GT LAB PRS Project

Graph Theory

Optimal Routing in Networks using Floyd-Warshall and Johnson algorithms

Kunal Sharma
2K18/MC/060

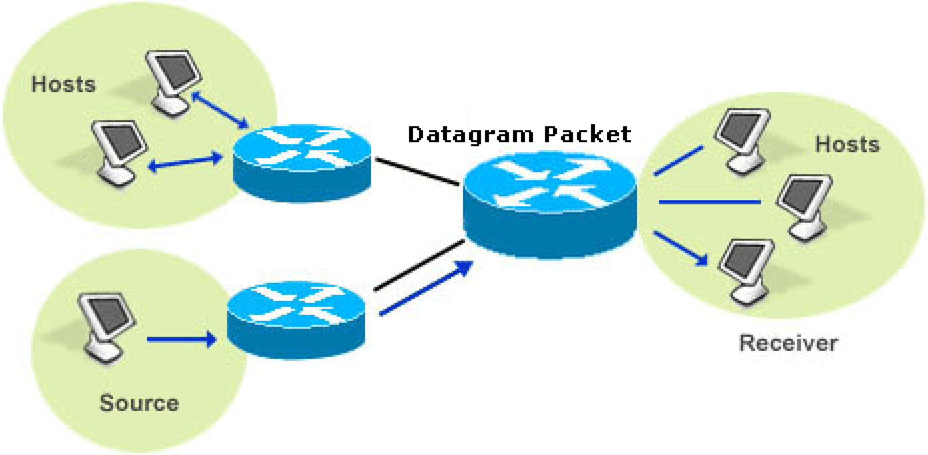
Janardan Upadhyay
2K18/MC/047

Department of Applied Mathematics

Submitted To:- Prof Sangita Kansal

Index

Abstract



Datagrams can move from a source to a destination in a Network via a variety of pathways. The process of determining a path for traffic in a network or across networks is known as routing. Routing concepts may be applied to any sort of network. The goal of optimal routing is to choose the best available path. Floyd-Warshall and Johnson's techniques can be used to identify the best path when implementing static routing. We compare the run time efficiency of both techniques on Dense and Sparse networks of various sizes as part of our study. In many network settings comparison analysis helps us discover the optimum method for static routing.

Introduction

1. Objective and Motivation of Studying the Network Routing Algorithm:

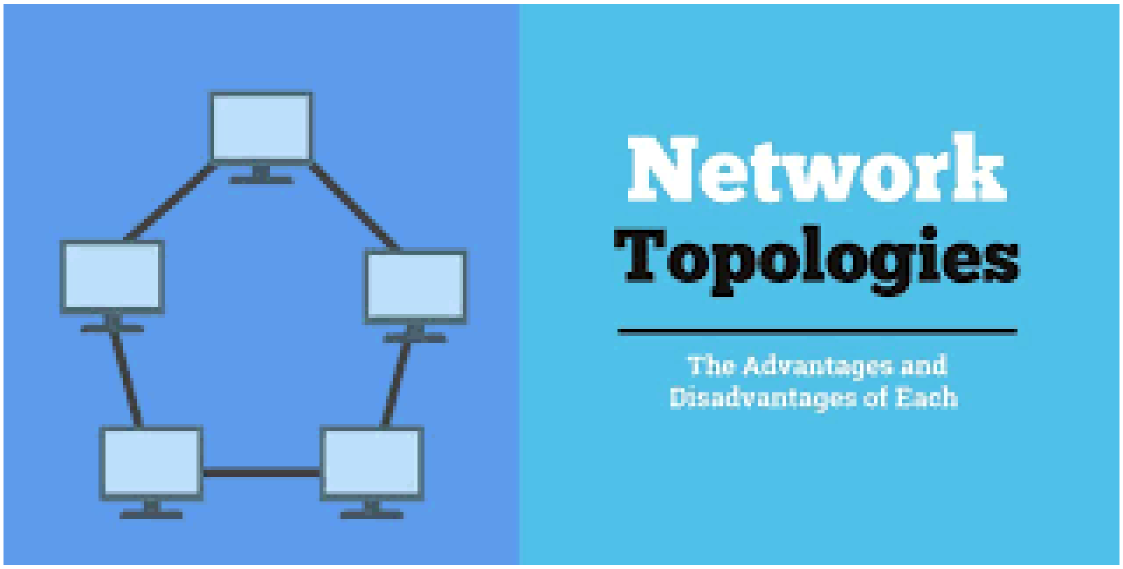
A. A datagram can move from a source to a destination in a Network via a variety of pathways. Routing is the process of choosing a path for traffic inside a network, as well as between and across different networks, and its concepts apply to any sort of network, from telephone networks to public transit. Optimal Routing ensures that the best possible path is chosen to meet the user's needs. Floyd-Warshall and Johnson's methods are two algorithms that may be used to discover the best route when using Static Routing. We hope to examine the run time efficiency of both methods on Dense and Sparse networks of various sizes as part of our study.

Packet
Host
Transmission medium

B.The act of picking a path across one or more networks is known as network routing, and its concepts may be applied to any form of network, from telephone networks to public transit. Routing chooses the pathways for Internet Protocol (IP) packets to go from their origin to their destination in packet-switching networks like the Internet. Routers, which are specialised pieces of network gear, make these Internet routing decisions. Internal routing tables are used by routers to make judgments about how to route packets along network routes. A routing table keeps track of the pathways packets should take to reach each of the router's destinations.

C. It's critical to understand which optimum routing method is ideal in order to make the most of the limited resources available and get the highest possible output. A comparison of the two static optimum routing methods can save money and effort by determining which algorithm is better for the underlying network. Especially in growing voice over IP (VoIP) networks, quality of service (QoS)-centric virtual private networks (VPNs), and networks carrying other delay-sensitive traffic flows, minimising packet delay is crucial. A specific instance of a classical inverse shortest-path issue is finding optimum pathways that result in shortest-path routing.

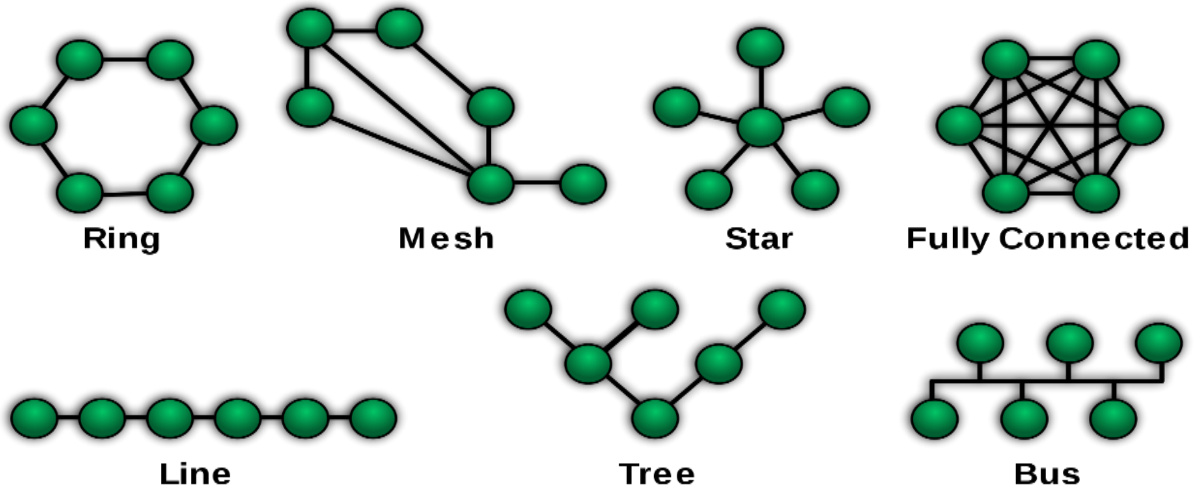
2. Academic Research and Topics Involved in NPR:



A computer network is a group of computers that use a set of common communication protocols over digital interconnections for the purpose of sharing resources located on or provided by the network nodes. The interconnections between nodes are formed from a broad spectrum of telecommunication network technologies, based on physically wired, optical, and wireless radio-frequency methods that may be arranged in a variety of network topologies

Network Topologies

Network topology is the layout, pattern, or organizational hierarchy of the interconnection of network hosts, in contrast to their physical or geographic location. Typically, most diagrams describing networks are arranged by their topology. The network topology can affect throughput, but reliability is often more critical. With many technologies, such as bus networks, a single failure can cause the network to fail entirely. In general, the more interconnections there are, the more robust the network is; but the more expensive it is to install.



