Kimtaiyo Mech

Professor Huong Luu

CS 3310

May 20, 2022

Project 2 Report

Objective: find the length of all pairs shortest paths for directed weighted graphs with all non-negative weights using Dijkstra's algorithm as a subroutine and Floyd-Warshall algorithm.

Implementation: C++

The graphs are represented by adjacency matrices, and the matrices are represented by 2d vectors. The graphs are non-negatively weighted and directed. All indexes are from 0 to n-1

Dijkstra's algorithm is used for finding the shortest path from one source vertex to other vertices. However, this algorithm doesn't work for negatively weighted graphs. By using Dijkstra's algorithm as a subroutine, we can find the shortest paths between all pairs of vertices. This means that we will run Dijkstra's algorithm for every vertex in the graph. Whereas the Floyd-Warshall algorithm is designed with the goal of finding the shortest path between all pairs of vertices. Furthermore, it can also handle negatively weighted graphs.

Testing:

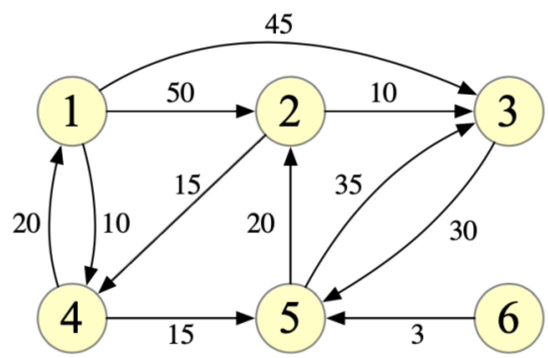
- It is done on XCode compiler.
- Each edge is randomly given the weight between [1, 20]
- To get the time spent on each graph with n vertices, each algorithm is tested 10 times with different matrices of the same size and the total time is divided by 10.

- Number of vertices are incremented by 10 each time. e.g., 10, 20, 30, …
- The largest size of n is 660.
- Time is calculated in microseconds.

Sanity Check:

- Checking Dijkstra can correctly find single path.

Graph:



Adjacency matrix:

```
Matrix graph = {
    {0, 50,45,10,inf,inf},
    {inf,0,10,15,inf,inf},
    {inf,inf,0,inf,30,inf},
    {20,inf,inf,0,15,inf},
    {inf,20,35,inf,0,inf},
    {inf,inf,inf,inf,3,0}};
```

v	1	2	3	4	5	6
cost	0	45	45	10	25	∞
prev	1	5	1	1	4	1

Shortest path from 1:

- $1 \to 4 \to 5 \to 2$: length 45
- $1 \to 3$: length 45
- $1 \to 4$: length 10
- $1 \to 4 \to 5$: length 25
- $1 \to 6$: length ∞ (there is no path from $1 \to 6$)

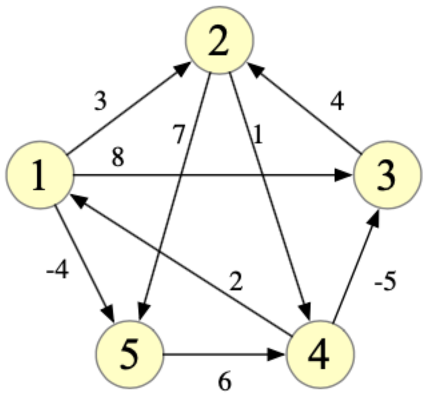
```
cost: 0 45 45 10 25 ∞
prev: 0 4 0 0 3 0

0 -> 3 -> 4 -> 1: length 45
0 -> 2: length 45
0 -> 3: length 10
0 -> 3 -> 4: length 25
0 -> 5: length ∞
```

- Note: my index starts from 0 to n-1

- Checking Floyd-Warshall algorithm

Graph:	Adjacency Matrix:




```
Matrix graphFW = {
    {0,3,8,inf,-4},
    {inf,0,inf,1,7},
    {inf,4,0,inf,inf},
    {2,inf,-5,0,inf},
    {inf,inf,inf,6,0}};
```

D^5

	1	2	3	4	5
1	0	1	-3	2	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

P^5

	1	2	3	4	5
1	NIL	3	4	5	1
2	4	NIL	4	2	1
3	4	3	NIL	2	1
4	4	3	4	NIL	1
5	4	3	4	5	NIL

```
Distance matrix:
| 0  1  -3 2  -4|
| 3  0  -4 1  -1|
| 7  4  0  5  3 |
| 2  -1 -5 0  -2|
| 8  5  1  6  0 |

Predecessor matrix:
| 0  2  3  4  0 |
| 3  1  3  1  0 |
| 3  2  2  1  0 |
| 3  2  3  3  0 |
| 3  2  3  4  4 |
```

