University of Dhaka

Department of Computer Science and Engineering

CSE-3111 : Computer Networking Lab

Lab Experiment 7 : Implementation of Link State Algorithm

Submitted By:

Name: Muztoba Hasan Sinha

Roll No : 15

Name: Bholanath Das Niloy

Roll No : 22

Submitted On :

March 25, 2023

Submitted To :

Dr. Md. Abdur Razzaque

Md Mahmudur Rahman

Md. Ashraful Islam

Md. Fahim Arefin

Contents

1 Introduction

We were tasked with implementing multiple servers for simulating the Link-State Algorithm. This was to improve our understanding of the link-state algorithm and further develop the practical understanding required to use this algorithm in distributed systems.

2 Theory

2.1 Overview

Link-state routing protocols are one of the two main classes of routing protocols used in packet switching networks for computer communications, the others being distance-vector routing protocols. The link-state protocol is performed by every switching node in the network (i.e., nodes that are prepared to forward packets; in the Internet, these are called routers). The basic concept of link-state routing is that every node constructs a map of the connectivity to the network, in the form of a graph, showing which nodes are connected to which other nodes. Each node then independently calculates the next best logical path from it to every possible destination in the network. Each collection of best paths will then form each node's routing table.

In link-state routing protocols, each router possesses information about the complete network topology. Each router then independently calculates the best next hop from it for every possible destination in the network using local information of the topology. The collection of best-next-hops forms the routing table.

2.2 Distributing the Maps to all the nodes

The first main stage in the link-state algorithm is to give a map of the network to every node. This is done with several subsidiary steps.

2.2.1 Determining the neighbours of each node

First, each node needs to determine what other ports it is connected to, over fully working links; it does this using reachability protocol it runs periodically and separately with each of its directly connected neighbours.

2.2.2 Distributing the information for the map

Next, each node periodically (and in case of connectivity changes) sends a short message, the link-state advertisement, which:

1. Identifies the node which is producing it.
2. Identifies all the other nodes (either routers or networks) to which it is directly connected.
3. Includes a 'sequence number', which increases every time the source node makes up a new version of the message.

This message is sent to all the nodes on a network. As a necessary precursor, each node in the network remembers, for every one of its neighbors, the sequence number of the last link-state message which it received from that node. When a link-state advertisement is received at a node, the node looks up the sequence number it has stored for the source of that link-state message: if this message is newer (i.e., has a higher sequence number), it is saved, the sequence number is updated, and a copy is sent in turn to each of that node's neighbors. This procedure rapidly gets a copy of the latest version of each node's link-state advertisement to every node in the network.

2.2.3 Creating the map

Finally, with the complete set of link-state advertisements (one from each node in the network) in hand, each node produces the graph for the map of the network. The algorithm iterates over the collection of link-state advertisements; for each one, it makes links on the map of the network, from the node which sent that message, to all the nodes which that message indicates are neighbors of the sending node.

No link is considered to have been correctly reported unless the two ends agree; i.e., if one node reports that it is connected to another, but the other node does not report that it is connected to the first, there is a problem, and the link is not included on the map.

2.3 Calculating the routing table

The second main stage in the link-state algorithm is to produce routing tables, by inspecting the maps. This is again done with several steps.

2.3.1 Calculating the shortest paths

Each node independently runs an algorithm over the map to determine the shortest path from itself to every other node in the network; generally some variant of Dijkstra's algorithm is used. This is based around a link cost across each path which includes available bandwidth among other things.

A node maintains two data structures: a tree containing nodes which are "done", and a list of candidates. The algorithm starts with both structures empty; it then adds to the first one the node itself. The variant of a greedy algorithm then repetitively does the following:

1. All neighbour nodes which are directly connected to the node are just added to the tree (excepting any nodes which are already in either the tree or the candidate list). The rest are added to the second (candidate) list.
2. Each node in the candidate list is compared to each of the nodes already in the tree. The candidate node which is closest to any of the nodes already in the tree is itself moved into the tree and attached to the appropriate neighbor node. When a node is moved from the candidate list into the tree, it is removed from the candidate list and is not considered in subsequent iterations of the algorithm.

The above two steps are repeated as long as there are any nodes left in the candidate list. (When there are none, all the nodes in the network will have been added to the tree.) This procedure ends with the tree containing all the nodes in the network, with the node on which the algorithm is running as the root of the tree. The shortest path from that node to any other node is indicated by the list of nodes one traverses to get from the root of the tree, to the desired node in the tree.