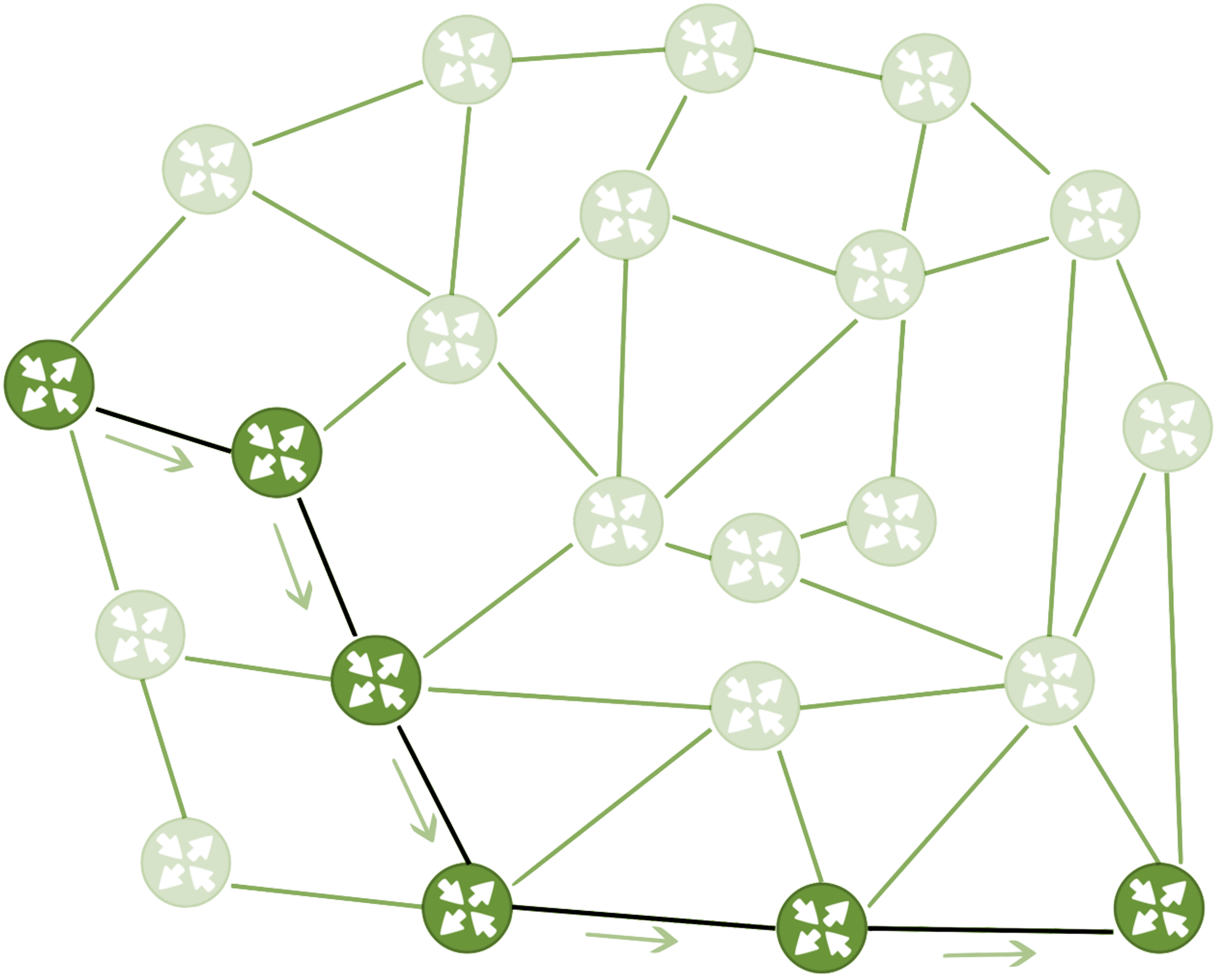
Common layouts are:

- **Bus network:** all nodes are connected to a common medium along this medium. This was the layout used in the original Ethernet, called 10BASE5 and 10BASE2. This is still a common topology on the data link layer, although modern physical layer variants use point-to-point links instead.
- **Line network:** type of bus topology in that all nodes are connected in line one by one with a host computer. The nodes are connected through the cables, there is a bus to carry information from one node to another node. This is the simple topology to create a network adding or removing the node affects the whole network. If any node or cable is in failure then the whole network will be affected.
- **Star network:** all nodes are connected to a special central node. This is the typical layout found in a Wireless LAN, where each wireless client connects to the central Wireless access point.

- **Ring network:** each node is connected to its left and right neighbour node, such that all nodes are connected and that each node can reach each other node by traversing nodes left- or rightwards. The Fiber Distributed Data Interface (FDDI) made use of such a topology.
- **Mesh network:** each node is connected to an arbitrary number of neighbours in such a way that there is at least one traversal from any node to any other.
- **Fully connected network:** each node is connected to every other node in the network.
- **Tree network:** nodes are arranged hierarchically.

3. Routing and its Types:

Routing is the process of selecting a path for traffic in a network or between or across multiple networks, performed in various types of networks including Packet switched networks and Circuit switched networks. In packet switching networks, routing is the higher-level decision making that directs network packets from their source toward their destination through intermediate network nodes by specific packet forwarding mechanisms. Packet forwarding is the transit of network packets from one network interface to another. Intermediate nodes are typically network hardware devices.



Types of Routing

Static routing: process in which we have to manually add routes in the routing table.

Advantages

- No routing overhead for the router CPU, which means a cheaper router can be used to do routing.
- It adds security because only administrators can allow routing to particular networks.

Disadvantages

- For a large network, it is a hectic task for the administrator to manually add each route for the network in the routing table on each router.
- The administrator should have good knowledge of the topology. If a new administrator comes, then he has to manually add each route so he should have very good knowledge of the routes of the topology.

Default Routing: A special case of static routing, this is the method where the router is configured to send all packets towards a single router (next hop). It doesn’t matter to which network the packet belongs, it is forwarded to the router which is configured for default routing. It is generally used with stub routers. A stub router is a router which has only one route to reach all other networks.

Dynamic Routing: Dynamic routing makes automatic adjustment of the routes according to the current state of the route in the routing table. Dynamic routing uses protocols to discover network destinations and the routes to reach it. RIP and OSPF are the best examples of dynamic routing protocol. Automatic adjustment will be made to reach the network destination if one route goes down.

A dynamic protocol has the following features:

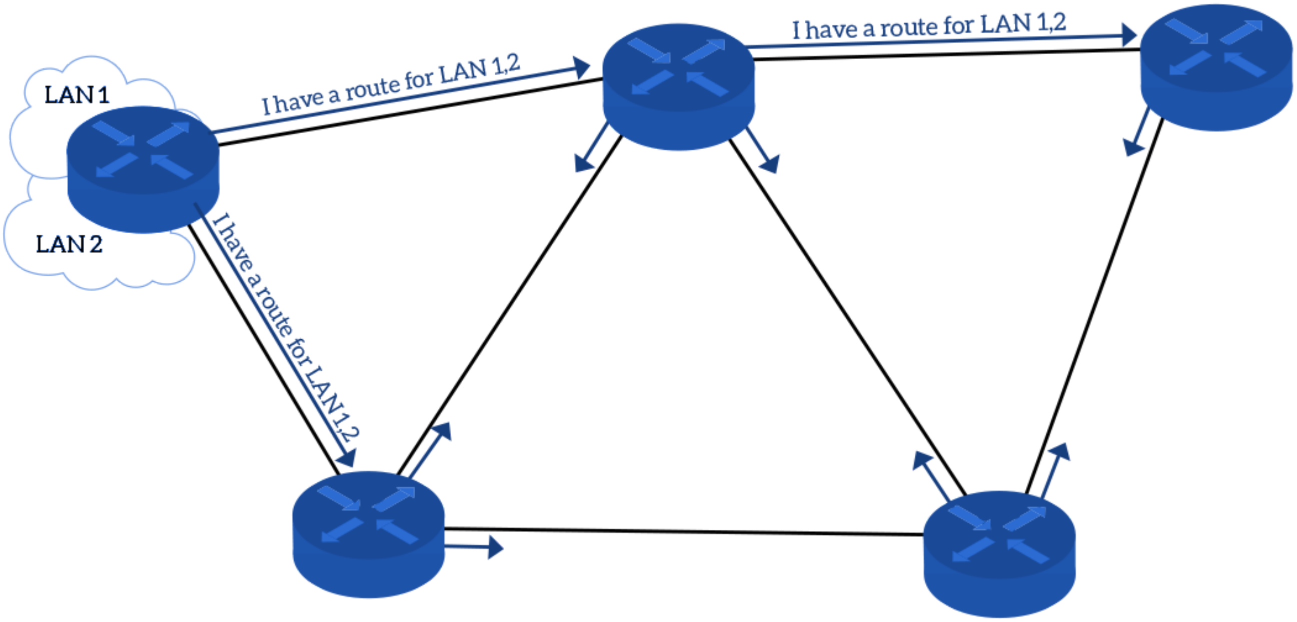
1. The routers should have the same dynamic protocol running in order to exchange routes.
2. When a router finds a change in the topology then the router advertises it to all other routers.

Advantages

- Easy to configure.
- More effective at selecting the best route to a destination remote network and also for discovering remote networks.

Disadvantages

- Consumes more bandwidth for communicating with other neighbors.
- Less secure than static routing.


LAN 1
LAN 2
I have a route for LAN 1,2
I have a route for LAN 1,2
I have a route for LAN 1,2

Methodology

2.3 Static Routing

Static routing is a form of routing that occurs when a router uses a manually-configured routing entry, rather than information from dynamic routing traffic. In many cases, static routes are manually configured by a network administrator by adding in entries into a routing table, though this may not always be the case. Unlike dynamic routing, static routes are fixed and do not change if the network is changed or reconfigured. Static routing and dynamic routing are not mutually exclusive. Both dynamic routing and static routing are usually used on a router to maximise routing efficiency and to provide backups in the event that dynamic routing information fails to be exchanged. Static routing can also be used in stub networks, or to provide a gateway of last resort.