Shortest path from 5 to 2: 5 – 4 – 3 – 2

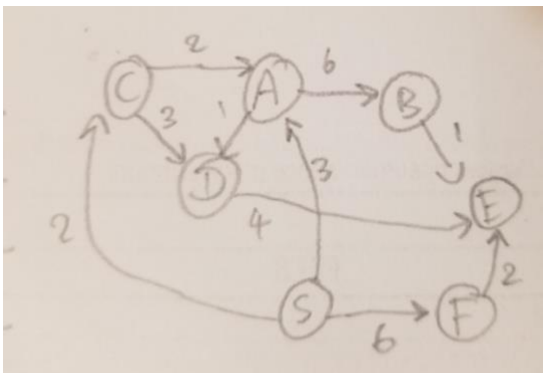
```
Shortest path from 4:
4 -> 3 -> 0: length 8
4 -> 3 -> 2 -> 1: length 5
4 -> 3 -> 2: length 1
4 -> 3: length 6
```

- For the diagonals on the predecessor matrix, I changed from NIL to the index of the row to make it consistent with the predecessor matrix generated by the Dijkstra subroutine algorithm. This way, I can iterate through each element of both predecessor matrices and can easily compare them. This change doesn't affect the ability to find the actual path because the diagonals are ignored.

- Checking both Floyd-Warshall and Dijkstra subroutine:

  - the vertices {A, B, C, D, E, F, S} are represented by {0, 1, 2, 3, 4, 5, 6}

Graph



Adjacency matrix

	A	B	C	D	E	F	S
A	0	6	∞	1	∞	∞	∞
B	∞	0	∞	∞	1	∞	∞
C	2	∞	0	3	∞	∞	∞
D	∞	∞	∞	0	4	∞	∞
E	∞	∞	∞	∞	0	∞	∞
F	∞	∞	∞	∞	2	0	∞
S	3	∞	2	∞	∞	6	0

v	A	**B**	C	D	E	F	S
dist	3	**9**	2	4	8	6	0
prev	S	**A**	S	A	D	S	S

```
Using Dijkstra's algorithm as a subroutine:
   Distance matrix:
| 0  6  ∞  1  5  ∞  ∞ |
| ∞  0  ∞  ∞  1  ∞  ∞ |
| 2  8  0  3  7  ∞  ∞ |
| ∞  ∞  ∞  0  4  ∞  ∞ |
| ∞  ∞  ∞  ∞  0  ∞  ∞ |
| ∞  ∞  ∞  ∞  2  0  ∞ |
| 3  9  2  4  8  6  0 |

   Predecessor matrix:
| 0  0  0  0  3  0  0 |
| 1  1  1  1  1  1  1 |
| 2  0  2  2  3  2  2 |
| 3  3  3  3  3  3  3 |
| 4  4  4  4  4  4  4 |
| 5  5  5  5  5  5  5 |
| 6  0  6  0  3  6  6 |
```

