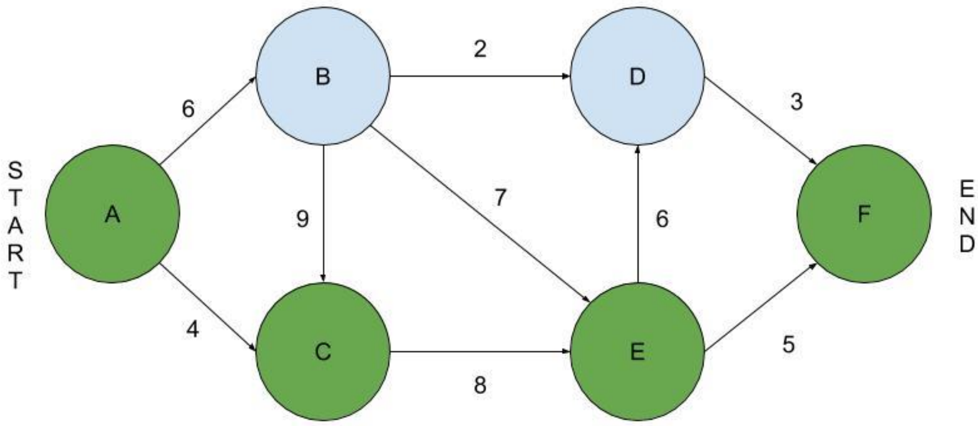


Figure: Example of shortest path in a graph

In the above figure, the shortest path is given by $P = (A, C, E, F)$ with a total cost of $4 + 8 + 5 = 17$

4. A historical perspective

The Floyd-Warshall algorithm is an example of a dynamic programming problem. It was first published by **Robert Floyd** in 1962. However, the working of this algorithm is similar to the algorithms previously published by Bernard Roy in 1959 and **Stephen Warshall** in 1962 for finding the transitive closure of a graph. The current formulation of the algorithm as three nested for loops was first described by Peter Ingerman in 1962. Hence, the algorithm is also known as the **Floyds algorithm**, the **Roy-Warshall algorithm** or the **Roy-Floyd algorithm.**

5. Algorithm description

The Floyd-Warshall algorithm considers the intermediate vertices of a shortest path, where an *intermediate vertex* of a simple path $P = (v_1, v_2, \dots, v_n)$ is any vertex of P other than v_1 or v_n that is, any vertex in the set $P = (v_2, v_3, \dots, v_{n-1})$

The algorithm relies on the observation that the vertices of G are $V = (1, 2, \dots, n)$, consider a subset $V = (1, 2, \dots, k)$ of vertices for some k. For any pair of vertices $i, j \in V$, consider all paths from i to j whose intermediate vertices are all drawn from $V = (1, 2, \dots, k)$, and let p be a minimum-weight path from among them.

The algorithm exploits a relationship between path p and shortest paths from i to j with all intermediate vertices in the set $V = (1, 2, .., k-1)$. The relationship depends on whether or not k is an intermediate vertex of path p.

This gives rise to the following cases:

1. *If k is not an intermediate vertex of path* p, then all intermediate vertices of path p are in the set $V = (1, 2, \dots, k-1)$. Thus, a shortest path from vertex i to vertex j with all intermediate vertices in the set $V = (1, 2, \dots, k-1)$ is also a shortest path from i to j with all intermediate vertices in the set $V = (1, 2, \dots, k-1)$.

2. *If k is an intermediate vertex of path* p, then we decompose p into i, k *and* j. p_1 is a shortest path from i to k with all intermediate vertices in the set $V = (1, 2, \dots, k-1)$. Similarly, p_2 is a shortest path from vertex k to vertex j with all intermediate vertices in the set $V = (1, 2, \dots, k-1)$.

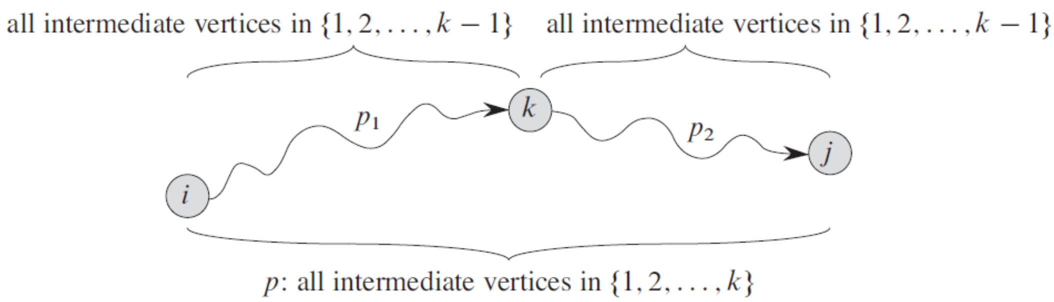


Figure: Path p is a shortest path from vertex i to vertex j, and k is the highest-numbered intermediate vertex of p [2]