Algorithms to find shortest paths in (sub)network

The (sub)network is mapped as a graph with the nodes as vertices and connections as edges. Then graph theory algorithms are applied to obtain the shortest path from the source node to all other nodes.

Algorithms like Dijkstra's shortest path algorithm and Bellman-Ford's algorithm are commonly used on each node to determine the routing table entries. All pairs shortest path algorithms like Floyd-Warshall's algorithm and Johnson's algorithm can be used to determine the routing table for the whole (sub)network. These may be used by the network administrator while manually configuring the routing tables.

2.3.1 Dijkstra's Algorithm

Dijkstra's algorithm (or Dijkstra's Shortest Path First algorithm, SPF algorithm) is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.

The algorithm exists in many variants. Dijkstra's original algorithm found the shortest path between two given nodes, but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree.

For a given source node in the graph, the algorithm finds the shortest path between that node and every other. It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined.

Dijkstra's algorithm uses a data structure for storing and querying partial solutions sorted by distance from the start. While the original algorithm uses a min-priority queue and runs in time $\Theta((|V| + |E|)log|V|)$where $|V|$is the number of nodes and $|E|$ is the number of edges), it can also be implemented in$\Theta(|V|^2)$ using an array.

Algorithm

Let the node at which we are starting be called the initial node. Let the distance of node Y be the distance from the initial node to Y. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.

1. Mark all nodes unvisited. Create a set of all the unvisited nodes called the unvisited set.
2. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes. Set the initial node as current.
3. For the current node, consider all of its unvisited neighbours and calculate their tentative distances through the current node. Compare the newly calculated tentative distance to the current assigned value and assign the smaller one.
4. When we are done considering all of the unvisited neighbours of the current node, mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again.
5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the unvisited set is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.
6. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3.

When planning a route, it is actually not necessary to wait until the destination node is "visited" as above: the algorithm can stop once the destination node has the smallest tentative distance among all "unvisited" nodes (and thus could be selected as the next "current").

2.3.2 Bellman-Ford Algorithm

The Bellman–Ford algorithm is an algorithm that computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph. It is slower than Dijkstra's algorithm for the same problem, but more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers. The algorithm was first proposed by Alfonso Shimbel (1955), but is instead named after Richard Bellman and Lester Ford Jr., who published it in 1958 and 1956, respectively.

Negative edge weights are found in various applications of graphs, hence the usefulness of this algorithm. If a graph contains a "negative cycle" (i.e. a cycle whose edges sum to a negative value) that is reachable from the source, then there is no cheapest path: any path that has a point on the negative cycle can be made cheaper by one more walk around the negative cycle. In such a case, the Bellman–Ford algorithm can detect and report the negative cycle.

Algorithm

Like Dijkstra's algorithm, Bellman–Ford proceeds by relaxation, in which approximations to the correct distance are replaced by better ones until they eventually reach the solution. In both algorithms, the approximate distance to each vertex is always an overestimate of the true distance, and is replaced by the minimum of its old value and the length of a newly found path. However, Dijkstra's algorithm uses a priority queue to greedily select the closest vertex that has not yet been processed, and performs this relaxation process on all of its outgoing edges; by contrast, the Bellman–Ford algorithm simply relaxes all the edges, and does this |V|-1 times, where |V| is the number of vertices in the graph. In each of these repetitions, the number of vertices with correctly calculated distances grows, from which it follows that eventually all vertices will have their correct distances. This method allows the Bellman–Ford algorithm to be applied to a wider class of inputs than Dijkstra.

Bellman–Ford runs in O(|V|.|E|) time, where |V| and |E| are the number of vertices and edges respectively.

2.3.3 Johnson's Algorithm

Johnson's algorithm is a way to find the shortest paths between all pairs of vertices in an edge-weighted, directed graph. It allows some of the edge weights to be negative numbers, but no negative-weight cycles may exist. It works by using the Bellman–Ford algorithm to compute a transformation of the input graph that removes all negative weights, allowing Dijkstra's algorithm to be used on the transformed graph. It is named after Donald B. Johnson, who first published the technique in 1977.

Algorithm

Johnson's algorithm consists of the following steps:

1. First, a new node *q* is added to the graph, connected by zero-weight edges to each of the other nodes.
2. Second, the Bellman–Ford algorithm is used, starting from the new vertex *q*, to find for each vertex *v* the minimum weight *h(v)* of a path from *q* to *v*. If this step detects a negative cycle, the algorithm is terminated.
3. Next the edges of the original graph are reweighted using the values computed by the Bellman–Ford algorithm: an edge from *u* to *v*, having length *w(u, v)*, is given the new length *w(u, v)* + *h(u)* − *h(v)*.
4. Finally, *q* is removed, and Dijkstra's algorithm is used to find the shortest paths from each node *s* to every other vertex in the reweighted graph. The distance in the original graph is then computed for each distance *D(u , v)*, by adding *h(v)* − *h(u)* to the distance returned by Dijkstra's algorithm.

The time complexity of this algorithm, using Fibonacci heaps in the implementation of Dijkstra's algorithm, is $O(|V|^2 log|V| \ + \ |V|.|E|)$: the algorithm uses $O(|V|.|E|)$ time for the Bellman–Ford stage of the algorithm, and $O(|V|log|V| \ + \ |E|)$ for each of the $|V|$ instantiations of Dijkstra's algorithm.

2.3.4 Floyd-Warshall Algorithm

In computer science, the Floyd–Warshall algorithm (also known as Floyd's algorithm, the Roy–Warshall algorithm, the Roy–Floyd algorithm, or the WFI algorithm) is an algorithm for finding shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles). A single execution of the algorithm will find the lengths (summed weights) of shortest paths between all pairs of vertices. Although it does not return details of the paths themselves, it is possible to reconstruct the paths with simple modifications to the algorithm.

The Floyd–Warshall algorithm compares all possible paths through the graph between each pair of vertices. It is able to do this with $\Theta(|V|^3)$ comparisons in a graph, even though there may be up to $\Omega(|V|^2)$ edges in the graph, and every combination of edges is tested. It does so by incrementally improving an estimate on the shortest path between two vertices, until the estimate is optimal.

Algorithm

Consider a graph G with vertices V numbered 1 through N. Further consider a function *shortestPath(i,j,k)* that returns the shortest possible path from *i* to *j* using vertices only from the set {1, 2, …, k} as intermediate points along the way. Now, given this function, our goal is to find the shortest path from each *i* to each *j* using any vertex in {1, 2, …, N}.

For each of these pairs of vertices, the *shortestPath(i, j, k)* could be either

a path that does not go through *k* (only uses vertices in the set {1, 2, …, k-1}

Or

a path that does go through *k* (from *i* to *k* and then from *k* to *j*, both only using intermediate vertices in {1, 2, …, k-1}).

We know that the best path from *i* to *j* that only uses vertices *1* through *k-1* is defined by *shortestPath(i, j, k-1)*, and it is clear that if there was a better path from *i* to *k* to *j*, then the length of this path would be the concatenation of the shortest path from *i* to *k* (only using intermediate vertices in {1, 2, …, k-1}) and the shortest path from *k* to *j* (only using intermediate vertices in {1, 2, …, k-1}).

If *w(i,j)* is the weight of the edge between vertices *i* and *j*, we can define *shortestPath(i,j,k)* in terms of the following recursive formula:

the base case is
shortestPath(i,j,0) = w(i,j)