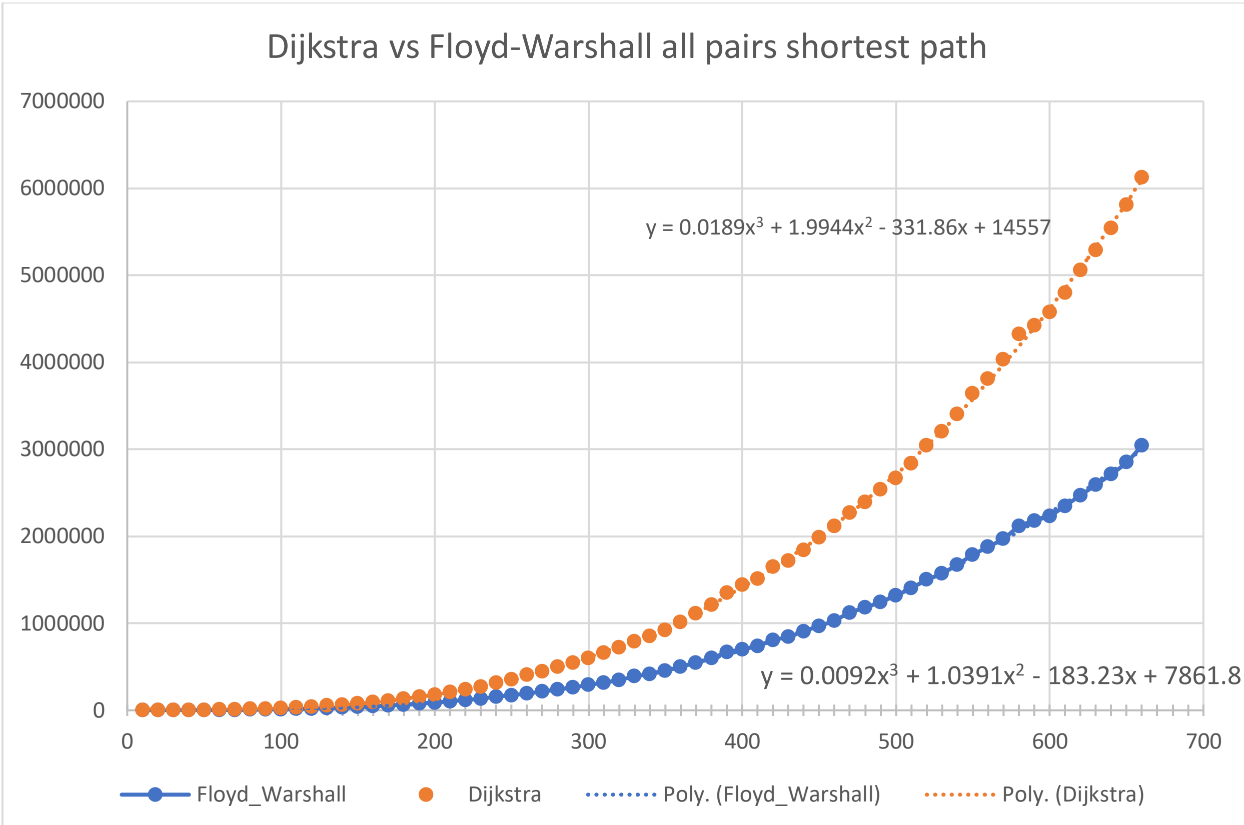
<table>
<tr>
<td></td>
<td>
Using Floyd-Warshall algorithm:
Distance matrix:
| 0 6 ∞ 1 5 ∞ ∞ |
| ∞ 0 ∞ ∞ 1 ∞ ∞ |
| 2 8 0 3 7 ∞ ∞ |
| ∞ ∞ ∞ 0 4 ∞ ∞ |
| ∞ ∞ ∞ ∞ 0 ∞ ∞ |
| ∞ ∞ ∞ ∞ 2 0 ∞ |
| 3 9 2 4 8 6 0 |

Predecessor matrix:
| 0 0 0 0 3 0 0 |
| 1 1 1 1 1 1 1 |
| 2 0 2 2 3 2 2 |
| 3 3 3 3 3 3 3 |
| 4 4 4 4 4 4 4 |
| 5 5 5 5 5 5 5 |
| 6 0 6 0 3 6 6 |

Comaprison:
Dist: ✅
Pre: ✅
</td>
</tr>
<tr>
<td>
Shortest path from S:
• $S \to A$: length 3
• $S \to A \to B$: length 9
• $S \to C$: length 2
• $S \to A \to D$: length 4
• $S \to A \to D \to E$: length 8
• $S \to F$: length 6
</td>
<td>
Shortest path from 6 to all vertex:
Dijkstra:
6 -> 0: length 3
6 -> 0 -> 1: length 9
6 -> 2: length 2
6 -> 0 -> 3: length 4
6 -> 0 -> 3 -> 4: length 8
6 -> 5: length 6
Floyd-Warshall:
6 -> 0: length 3
6 -> 0 -> 1: length 9
6 -> 2: length 2
6 -> 0 -> 3: length 4
6 -> 0 -> 3 -> 4: length 8
6 -> 5: length 6
</td>
</tr>
</table>