5.1. A recursive solution

Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j for which all *intermediate vertices* are in the set $V = (1, 2, \dots, k)$. When $k = 0$, a path from vertex i to vertex j with no intermediate vertex numbered higher than 0 has no intermediate vertices at all. Such a path has at most one edge, and hence $d_{ij}^{(0)} = w_{ij}$. [2]

This can be represented as:

$$d_{ij}^{(k)} = \begin{cases} w_{ij}, & if\ k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)}, d_{kj}^{(k-1)}), & if k \geq 1 \end{cases}$$

$\boldsymbol{FLOYD-WARSHALL}(W)$

$n = W.rows$

$D^{(0)} = W$

$\boldsymbol{for}\ k = 1\ \boldsymbol{to}\ n$

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

  $\boldsymbol{for}\ i = 1\ \boldsymbol{to}\ n$

    $\boldsymbol{for}\ j = 1\ \boldsymbol{to}\ n$

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

$\boldsymbol{return}\ D^{(n)}$

Figure: The Floyd-Warshall algorithmic steps

```
function Floyd-Warshall(V)
    dist[v][v] = ∞
    for each vertex v
        dist[v][v] := 0
    for each edge (u,v)
        dist[u][v] := w(u,v)
    for k = 1 to V
        for i = 1 to V
            for j = 1 to V
                if dist[i][j] > dist[i][k] + dist[k][j]
                    dist[i][j] := dist[i][k] + dist[k][j]
                end if
```

Figure: The Floyd-Warshall algorithm pseudocode

5.2. Matrix representation

Given the following graph:

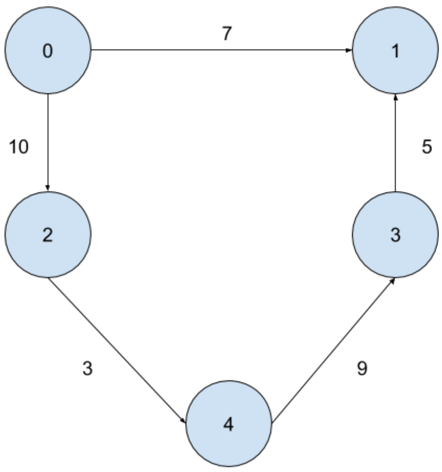


Figure: Example directed graph

Suppose the above graph is denoted by G. Then, the corresponding matrix is given as:

$$
\begin{array}{cc}
 & \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\
\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} &
\begin{vmatrix}
0 & 7 & 10 & \infty & \infty \\
\infty & 0 & \infty & \infty & \infty \\
\infty & \infty & 0 & \infty & 3 \\
\infty & 5 & \infty & 0 & \infty \\
\infty & \infty & \infty & 9 & 0
\end{vmatrix}
\end{array}
$$

Programmatically, this can be represented as:

```
g[][] = {  {0   , 7   , 10 , INF, INF},
           {INF, 0   , INF, INF, INF},
           {INF, INF, 0   , INF, 3   },
           {INF, 5   , INF, 0   , INF},
           {INF, INF, INF, 9   , 0   }   }
```

6. Implementation

```
/* Author: Zubin Kadva
 * Class: Analysis of Algorithms, Spring 2017
 * Project: Floyd-Warshall algorithm
 */

public class FloydWarshall {
    final static int INF = 999;
    static void shortestPath(int graph[][]) {
        int v = graph.length;
        int dist[][] = new int[v][v];
        // Matrix initialization is done here
        for (int i = 0; i < v; i++) {
            System.arraycopy(graph[i], 0, dist[i], 0, v);
        }
        for (int k = 0; k < v; k++) {
            // Start from a vertex as source
            for (int i = 0; i < v; i++) {
     // End at a vertex at the destination staring from the source
                for (int j = 0; j < v; j++) {
    // If vertex k is on the shortest path from i to j, then update
the value of dist[i][j]
                    if (dist[i][k] + dist[k][j] < dist[i][j]) {
                        dist[i][j] = dist[i][k] + dist[k][j];
                    }
                }
            }
        }
        // Print the shortest distance matrix
        print(dist);
    }
```

```
    static void print(int dist[][]) {
        int v = dist.length;
        for (int i = 0; i < v; i++) {
            for (int j = 0; j < v; j++) {
                if (dist[i][j] == INF) {
                    System.out.print("INF\t");
                } else {
                    System.out.print(dist[i][j] + "\t");
                }
            }
        }
    }
}
```