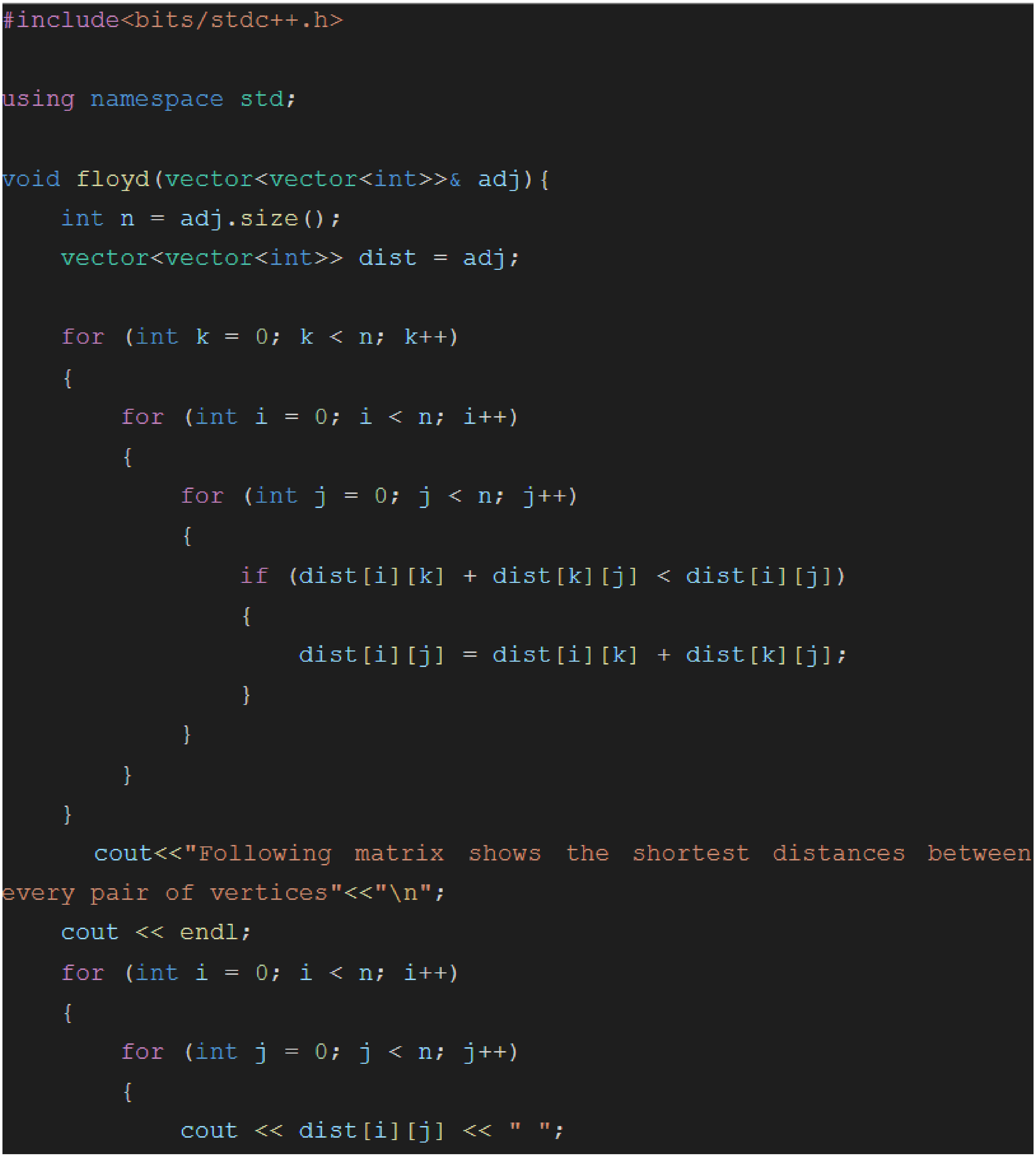
and the recursive case is
shortestPath(i,j,k)=min{shortestPath(i,j,k-1), shortestPath(i,k,k-1) + shortestPath(k,j,k-1)}.

This formula is the heart of the Floyd–Warshall algorithm. The algorithm works by first computing *shortestPath(i,j,k)* for all *(i,j)* pairs for *k*=1, then *k*=2, and so on. This process continues until *k*=N, and we have found the shortest path for all *(i,j)* pairs using any intermediate vertices.

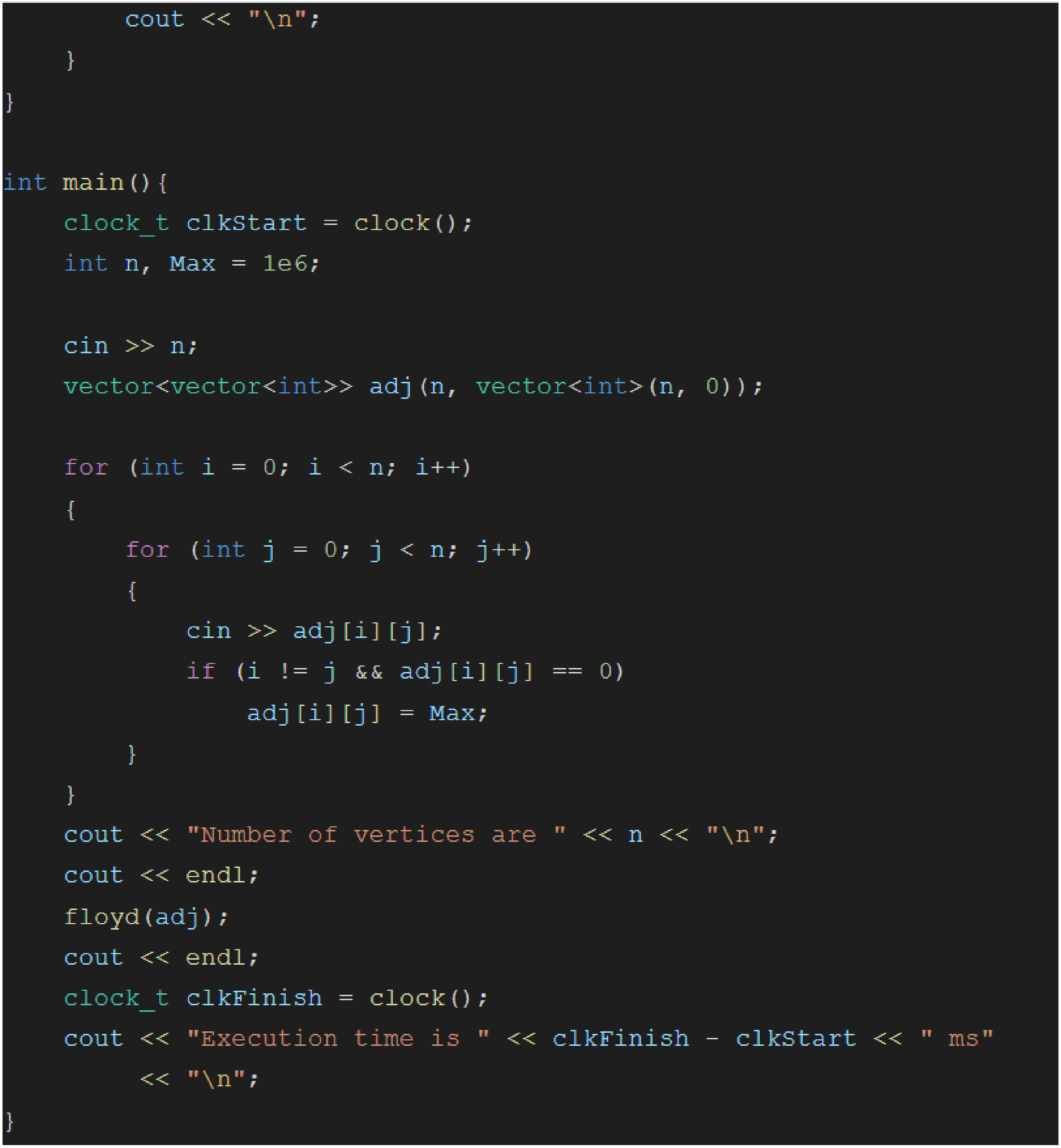
Let n be |V|, the number of vertices. To find all n^2 of *shortestPath(i, j, k)* (for all *i* and *j*) from those of *shortestPath(i, j, k-1)* requires $2n^2$ operations.
Since we begin with *shortestPath(i, j, 0) = edgeCost(i, j)* and compute the sequence of n matrices *shortestPath(i, j, 1), shortestPath(i, j, 2)..., shortestPath(i, j, n)*, the total number of operations used is $n.2n^2 = 2n^3$. Therefore, the complexity of the algorithm is $\Theta(n^3)$.

C++ Code Implementation

Floyd-Warshall Algorithm

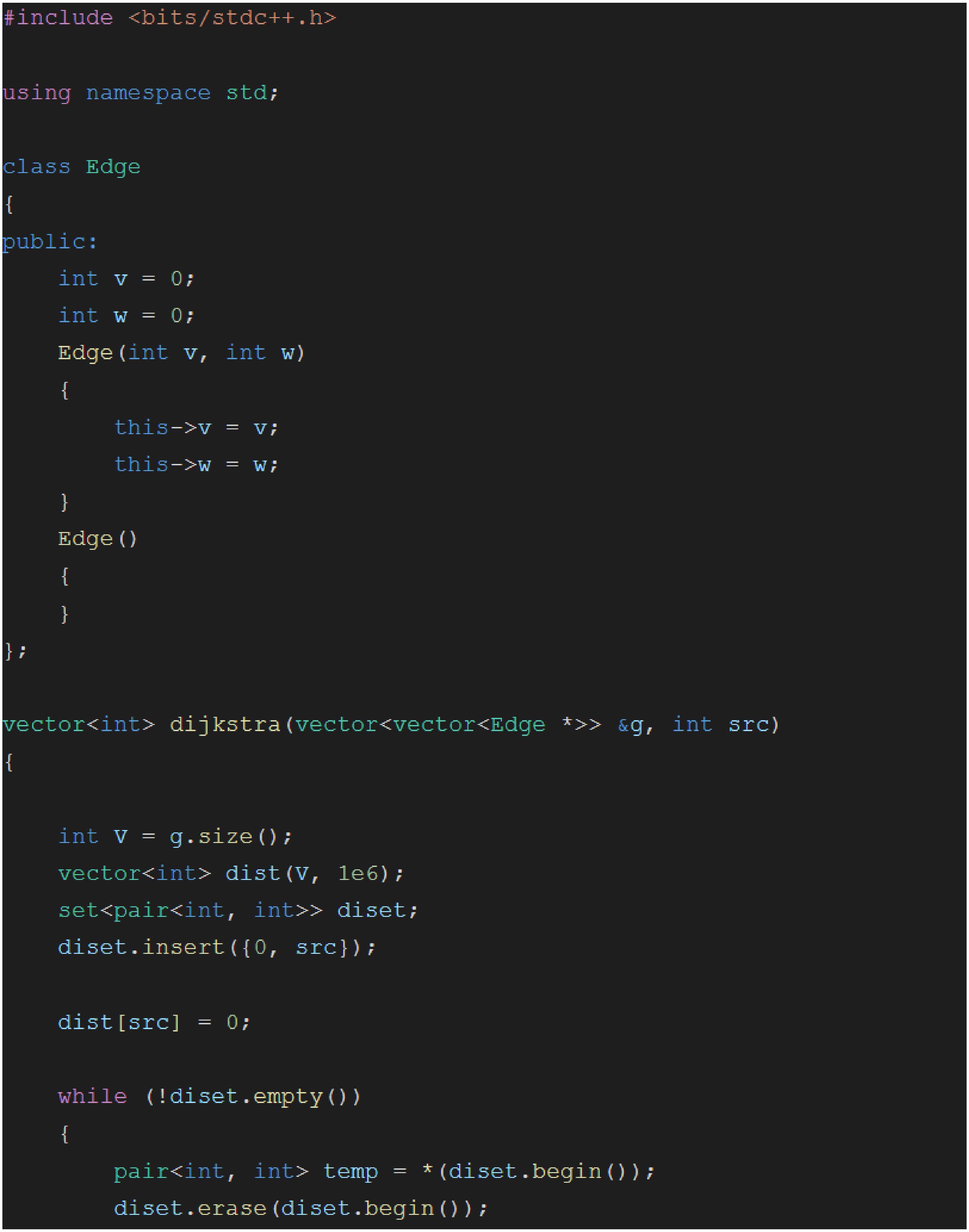


```
#include<bits/stdc++.h>

using namespace std;

void floyd(vector<vector<int>>& adj){
    int n = adj.size();
    vector<vector<int>> dist = adj;

    for (int k = 0; k < n; k++)
    {
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < n; j++)
            {
                if (dist[i][k] + dist[k][j] < dist[i][j])
                {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }
      cout<<"Following matrix shows the shortest distances between
every pair of vertices"<<"\n";
    cout << endl;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            cout << dist[i][j] << " ";
```