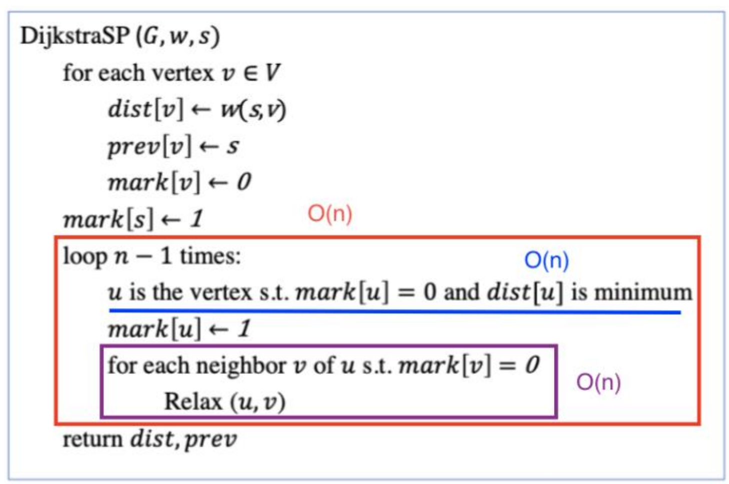
Data:

Dijkstra	**Floyd-Warshall**	**vertices**
128.348	34.1212	10
467.494	150.691	20
1237.31	450.484	30
2558.15	924.976	40
4094.31	1675.73	50
6512.63	2785.44	60
9565.36	4250.55	70
13753.8	6207.57	80
18659.3	8625.36	90
25003.1	11907.7	100
32640.4	15389.6	110
41243.5	19716.7	120
52533.6	25387.5	130
64491	31176.9	140
78722.6	38406	150
94553.4	46517.6	160
112009	54952.2	170
132764	65047.6	180
157507	76983	190
180941	89070.8	200
210692	102937	210
241423	118690	220
273243	134316	230
315341	156113	240
357265	173216	250
409285	194904	260
448739	214462	270
496831	238651	280
544860	263962	290
598838	289836	300
659234	318612	310
719170	349432	320
794609	388980	330
853491	417331	340
924667	452672	350
1.02E+06	499314	360
1.12E+06	546509	370
1.21E+06	595719	380
1.35E+06	669361	390
1.44E+06	699300	400
1.51E+06	740713	410
1.65E+06	808213	420
1.72E+06	843058	430
1.84E+06	908649	440
1.99E+06	968367	450
2.12E+06	1.03E+06	460
2.27E+06	1.12E+06	470
2.39E+06	1.18E+06	480
2.54E+06	1.24E+06	490
2.67E+06	1.32E+06	500
2.84E+06	1.40E+06	510
3.05E+06	1.50E+06	520
3.21E+06	1.57E+06	530
3.41E+06	1.68E+06	540
3.64E+06	1.79E+06	550
3.81E+06	1.88E+06	560
4.03E+06	1.97E+06	570
4.32E+06	2.11E+06	580
4.42E+06	2.18E+06	590
4.57E+06	2.23E+06	600
4.80E+06	2.35E+06	610
5.06E+06	2.47E+06	620
5.29E+06	2.59E+06	630
5.55E+06	2.71E+06	640
5.81E+06	2.85E+06	650
6.12E+06	3.04E+06	660



From the chart, we can observe that Floyd-Warshall algorithm performs better than using Dijkstra's algorithm as a subroutine, and the disparity increases further as the number of vertices increases.

Although Dijkstra's single path algorithm has the theoretical runtime of $O(n^2)$, it includes performing two $O(n^2)$ operations. These two operations are finding the min and iterating to the neighbor and perform relaxation.



So, when we use it as a subroutine and run it for every vertex, it becomes $O(n^3)$. However, it has a much larger Big-O constant than Floyd-Warshall, which has little to no Big-O constant.

AllPairsShortestPath(G, w)
    for $u \leftarrow 1$ to n do
        for $v \leftarrow 1$ to n do
            $D^0_{u,v} \leftarrow w(u, v)$
$O(n)$ for $k \leftarrow 1$ to n do
    $O(n)$ for $u \leftarrow 1$ to n do
        $O(n)$ for $v \leftarrow 1$ to n do   O(1)
            $\underline{D^k_{u,v} \leftarrow \min(D^{k-1}_{u,v}, D^{k-1}_{u,k} + D^{k-1}_{k,v})}$
    return D^k

Thus, both algorithms are theoretically $O(n^3)$, but the difference in Big-O constant shows up in the empirical data. From the trendline equations, we can also observe that it is a polynomial of order 3, which agrees with the theoretical complexity.