7. Generating random data

```
/* Author: Zubin Kadva
 * Class: Analysis of Algorithms, Spring 2017
 * Project: Floyd-Warshall algorithm
 */

static int[][] generate() {
        final int DIAGONAL = 9999;
        int V = 225;
        int[][] graph = new int[V][V];

        // Initalize diagonals first
        for (int i = 0; i < V; i++) {
            graph[i][i] = DIAGONAL;
        }
```

```
        // Pick a random distance from 1 to 10
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (i == j) {
                    continue;
                }
                graph[i][j] = new Random().nextInt(10);
            }
        }

        // Make sure there are INF distances also
        for (int i = 0; i < V; i++) {
            int a = new Random().nextInt(V);
            int b = new Random().nextInt(V);
            if (graph[a][b] != DIAGONAL) {
                graph[a][b] = INF;
            }
        }

        // Reset diagonals to 0
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (graph[i][j] == DIAGONAL) {
                    graph[i][j] = 0;
                }
            }
        }
        return graph;
    }

shortestPath(generate());
```

8. Analysis

Time Complexity

The `print` subroutine is expressed as:

$$T(n) = \sum_{i=0}^{v-1}\sum_{j=0}^{v-1}(1)$$

$$\therefore\ T(n) = \sum_{i=0}^{v-1} v$$

$$\therefore\ T(n) = v * v$$

$$\therefore\ T(n) = v^2$$

$$\therefore\ T(n) = O(v^2)$$

Thus, complexity is $\boldsymbol{O}\ (\boldsymbol{v^2})$**.**

The `shortestPath` routine is expressed as:

$$T(n) = \sum_{i=0}^{v-1}(1) + \sum_{k=0}^{v-1}\sum_{i=0}^{v-1}\sum_{j=0}^{v-1}(1)$$

$$\therefore\ T(n) = v + \sum_{k=0}^{v-1}\sum_{i=0}^{v-1} v$$

$$\therefore\ T(n) = v + \sum_{k=0}^{v-1} v * v$$

$$\therefore\ T(n) = v + v * v * v$$

$$\therefore\ T(n) = v + v^3$$

$$\therefore\ T(n) = O(v^3)$$

Thus, the worst case complexity is $\boldsymbol{O}\ (\boldsymbol{v^3})$**.**

The best, average case complexity of the algorithm is also v^3 and follows a similar proof. Hence, it follows that the **best case complexity** is $\boldsymbol{\Omega}\ (\boldsymbol{v^3})$ and the **average case complexity** is $\boldsymbol{\theta}\ (\boldsymbol{v^3})$**.**

Space complexity

We store the distances in a 2-D array with v rows and v columns. Therefore, the space complexity is given by

$$v * v = v^2$$

Thus, the complexity is $\boldsymbol{O}\ (\boldsymbol{v^2})$.

In summary,

Performance	**Best**	$\boldsymbol{\Omega}\ (\boldsymbol{v^3})$
	Average	$\theta\ (v^3)$
	Worst	$O\ (v^3)$
Space		$O\ (v^2)$