2.3.2 Filling the routing table

With the shortest paths in hand, the next step is to fill in the routing table. For any given destination node, the best path for that destination is the node which is the first step from the root node, down the branch in the shortest-path tree which leads toward the desired destination node. To create the routing table, it is only necessary to walk the tree, remembering the identity of the node at the head of each branch, and filling in the routing table entry for each node one comes across with that identity.

2.4 Dijkstra's Algorithm

Algorithm 1: Dijkstra's Algorithm

```
Input: Graph $G = (V, E)$, source vertex $s$
Result: Shortest path from $s$ to all other vertices
for each vertex $v \in V$ do
    $dist[v] \leftarrow \infty$;
    $visited[v] \leftarrow$ false;
$dist[s] \leftarrow 0$;
while there are unvisited vertices do
    $u \leftarrow$ vertex in $V$ with smallest $dist[u]$ value;
    $visited[u] \leftarrow$ true;
    for each neighbor $v$ of $u$ do
        $alt \leftarrow dist[u] +$ weight of edge $(u, v)$;
        if $alt < dist[v]$ then
            $dist[v] \leftarrow alt$;
```

3 Performance Metrics

3.1 Time Complexity Analysis

The Link State algorithm is an algorithm used for computing the shortest path in a network graph. The time complexity of the algorithm can be broken down into two parts: the initialization phase and the shortest path calculation phase.

3.1.1 Initialization Phase

During the initialization phase, the algorithm creates a list of all nodes in the network and their corresponding edges. The time complexity of this phase is O(V^2), where V is the number of nodes in the network. This is because for each node in the network, the algorithm needs to examine all other nodes in the network to determine if there is a direct connection between them.

3.1.2 Shortest Path Calculation Phase

During the shortest path calculation phase, the algorithm uses Dijkstra's algorithm to find the shortest path from the source node to all other nodes in the network. The time complexity of Dijkstra's algorithm is $\mathrm{O}(V^2)$ if a simple array is used to represent the set of unexplored nodes, or $\mathrm{O}(V \log V)$ if a priority queue is used. However, the Link State algorithm also requires the computation and distribution of link state updates, which may be more costly than the path calculation itself.

Assuming that the number of link state updates is proportional to the number of edges in the network, the time complexity of the Link State algorithm can be expressed as $\mathrm{O}(V^2 + E \log V)$, where E is the number of edges in the network. This is because the initialization phase takes $\mathrm{O}(V^2)$ time, while the shortest path calculation phase takes $\mathrm{O}(E \log V)$ time with a priority queue implementation of Dijkstra's algorithm.

3.2 Memory Complexity Analysis

The memory complexity of the link state algorithm depends on the size of the network and the amount of information that needs to be stored in each node.Assuming there are V nodes in the network, the memory complexity of the link state algorithm can be analyzed as follows:

1. Each node needs to store information about all other nodes in the network, including their addresses and link state information. This requires $V \cdot (V-1)$ entries in the database.
2. Each entry in the database requires a fixed amount of memory to store the node address and link state information. Let's assume this is m bytes per entry.
3. Therefore, the total memory required to store the network topology database is $V \cdot (V-1) \cdot m$ bytes.
4. In addition to the network topology database, each node also needs to maintain a list of its neighbors and the cost of the links to them. This requires $2 \cdot m$ bytes per neighbor.
5. Therefore, the total memory required for each node is $(V-1) \cdot 2 \cdot m + V \cdot (V-1) \cdot m$ bytes.

Overall, the memory complexity of the link state algorithm is $O(V^2 \cdot m)$ where V is the number of nodes in the network and m is the fixed amount of memory required to store each entry in the database. This means that as the network grows in size, the memory required to store the network topology database grows quadratically with respect to the number of nodes in the network.

4 Finding APSP

4.1 Graph

This is the graph we decided on as our network topology. Each node represents a router on the network.

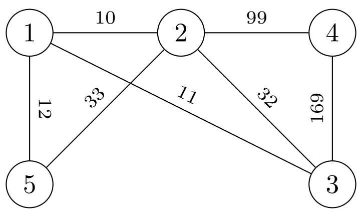


Figure 1: Network Topology

4.2 APSP

These are the results for after running the algorithm on the given graph as shown above.