Conclusion:

I was able to show that even though Dijkstra's algorithm is meant for single path, it can still be used as a subroutine to find all paths. Even though using Dijkstra as a subroutine works and has the same theoretical time complexity as Floyd-Warshall, it is much better to use Floyd-Warshall as it is designed specifically to tackle the paths between all pairs problem. Due to time and computing power constraints, I couldn't show that trendline equation becoming more fitting of a third order polynomial.

Extra: getting the actual shortest path between any pair of vertices

Whenever a vortex is visited, the array pre, short for predecessor, keeps track of which vortex we are coming from. This strategy works for both Dijkstra ad Floyd-Warshall.

Dijkstra

In the DijkstraSP function:

```
// for each neighbor v of u s.t. mark[v] = 0
for (int j = 0; j < n; j++) {
    // relaxation
    if (mark[j] != 1) {
        int k = dist[u] + g[u][j];
        if (dist[j] > k) {
            dist[j] = k;
            pre[j] = u;
        }
    }
}
```

Floyd-Warshall

In the FloydWarshall function:

```
for (int k = 0; k < n; k++) {
    for (int u = 0; u < n; u++) {
        for (int v = 0; v < n; v++) {
            if (dist[u][v] > dist[u][k]  + dist[k][v]) {
                pre[u][v] = pre[k][v];
                dist[u][v] = dist[u][k]  + dist[k][v];
            }
        }
    }
}
```

To get the actual path, we backtrack from the target destination.