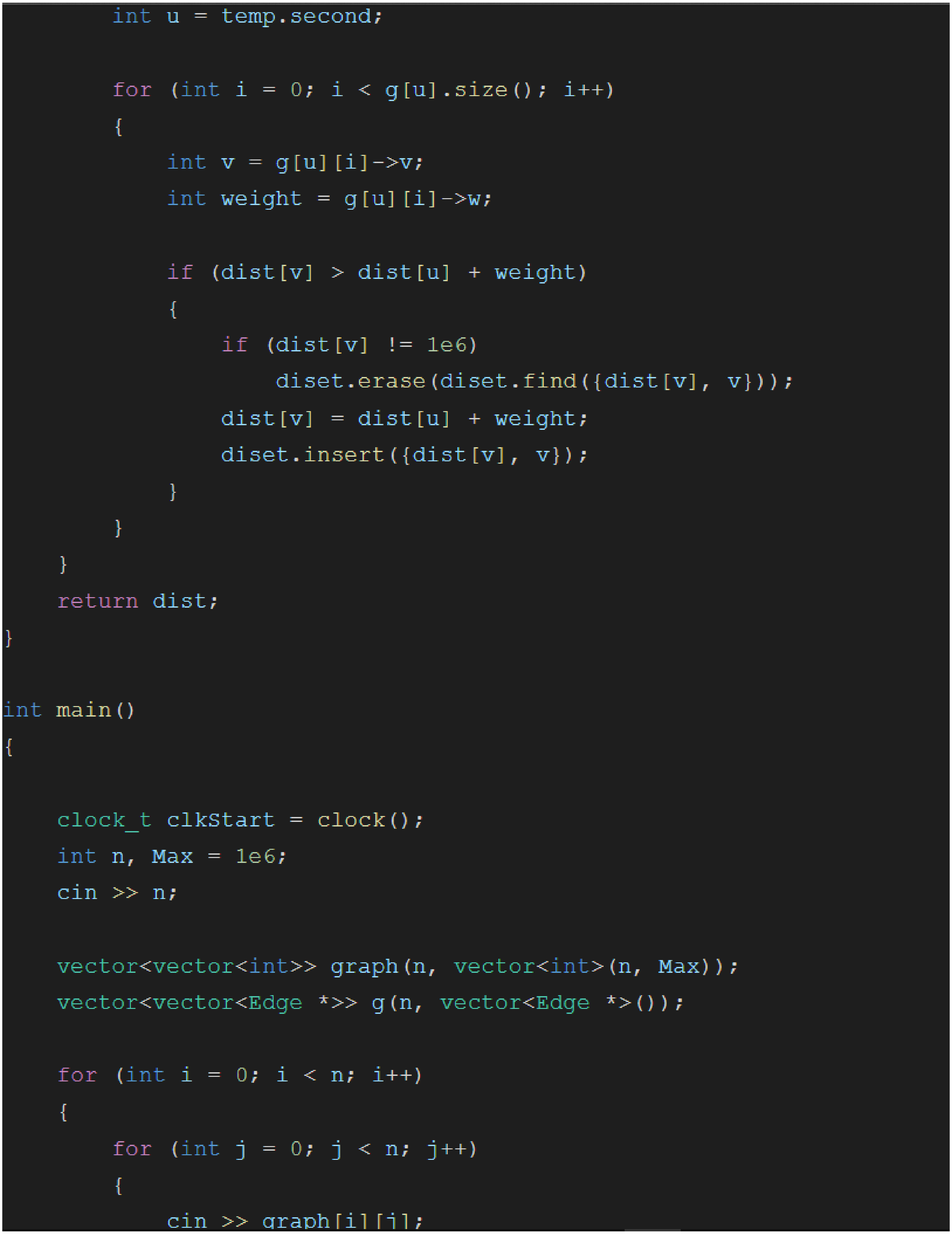


```
        cout << "\n";
    }
}

int main(){
    clock_t clkStart = clock();
    int n, Max = 1e6;

    cin >> n;
    vector<vector<int>> adj(n, vector<int>(n, 0));

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            cin >> adj[i][j];
            if (i != j && adj[i][j] == 0)
                adj[i][j] = Max;
        }
    }
    cout << "Number of vertices are " << n << "\n";
    cout << endl;
    floyd(adj);
    cout << endl;
    clock_t clkFinish = clock();
    cout << "Execution time is " << clkFinish - clkStart << " ms"
         << "\n";
}
```

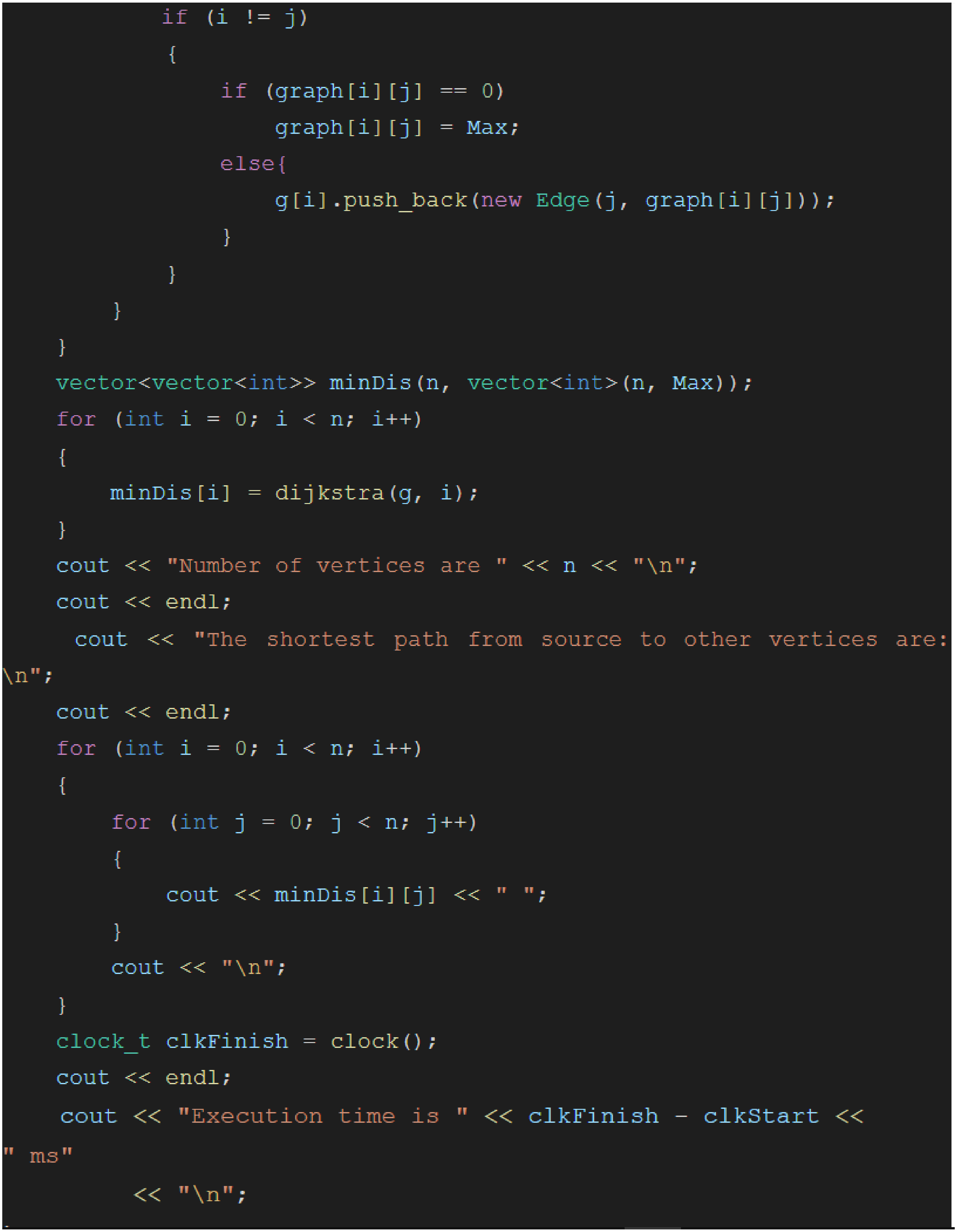
Johnson’s Algorithm



```
#include <bits/stdc++.h>

using namespace std;

class Edge
{
public:
    int v = 0;
    int w = 0;
    Edge(int v, int w)
    {
        this->v = v;
        this->w = w;
    }
    Edge()
    {
    }
};

vector<int> dijkstra(vector<vector<Edge *>> &g, int src)
{

    int V = g.size();
    vector<int> dist(V, 1e6);
    set<pair<int, int>> diset;
    diset.insert({0, src});

    dist[src] = 0;

    while (!diset.empty())
    {
        pair<int, int> temp = *(diset.begin());
        diset.erase(diset.begin());
```