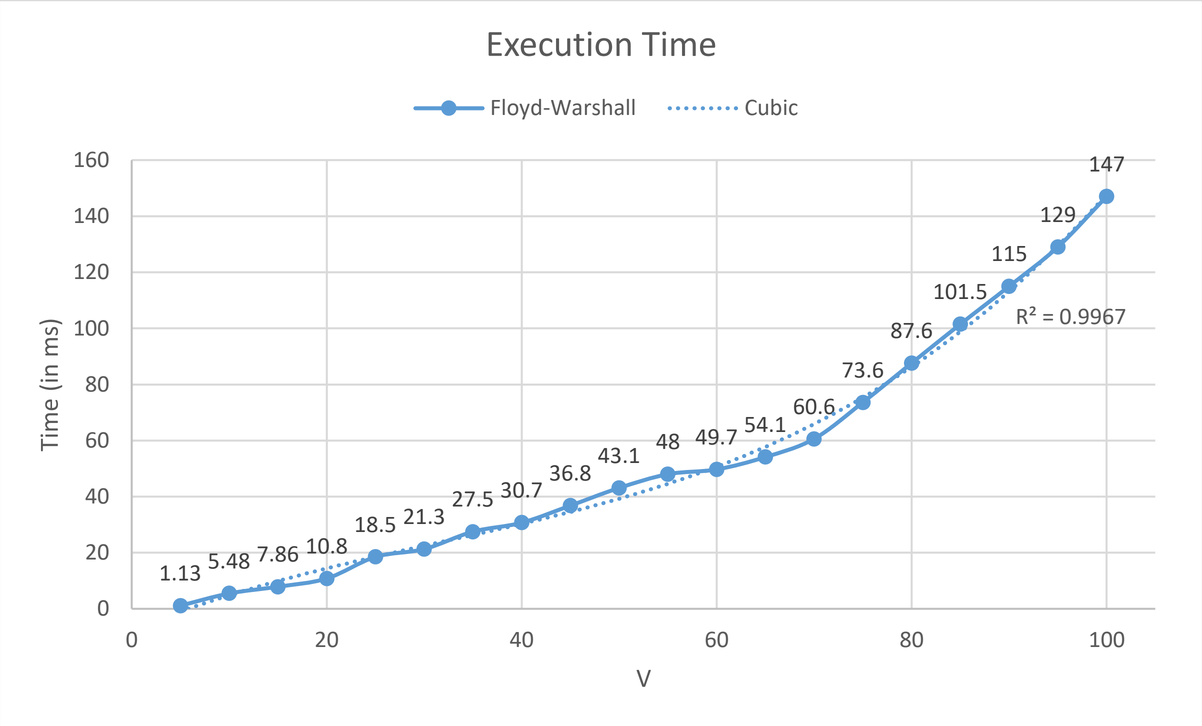
9. Graphs

9.1. Execution Time

V	Time (in ms)
5	1.13
10	5.48
15	7.86
20	10.8
25	18.5
30	21.3
35	27.5
40	30.7
45	36.8
50	43.1
55	48
60	49.7
65	54.1