Table 1: Dijkstra's Algorithm Results for all nodes

Source	Destination	Shortest Path	Path Length
1	1	$\{1\}$	0
	2	$\{1,2\}$	10
	3	$\{1,3\}$	11
	4	$\{1,2,4\}$	109
	5	$\{1,5\}$	12
2	1	$\{2,1\}$	10
	2	$\{2\}$	0
	3	$\{2,3\}$	32
	4	$\{2,4\}$	99
	5	$\{2,5\}$	33
3	1	$\{3,1\}$	11
	2	$\{3,1,2\}$	21
	3	$\{3\}$	0
	4	$\{3,1,2,4\}$	120
	5	$\{3,1,5\}$	23
4	1	$\{4,2,1\}$	109
	2	$\{4,2\}$	99
	3	$\{4,2,1,3\}$	120
	4	$\{4\}$	0
	5	$\{4,2,1,5\}$	121
5	1	$\{5,1\}$	12
	2	$\{5,1,2\}$	22
	3	$\{5,1,3\}$	23
	4	$\{5,1,2,4\}$	121
	5	$\{5\}$	0

4.3 Forwarding table

The forwarding table simulated for this graph by our algorithm was found to be

Table 2: Forwarding table for all nodes

Node	Destination	Next Hop	Path Length
1	1	*none*	0
	2	2	10
	3	3	11
	4	2	109
	5	5	12
2	1	1	10
	2	*none*	0
	3	3	32
	4	4	99
	5	5	33
3	1	1	11
	2	1	21
	3	*none*	0
	4	1	120
	5	1	23
4	1	2	109
	2	2	99
	3	2	120
	4	*none*	0
	5	2	121
5	1	1	12
	2	1	22
	3	1	23
	4	1	121
	5	*none*	0

5 Implementation

5.1 Considerations for implementation

We have considered a few restrictions for implementation. Which are listed as follows

1. Links among nodes/vertices do no break.
2. The graph is not a multigraph. That is two nodes are not directly connected by more than one edge/link.
3. The graph is undirected. Makes sense since connections are duplex.

These are for ease of implementation all of these cases could be handled trivially, for example we could simply delete a node and send a message of deletion to handle link breaks, multigraphs while handlable would not make sense since Djikstra's Algorithm will only route through the least cost path. And the last consideration is inherent to computer networks thus has to be handled.

5.1.1 Modelling the Network and Eastablishing Connection

We have implemented identical peer nodes. These open a TCP server to receive data and open a TCP connection when sending data. Since we have worked on a local machine, we have differentiated the nodes by port numbers. This functionality can be trivially extended by using an IP-Port tuple instead of the port. Each node also has the information of its links saved on an adjacency list.

This mapping of graph nodes in the figure drawn above to ports is given below

Table 3: Mapping of nodes in figure to ports

Node	Port
1	5001
2	5002
3	5003
4	5004
5	5005

5.2 Implementation of the Code Itself

Below we have described the major functions of the implementation.

5.2.1 Dijkstra's Algorithm Implementation

We have simply implemented the algorithm discussed above as per discussion. We can build a forwarding table in this function by simply noting every update to the distance and updating the new path from the node.

5.2.2 send

This function is used to build and send a link state packet from the local graph of the node. It has 2 distinct parts. Firstly it builds the link state packet. And then it finds all nodes that are connected to the node and opens a TCP connection. It sends one packet and immediately closes the connection. This way all of the connected edges are ensured to get a packet.

5.2.3 recvt

This function opens a TCP server with the localhost address and the port address of the node. then it waits for TCP connections. For each accepted connection it opens a thread with the receive function passing the connection objects where the data is received and parsed and then the thread is closed. This function is crucial to handle multiple packets from all over the network.

5.2.4 receive

This function accepts a connection and address object. It uses the connection object to receive the Link State Packet. It parses the LSP and splits them into edges and costs and passes them to the *update_edge_cost* function.

5.2.5 update_edge_cost

This function is passed every edge from then link state packet and it searches for changes in values of edges or new edges entirely. If found it either updates or extends the graph and calls Dijkstra's algorithm to minimize the cost. Else it simply passes. This minimizes unnecessary calls to the Dijkstra's Algorithm.

5.2.6 Program Initialization and Introduction of Link Cost Changes

We open a thread with the *recvt* function to handle incoming messages. Then we put a slight delay and open a thread to start transmitting LSP. While these run, We keep track of a timer and after a specific interval we randomly update a link cost. This results in changes in the adjacency list and subsequently changes to all of the nodes forwarding information.