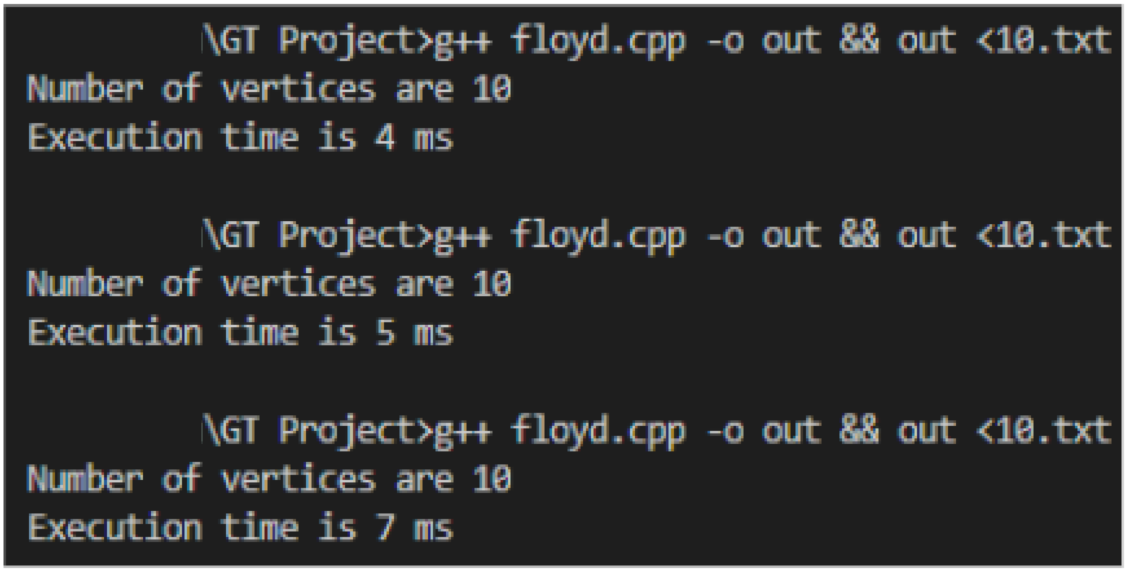
```
        int u = temp.second;

        for (int i = 0; i < g[u].size(); i++)
        {
            int v = g[u][i]->v;
            int weight = g[u][i]->w;

            if (dist[v] > dist[u] + weight)
            {
                if (dist[v] != 1e6)
                    diset.erase(diset.find({dist[v], v}));
                dist[v] = dist[u] + weight;
                diset.insert({dist[v], v});
            }
        }
    }
    return dist;
}

int main()
{

    clock_t clkStart = clock();
    int n, Max = 1e6;
    cin >> n;

    vector<vector<int>> graph(n, vector<int>(n, Max));
    vector<vector<Edge *>> g(n, vector<Edge *>());

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            cin >> graph[i][j];
```



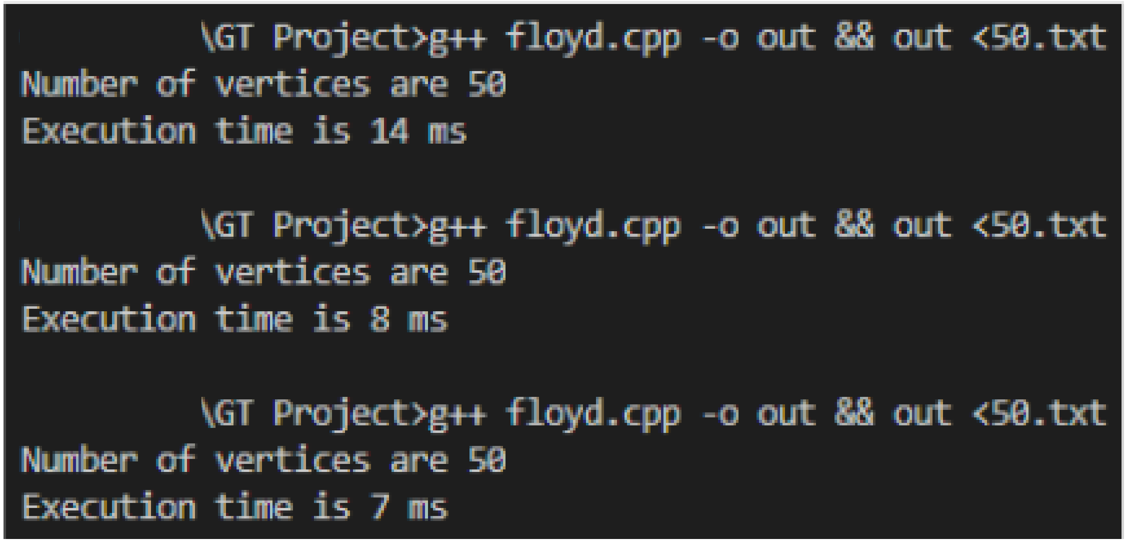
```
            if (i != j)
            {
                if (graph[i][j] == 0)
                    graph[i][j] = Max;
                else{
                    g[i].push_back(new Edge(j, graph[i][j]));
                }
            }
        }
    }
    vector<vector<int>> minDis(n, vector<int>(n, Max));
    for (int i = 0; i < n; i++)
    {
        minDis[i] = dijkstra(g, i);
    }
    cout << "Number of vertices are " << n << "\n";
    cout << endl;
     cout << "The shortest path from source to other vertices are:
\n";
    cout << endl;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            cout << minDis[i][j] << " ";
        }
        cout << "\n";
    }
    clock_t clkFinish = clock();
    cout << endl;
    cout << "Execution time is " << clkFinish - clkStart <<
" ms"
          << "\n";
```

Results

Floyd Warshall's Algorithm on Dense Network

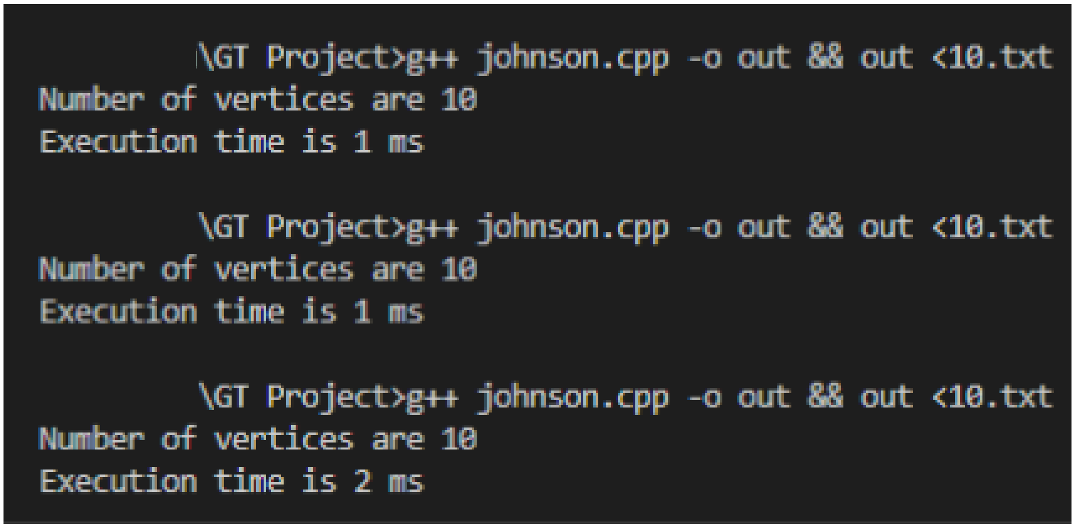


```
         \GT Project>g++ floyd.cpp -o out && out <10.txt
Number of vertices are 10
Execution time is 4 ms

         \GT Project>g++ floyd.cpp -o out && out <10.txt
Number of vertices are 10
Execution time is 5 ms

         \GT Project>g++ floyd.cpp -o out && out <10.txt
Number of vertices are 10
Execution time is 7 ms
```