70	60.6
75	73.6
80	87.6
85	101.5
90	115
95	129
100	147

Table: Floyd-Warshall execution time from v = 5 to 100

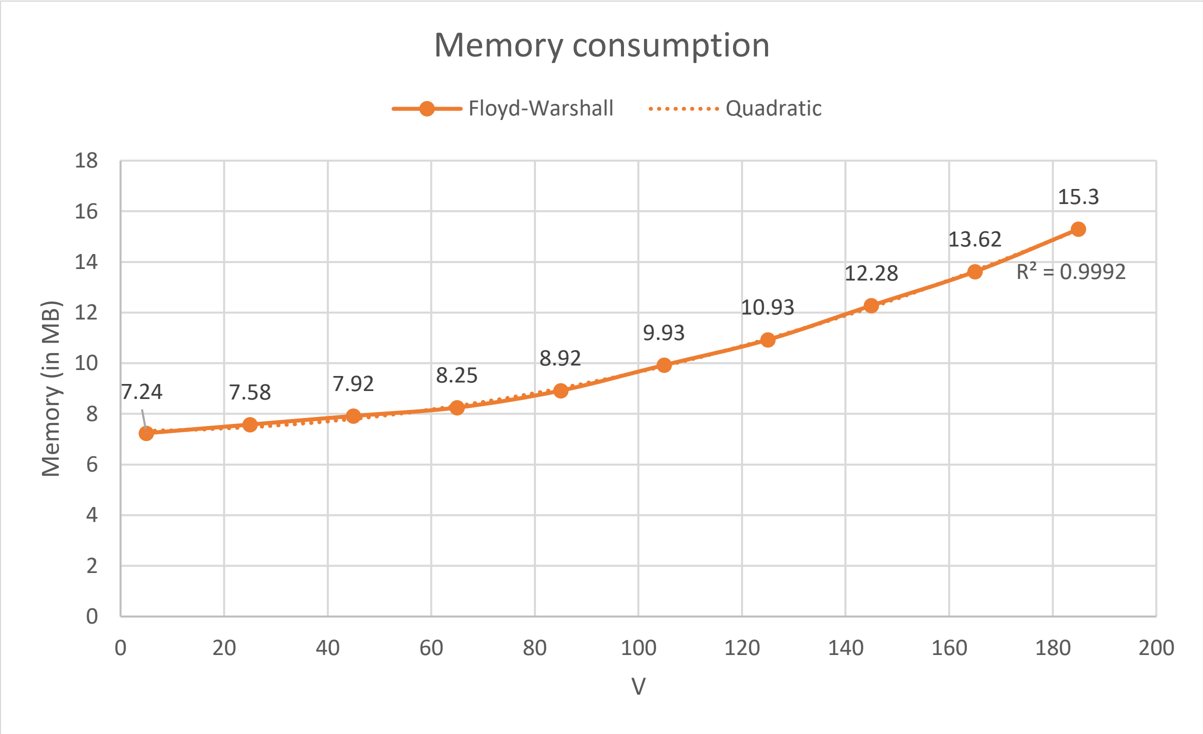


Graph: Scatter plot of the above data with a cubic trend line

9.2. Memory consumption

V	Memory (in MB)
5	7.24
25	7.58
45	7.92
65	8.25
85	8.92
105	9.93
125	10.93
145	12.28
165	13.62
185	15.3

Table: Floyd-Warshall execution time from v = 5 to 185

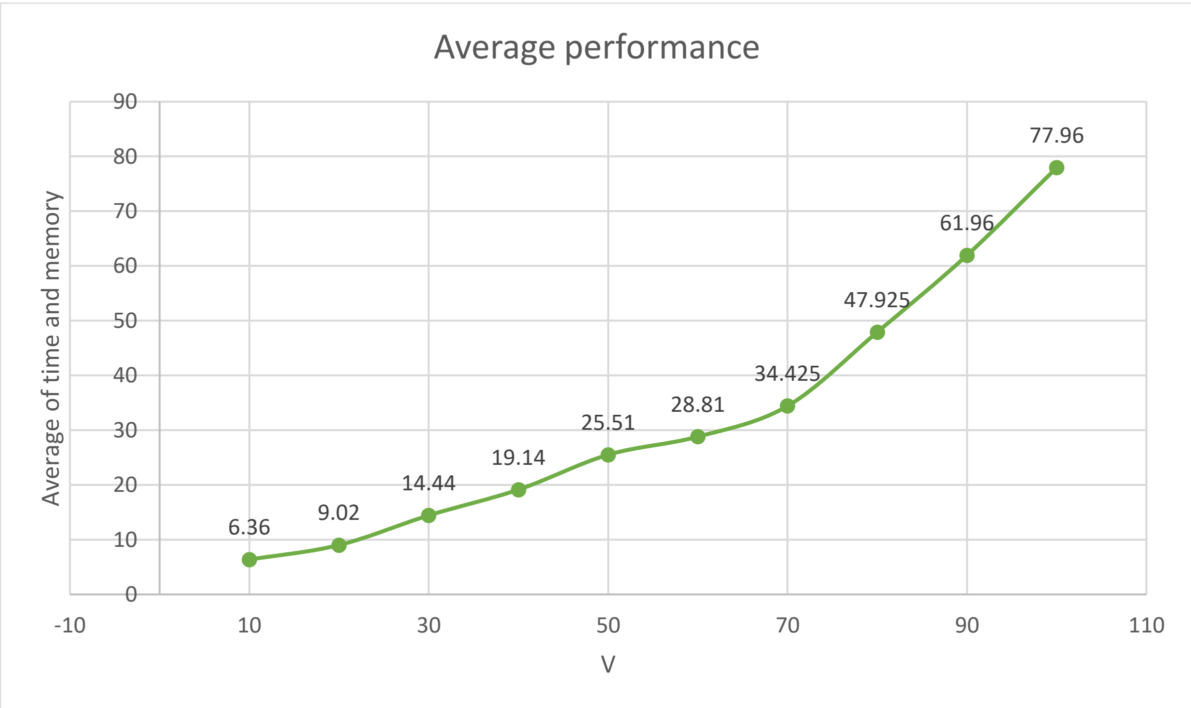


Graph: Scatter plot of the above data with a quadratic trend line

9.3. Average performance

V	Average
10	6.36
20	9.02
30	14.44
40	19.14
50	25.51
60	28.81
70	34.425
80	47.925
90	61.96
100	77.96

Table: Average performance for Floyd-Warshall



Graph: Scatter plot of the above data

10. References and tools

[1] Shortest path from https://en.wikipedia.org/wiki/Shortest_path_problem

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, "*Introduction to Algorithms*", MIT Press, 2009

[3] Algorithm pseudocode from https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm

The NetBeans profiler for profiling the performance of the algorithm.

Microsoft Excel for plotting graphs of the gathered data.

Google Drawings for diagrammatic representation of data.