5.3 Other consideration

Other extensions to the code include -

5.3.1 TTL

Each message fitted with a Time to Live by hop. For each forward we simply deduct the ttl and stop forwarding when TTL is less than or equal to zero. This introduces the problem of an outdated packet being looped back and causing an unwanted update, since our algorithm checks for changes even worse and updates it. This is mitigated by the next consideration.

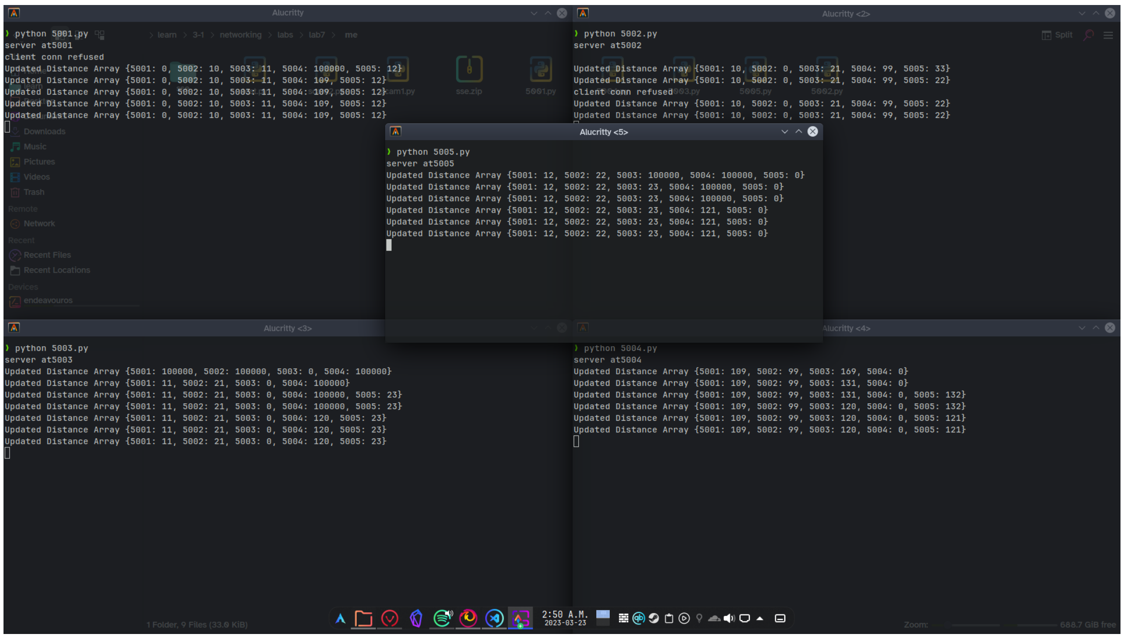
5.3.2 Packet ID

We generate a packet ID using the originating node IP and/or port as well as a sequence number. Then at each node after receiving a packet we add it to a map and process the packet. If the same packet is received at the node again. We simply consult the map and discard the packet. Thus eliminating any chance of a untimely update.

6 Screenshots of working code

6.1 Link State Routing with static link costs

Here we have a run of the code with the link cost changes disabled. Please note how the costs are correct for the given graph and how the implementation avoids unnecessary updates after the state has settled.



```
❯ python 5001.py
server at5001
client conn refused
Updated Distance Array {5001: 0, 5002: 10, 5003: 11, 5004: 100000, 5005: 12}
Updated Distance Array {5001: 0, 5002: 10, 5003: 11, 5004: 109, 5005: 12}
Updated Distance Array {5001: 0, 5002: 10, 5003: 11, 5004: 109, 5005: 12}
Updated Distance Array {5001: 0, 5002: 10, 5003: 11, 5004: 109, 5005: 12}
Updated Distance Array {5001: 0, 5002: 10, 5003: 11, 5004: 109, 5005: 12}
```

```
❯ python 5002.py
server at5002

Updated Distance Array {5001: 10, 5002: 0, 5003: 21, 5004: 99, 5005: 33}
Updated Distance Array {5001: 10, 5002: 0, 5003: 21, 5004: 99, 5005: 22}
client conn refused
Updated Distance Array {5001: 10, 5002: 0, 5003: 21, 5004: 99, 5005: 22}
Updated Distance Array {5001: 10, 5002: 0, 5003: 21, 5004: 99, 5005: 22}
```

```
❯ python 5005.py
server at5005
Updated Distance Array {5001: 12, 5002: 22, 5003: 100000, 5004: 100000, 5005: 0}
Updated Distance Array {5001: 12, 5002: 22, 5003: 23, 5004: 100000, 5005: 0}
Updated Distance Array {5001: 12, 5002: 22, 5003: 23, 5004: 100000, 5005: 0}
Updated Distance Array {5001: 12, 5002: 22, 5003: 23, 5004: 121, 5005: 0}
Updated Distance Array {5001: 12, 5002: 22, 5003: 23, 5004: 121, 5005: 0}
Updated Distance Array {5001: 12, 5002: 22, 5003: 23, 5004: 121, 5005: 0}
```