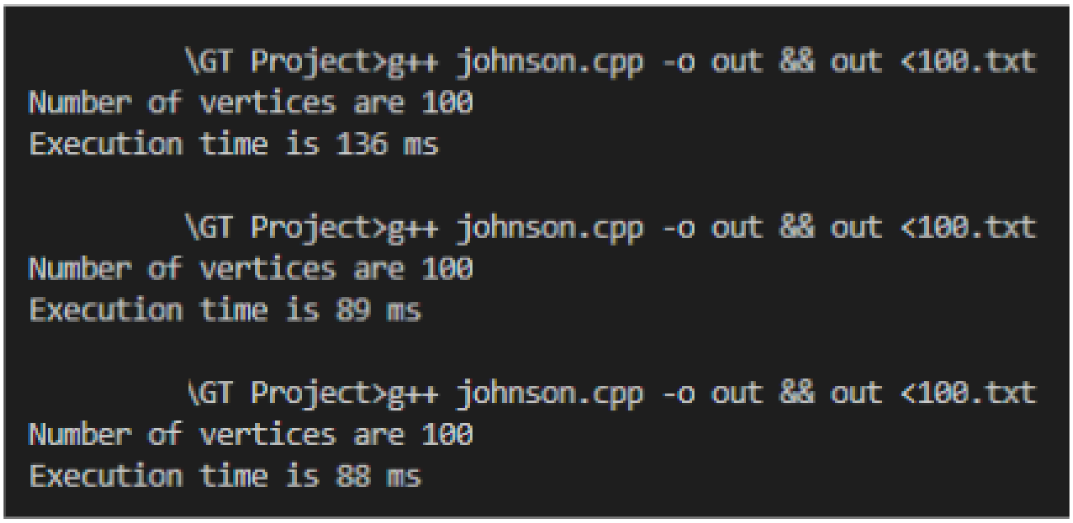


```
         \GT Project>g++ floyd.cpp -o out && out <50.txt
Number of vertices are 50
Execution time is 14 ms

         \GT Project>g++ floyd.cpp -o out && out <50.txt
Number of vertices are 50
Execution time is 8 ms

         \GT Project>g++ floyd.cpp -o out && out <50.txt
Number of vertices are 50
Execution time is 7 ms
```

Johnson’s Algorithm on Dense Network

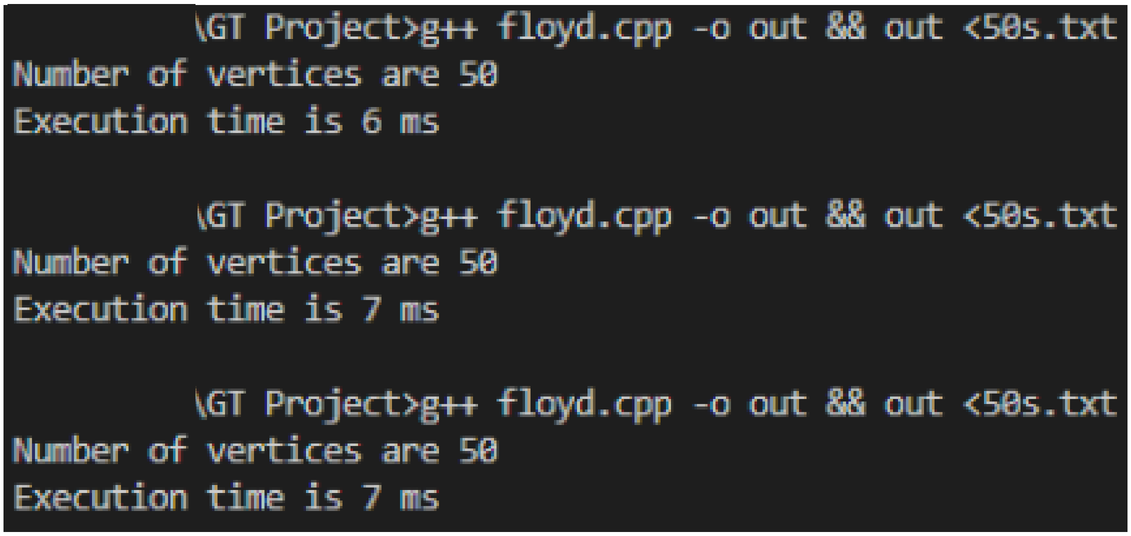


```
         \GT Project>g++ johnson.cpp -o out && out <10.txt
Number of vertices are 10
Execution time is 1 ms

         \GT Project>g++ johnson.cpp -o out && out <10.txt
Number of vertices are 10
Execution time is 1 ms

         \GT Project>g++ johnson.cpp -o out && out <10.txt
Number of vertices are 10
Execution time is 2 ms
```

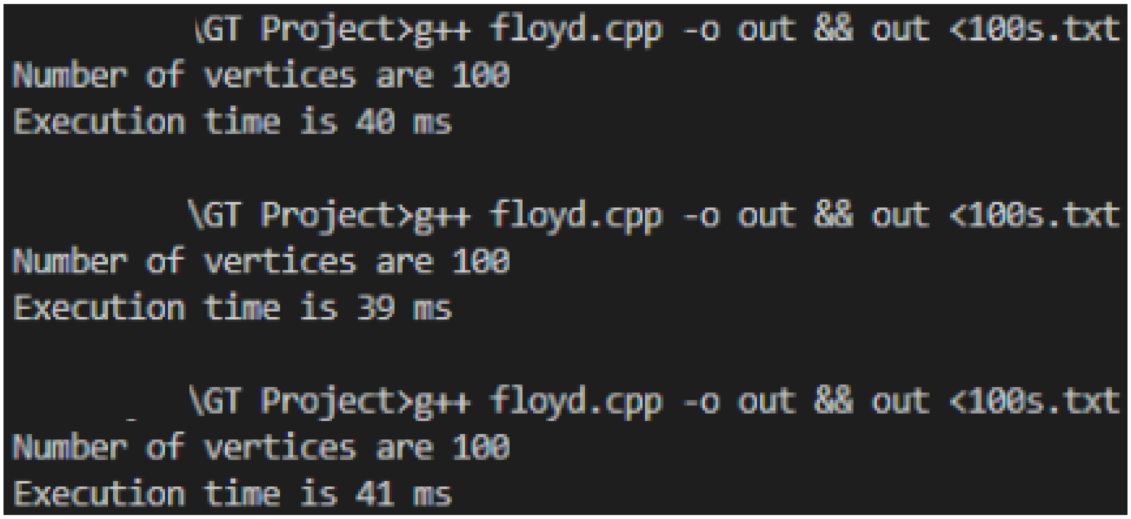


```
         \GT Project>g++ johnson.cpp -o out && out <100.txt
Number of vertices are 100
Execution time is 136 ms

         \GT Project>g++ johnson.cpp -o out && out <100.txt
Number of vertices are 100
Execution time is 89 ms

         \GT Project>g++ johnson.cpp -o out && out <100.txt
Number of vertices are 100
Execution time is 88 ms
```

Floyd Warshall’s Algorithm on Sparse Network

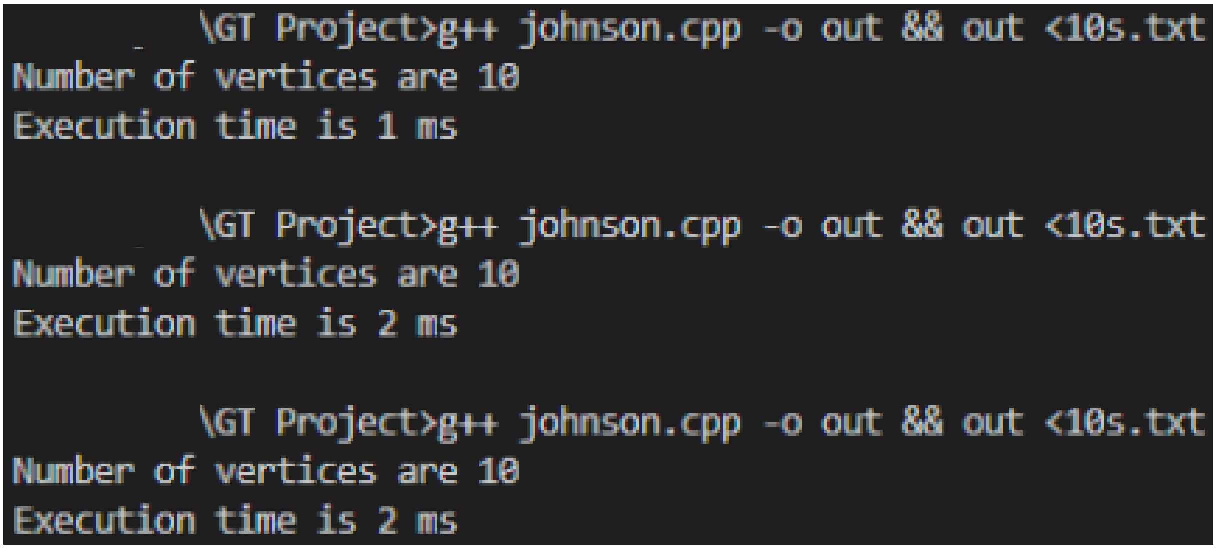


```
[redacted]\GT Project>g++ floyd.cpp -o out && out <50s.txt
Number of vertices are 50
Execution time is 6 ms

[redacted]\GT Project>g++ floyd.cpp -o out && out <50s.txt
Number of vertices are 50
Execution time is 7 ms

[redacted]\GT Project>g++ floyd.cpp -o out && out <50s.txt
Number of vertices are 50
Execution time is 7 ms
```