For example: source: 1, target: 2, we get 1 → 4 → 5 → 2

start from target

source

v	1	2	3	4	5	6
cost	0	45	45	10	25	∞
pre	1	5	1	1	4	1

- This function uses stacks to store all the vertices along the path. When we reach the source, the stack is popped into a string, which is then returned.

```
string getPathString(Array prev, int begin, int end) {
    string path = "";
    stack<int> stack;

    stack.push(end);
    int cur = prev[end];

    while(cur != begin) {
        stack.push(cur);
        cur = prev[cur];
    }
    stack.push(cur);

    while (!stack.empty()) {
        path += to_string(stack.top());
        stack.pop();
        path += (!stack.empty()) ? (" -> ") : ("");
    }

    return path;
}
```

```
Sanity check Dijkstra single path:
==================================

Adjacency matrix:
| 0  50 45 10 ∞  ∞ |
| ∞  0  10 15 ∞  ∞ |
| ∞  ∞  0  ∞  30 ∞ |
| 20 ∞  ∞  0  15 ∞ |
| ∞  20 35 ∞  0  ∞ |
| ∞  ∞  ∞  ∞  3  0 |

cost: 0 45 45 10 25 ∞
prev: 0 4 0 0 3 0

Shortest path from 0 to all vertices:
0 -> 3 -> 4 -> 1: length 45
0 -> 2: length 45
0 -> 3: length 10
0 -> 3 -> 4: length 25
0 -> 5: length ∞


Sanity check Floyd-Warshall:
===========================

Adjacency matrix:
| 0  3  8  ∞  -4|
| ∞  0  ∞  1  7 |
| ∞  4  0  ∞  ∞ |
| 2  ∞  -5 0  ∞ |
| ∞  ∞  ∞  6  0 |

Distance matrix:
| 0  1  -3 2  -4|
| 3  0  -4 1  -1|
| 7  4  0  5  3 |
| 2  -1 -5 0  -2|
| 8  5  1  6  0 |

Predecessor matrix:
| 0  2  3  4  0 |
| 3  1  3  1  0 |
| 3  2  2  1  0 |
| 3  2  3  3  0 |
| 3  2  3  4  4 |

Shortest path from 4 to all vertices:
4 -> 3 -> 0: length 8
4 -> 3 -> 2 -> 1: length 5
4 -> 3 -> 2: length 1
4 -> 3: length 6
```

```
Sanity check both Floyd-Warshall and Dijkstra subroutine:
==========================================================

Adjacency matrix for the sanity check graph:
| 0  6  ∞  1  ∞  ∞  ∞ |
| ∞  0  ∞  ∞  1  ∞  ∞ |
| 2  ∞  0  3  ∞  ∞  ∞ |
| ∞  ∞  ∞  0  4  ∞  ∞ |
| ∞  ∞  ∞  ∞  0  ∞  ∞ |
| ∞  ∞  ∞  ∞  2  0  ∞ |
| 3  ∞  2  ∞  ∞  6  0 |

Using Dijkstra's algorithm as a subroutine:
   Distance matrix:
| 0  6  ∞  1  5  ∞  ∞ |
| ∞  0  ∞  ∞  1  ∞  ∞ |
| 2  8  0  3  7  ∞  ∞ |
| ∞  ∞  ∞  0  4  ∞  ∞ |
| ∞  ∞  ∞  ∞  0  ∞  ∞ |
| ∞  ∞  ∞  ∞  2  0  ∞ |
| 3  9  2  4  8  6  0 |

   Predecessor matrix:
| 0  0  0  0  3  0  0 |
| 1  1  1  1  1  1  1 |
| 2  0  2  2  3  2  2 |
| 3  3  3  3  3  3  3 |
| 4  4  4  4  4  4  4 |
| 5  5  5  5  5  5  5 |
| 6  0  6  0  3  6  6 |

Using Floyd-Warshall algorithm:
   Distance matrix:
| 0  6  ∞  1  5  ∞  ∞ |
| ∞  0  ∞  ∞  1  ∞  ∞ |
| 2  8  0  3  7  ∞  ∞ |
| ∞  ∞  ∞  0  4  ∞  ∞ |
| ∞  ∞  ∞  ∞  0  ∞  ∞ |
| ∞  ∞  ∞  ∞  2  0  ∞ |
| 3  9  2  4  8  6  0 |

   Predecessor matrix:
| 0  0  0  0  3  0  0 |
| 1  1  1  1  1  1  1 |
| 2  0  2  2  3  2  2 |
| 3  3  3  3  3  3  3 |
| 4  4  4  4  4  4  4 |
| 5  5  5  5  5  5  5 |
| 6  0  6  0  3  6  6 |
```

```
Comaprison:
Dist: ✅
Pre: ✅

Shortest path from 6 to all vertices:
Dijkstra:
6 -> 0: length 3
6 -> 0 -> 1: length 9
6 -> 2: length 2
6 -> 0 -> 3: length 4
6 -> 0 -> 3 -> 4: length 8
6 -> 5: length 6

Floyd-Warshall:
6 -> 0: length 3
6 -> 0 -> 1: length 9
6 -> 2: length 2
6 -> 0 -> 3: length 4
6 -> 0 -> 3 -> 4: length 8
6 -> 5: length 6
```

```
//
//  main.cpp
//  Project2
//
//  Created by Kimtaiyo Mech on 5/19/22.
//

#include <iostream>
#include <utility>
#include <vector>
#include <stack>
#include <random>
#include <chrono>
#include <math.h>
#include <string>
#include <iomanip>
using namespace std;
using namespace std::chrono;

typedef vector<int> Array;
typedef vector<vector<int>> Matrix;
int inf = 999; //represents infinity

pair<Array, Array> DijkstraSP(Matrix graph, int s);
pair<Matrix, Matrix> DijkstraAP (Matrix graph);
pair<Matrix, Matrix> FloydWarshall(Matrix graph);

string getPathString(Array prev, int begin, int end);
void printPath(Array dist, Array prev, int source);
void printDistPrev(Array dist, Array prev);
void printMatrix(Matrix matrix);

Matrix genGraph(int n);

bool compare(Matrix m1, Matrix m2);
pair<double, double> testing(int vertex, int rounds);

int main(int argc, const char * argv[]) {

    cout << "Sanity check Dijkstra single path:\n";
    cout << "==================================\n\n";
    Matrix graphDijkstra = {
        {0, 50,45,10,inf,inf},
        {inf,0,10,15,inf,inf},
        {inf,inf,0,inf,30,inf},
        {20,inf,inf,0,15,inf},
        {inf,20,35,inf,0,inf},
        {inf,inf,inf,inf,3,0}};

    cout << "Adjacency matrix:\n";
    printMatrix(graphDijkstra);
```

```
cout << endl;

auto [dist, prev] = DijkstraSP(graphDijkstra, 0);

printDistPrev(dist, prev);
cout << endl << endl;
cout << "Shortest path from 0 to all vertices:\n";
printPath(dist, prev, 0);
cout << endl;

cout << "\nSanity check Floyd-Warshall:\n";
cout << "===========================\n\n";
Matrix graphFW = {
    {0,3,8,inf,-4},
    {inf,0,inf,1,7},
    {inf,4,0,inf,inf},
    {2,inf,-5,0,inf},
    {inf,inf,inf,6,0}};

cout << "Adjacency matrix:\n";
printMatrix(graphFW);
cout << endl;

auto [distMatrix, preMatrix] = FloydWarshall(graphFW);

cout << "Distance matrix:\n";
printMatrix(distMatrix);
cout << "\nPredecessor matrix:\n";
printMatrix(preMatrix);

cout << "\nShortest path from 4 to all vertices: \n";
printPath(distMatrix[4], preMatrix[4], 4);

cout << "\n\nSanity check both Floyd-Warshall and Dijkstra subroutine:\n";
cout << "=========================================================\n\n";
Matrix sanityDFW = {
    {0, 6, inf, 1, inf, inf, inf},
    {inf, 0, inf, inf, 1, inf, inf},
    {2, inf, 0, 3, inf, inf, inf},
    {inf, inf, inf, 0, 4, inf, inf},
    {inf, inf, inf, inf, 0, inf, inf},
    {inf, inf, inf, inf, 2, 0, inf},
    {3, inf, 2, inf, inf, 6, 0}};

cout << "Adjacency matrix for the sanity check graph:\n";
printMatrix(sanityDFW);
cout << endl;

cout << "Using Dijkstra's algorithm as a subroutine:\n";
auto [distMatrix1, preMatrix1] = DijkstraAP(sanityDFW);
cout << "   Distance matrix:\n";
```

```
    printMatrix(distMatrix1);
    cout << endl;
    cout << "   Predecessor matrix:\n";
    printMatrix(preMatrix1);
    cout << endl;

    cout << "Using Floyd-Warshall algorithm:\n";
    auto [distMatrix2, prevMatrix2] = FloydWarshall(sanityDFW);
    cout << "   Distance matrix:\n";
    printMatrix(distMatrix2);
    cout << endl;
    cout << "   Predecessor matrix:\n";
    printMatrix(prevMatrix2);
    cout << "\nComaprison: " << endl;
    cout << "Dist: " << (compare(distMatrix1, distMatrix2) ? "✅" : "❌") <<
     endl;
    cout << "Pre: " << (compare(preMatrix1, prevMatrix2) ? "✅" : "❌") <<
     endl << endl;

    cout << "Shortest path from 6 to all vertices:\n";
    cout << "Dijkstra: \n";
    printPath(distMatrix1[6], preMatrix1[6], 6);
    cout << "\nFloyd-Warshall: \n";
    printPath(distMatrix2[6], preMatrix1[6], 6);

//    This bit is for testing the algorithm time

//    cout << endl << endl;
//    for (int i = 10; i <= 1000; i+=10) {
//        cout << "Calculating time for " << i << " verticces: ...\n";
//        auto [time1, time2] = testing(i,10);
//        cout << "Dijkstra: " << time1;
//        cout << "\nFloyd-Warshall: " << time2 << endl << endl;
//    }

    return 0;
}

pair<double, double> testing(int vertex, int rounds) {
    double time1 = 0.0, time2 = 0.0;

    for (int i = 0; i < rounds; i++) {
        Matrix graph = genGraph(vertex);

        std::chrono::steady_clock::time_point start;
        std::chrono::steady_clock::time_point stop;
        duration<double, std::micro> duration;

        start = high_resolution_clock::now();
        auto [dist1, pre1] = DijkstraAP(graph);
```

```
        stop = high_resolution_clock::now();
        duration = stop - start;

        time1 += duration.count();

        start = high_resolution_clock::now();
        auto [dist2, pre2] = FloydWarshall(graph);
        stop = high_resolution_clock::now();
        duration = stop - start;

        time2 += duration.count();
    }

    return {time1/rounds, time2/rounds};
}

bool compare(Matrix m1, Matrix m2) {
    int n = (int)m1.size();
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (m1[i][j] != m2[i][j]) {
                return false;
            }
        }
    }
    return true;
}

pair<Array, Array> DijkstraSP(Matrix g, int s) {
    int n = (int)g.size();
    Array dist(n);
    Array pre(n);
    Array mark(n); // only contains 1 or 0
    bool initU = true;

    for (int i = 0; i < n; i++) {
        dist[i] = g[s][i];
        pre[i] = s;
        mark[i] = 0;
    }
    mark[s] = 1;

    for (int i = 0; i < n-1; i++) {
        // finding the vertex u s.t. mark[u] = 0 and dist[u] is minimum
        int u = 0;

        for (int j = 0; j < n; j++) {