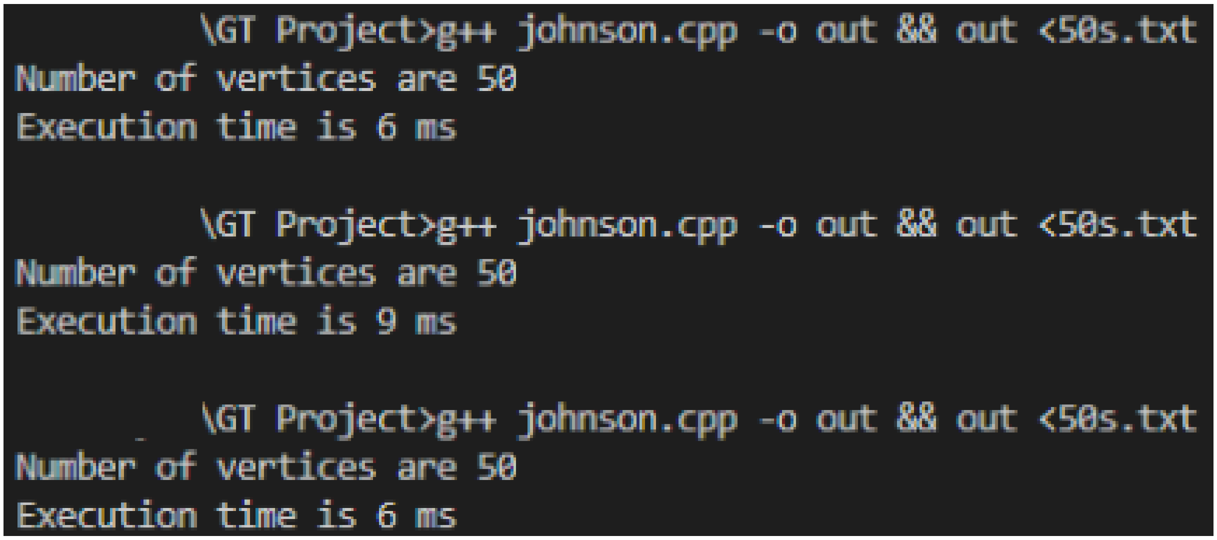


```
[redacted]\GT Project>g++ floyd.cpp -o out && out <100s.txt
Number of vertices are 100
Execution time is 40 ms

[redacted]\GT Project>g++ floyd.cpp -o out && out <100s.txt
Number of vertices are 100
Execution time is 39 ms

[redacted]\GT Project>g++ floyd.cpp -o out && out <100s.txt
Number of vertices are 100
Execution time is 41 ms
```

Johnson’s Algorithm on Sparse Network



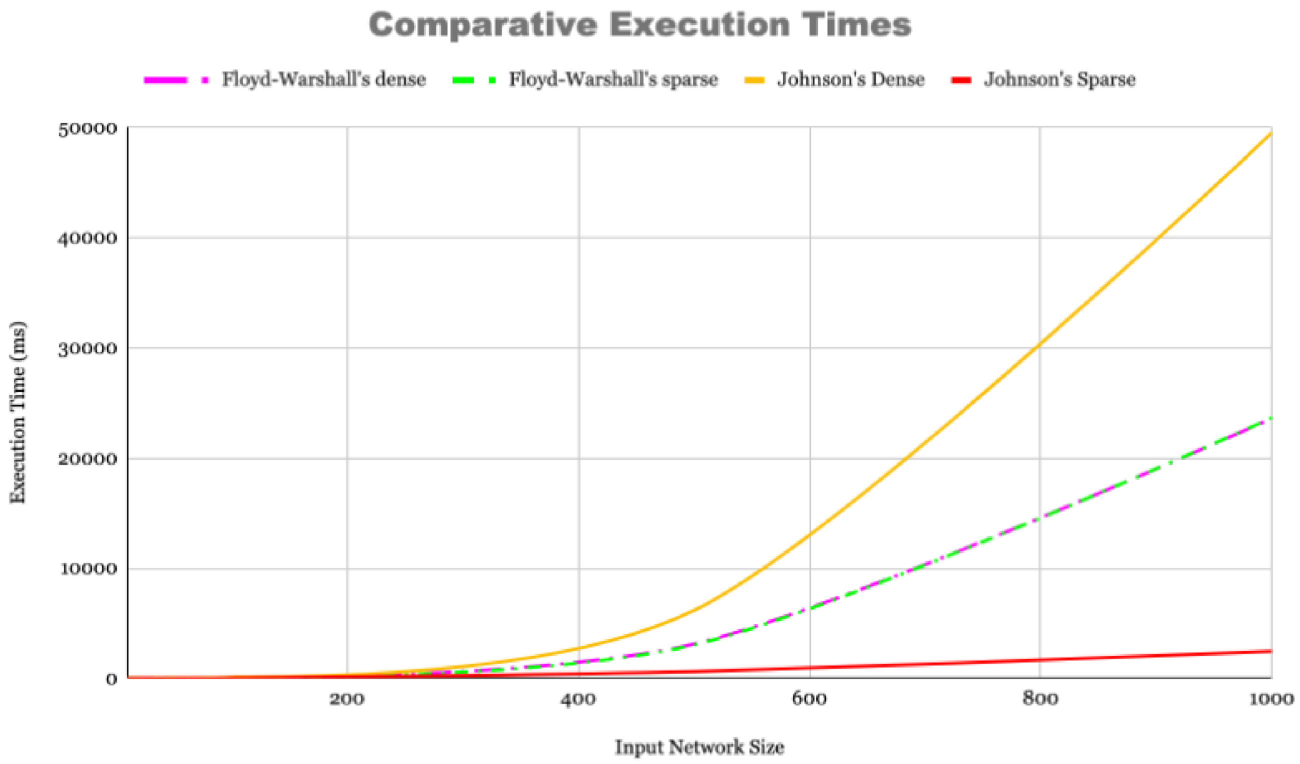
```
         \GT Project>g++ johnson.cpp -o out && out <10s.txt
Number of vertices are 10
Execution time is 1 ms

         \GT Project>g++ johnson.cpp -o out && out <10s.txt
Number of vertices are 10
Execution time is 2 ms

         \GT Project>g++ johnson.cpp -o out && out <10s.txt
Number of vertices are 10
Execution time is 2 ms
```



```
         \GT Project>g++ johnson.cpp -o out && out <50s.txt
Number of vertices are 50
Execution time is 6 ms

         \GT Project>g++ johnson.cpp -o out && out <50s.txt
Number of vertices are 50
Execution time is 9 ms

         \GT Project>g++ johnson.cpp -o out && out <50s.txt
Number of vertices are 50
Execution time is 6 ms
```

Conclusions

We experimentally observe the variation of execution times of Floyd Warshall and Johnson's algorithms on dense and sparse networks. The observation may be represented as follows:



We observe almost no variation in execution time of Floyd Warshall algorithm as the number of edges in the graph/ connections in the network change, which is in accordance with the theoretical complexity of order of (number of vertices)^3. The execution time of Johnson's algorithm, however, varies greatly as the number of edges in the network/ connections in the graph changes, verifying the theoretical dependence on the number of edges.

We safely conclude, on the basis of above data, that Johnson's algorithm performs more efficiently on sparse networks, and less efficiently on dense networks when compared to Floyd Warshall algorithm, whose execution does not vary.

Floyd Warshall algorithm provides an ideal choice to be used for routing in dense networks, being asymptotically faster than Johnson's algorithm.

Johnson's algorithm is ideal for use in sparse networks, being asymptotically faster than Floyd Warshall algorithm.
]

References

- https://ieeexplore.ieee.org/document/6770643/
- https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-263j-data-communication-networks-fall-2002/lecture-notes/Lecture21.pdf
- https://www.universiteitleiden.nl/binaries/content/assets/science/mi/scripties/master/2017-2018/optimalroutingalgorithms-jurrevanderlaan-171124-29nov2017.pdf
- https://en.wikipedia.org/wiki/Computer_network
- https://en.wikipedia.org/wiki/Routing
- https://www.geeksforgeeks.org/types-of-routing/
- https://www.tutorialspoint.com/difference-between-static-routing-and-dynamic-routing#:~:text=Static%20Routing%20or%20Non%2DAdaptive,more%20security%20than%20dynamic%20routing
- https://orhanergun.net/what-is-optimal-routing-and-suboptimal-routing-in-networking/
- https://en.wikipedia.org/wiki/Static_routing
- https://en.wikipedia.org/wiki/Dijkstra's_algorithm#Algorithm
- https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm#Algorithm
- https://en.wikipedia.org/wiki/Johnson's_algorithm
- https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm#Analysis