            if (mark[j] == 0) {
                // finding a value to initialize u to before comparing. we
                 initialize it to one of the non-marked
                if (initU) {
```

```
                u = j;
                initU = false;
            }
            if (dist[j] < dist[u]) {
                u = j;
            }
        }
    }

    mark[u] = 1;

    // for each neighbor v of u s.t. mark[v] = 0
    for (int j = 0; j < n; j++) {
        // relaxation
        if (mark[j] != 1) {
            int k = dist[u] + g[u][j];
            if (dist[j] > k) {
                dist[j] = k;
                pre[j] = u;
            }
        }
    }

    initU = true;
}

return  {dist, pre};
}

pair<Matrix, Matrix> DijkstraAP (Matrix g) {
    Matrix distMatrix;
    Matrix prevMatrix;
    int n = (int)g.size();

    for (int i = 0; i < n; i++) {
        auto [dist, prev] = DijkstraSP(g, i);
        distMatrix.push_back(dist);
        prevMatrix.push_back(prev);
    }

    return {distMatrix, prevMatrix};
}

pair<Matrix, Matrix> FloydWarshall(Matrix graph) {
    Matrix dist;
    Matrix pre;

    int n = (int)graph.size();

    //initialize both matrices
    dist = graph;
```

```
    pre = graph;
    for (int u = 0; u < n; u++) {
        for (int v = 0; v < n; v++) {
            if (u == v || graph[u][v] == inf) {
                pre[u][v] = u; // can also set to NIL to indicate no path
            }
            else if (u != v && graph[u][v] < inf) {
                pre[u][v] = u;
            }
        }
    }

    for (int k = 0; k < n; k++) {
        for (int u = 0; u < n; u++) {
            for (int v = 0; v < n; v++) {
                if (dist[u][v] > dist[u][k]  + dist[k][v]) {
                    pre[u][v] = pre[k][v];
                    dist[u][v] = dist[u][k]  + dist[k][v];
                }
            }
        }
    }

    return {dist, pre};
}

string getPathString(Array prev, int begin, int end) {
    string path = "";
    stack<int> stack;

    stack.push(end);
    int cur = prev[end];

    while(cur != begin) {
        stack.push(cur);
        cur = prev[cur];
    }
    stack.push(cur);

    while (!stack.empty()) {
        path += to_string(stack.top());
        stack.pop();
        path += (!stack.empty()) ? (" -> ") : ("");
    }

    return path;
}

void printPath(Array dist, Array prev, int source) {
    int n = (int)dist.size();
    for (int i =  0; i < n; i++) {
```

```
        if (i != source) {
            int val = dist[i];
            string str = ((val == 999) ? ("\u221E ") : (to_string(val)));
            cout << getPathString(prev, source, i) << ": length " << str <<
             endl;
        }
    }
}

void printDistPrev(Array dist, Array prev) {
    int n = (int)dist.size();
    cout << "cost: ";
    for (int i = 0; i < n; i++) {
        int val = dist[i];
        string str = ((val == 999) ? ("\u221E ") : (to_string(val)));
        cout << str << " ";
    }
    cout << "\nprev: ";
    for (int i = 0; i < n; i++) {
        cout << prev[i] << " ";
    }
}

void printMatrix(Matrix matrix) {
    int n = (int)matrix.size(), width = 2;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            int val = matrix[i][j];
            string str = ((val == 999) ? ("\u221E ") : (to_string(val)));

            if (j == 0)
                cout << "| " << setw(width) << left << str;
            else if (j == n-1)
                cout << " " << setw(width) << left << str << "|\n";
            else
                cout << " " << setw(width) << left << str;
        }
    }
}

Matrix genGraph(int n) {
    Matrix g;
    for (int i = 0; i < n; i++) {
        Array temp(n, inf);
        g.push_back(temp);
    }
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i == j) {
                g[i][j] = 0;
            }
```

```
            else {
                random_device dev;
                mt19937 rng(dev());
                uniform_int_distribution<mt19937::result_type> dist1(0,2);

                // probability of two vertices having an edge: 0.66
                if(dist1(rng) != 0) {
                    uniform_int_distribution<mt19937::result_type> dist2(1,20);
                    g[i][j] = dist2(rng); // random weight of edge
                }
            }
        }
    }

    return g;
}
```