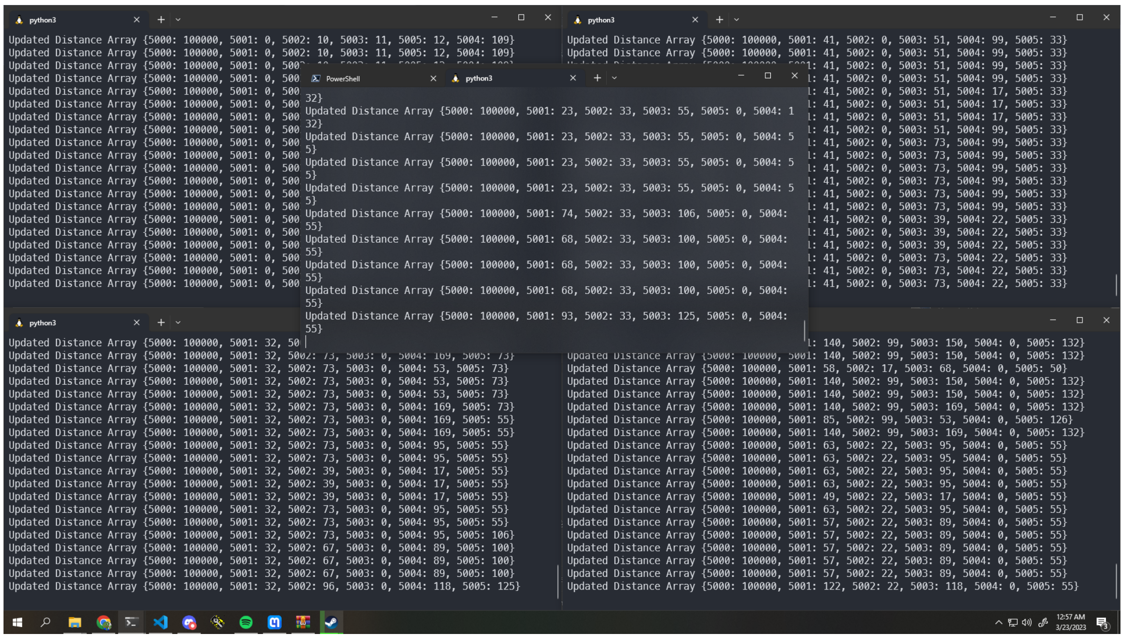
```
❯ python 5003.py
server at5003
Updated Distance Array {5001: 100000, 5002: 100000, 5003: 0, 5004: 100000}
Updated Distance Array {5001: 11, 5002: 21, 5003: 0, 5004: 100000}
Updated Distance Array {5001: 11, 5002: 21, 5003: 0, 5004: 100000, 5005: 23}
Updated Distance Array {5001: 11, 5002: 21, 5003: 0, 5004: 100000, 5005: 23}
Updated Distance Array {5001: 11, 5002: 21, 5003: 0, 5004: 120, 5005: 23}
Updated Distance Array {5001: 11, 5002: 21, 5003: 0, 5004: 120, 5005: 23}
Updated Distance Array {5001: 11, 5002: 21, 5003: 0, 5004: 120, 5005: 23}
```

```
❯ python 5004.py
server at5004
Updated Distance Array {5001: 109, 5002: 99, 5003: 169, 5004: 0}
Updated Distance Array {5001: 109, 5002: 99, 5003: 131, 5004: 0}
Updated Distance Array {5001: 109, 5002: 99, 5003: 131, 5004: 0, 5005: 132}
Updated Distance Array {5001: 109, 5002: 99, 5003: 120, 5004: 0, 5005: 132}
Updated Distance Array {5001: 109, 5002: 99, 5003: 120, 5004: 0, 5005: 121}
Updated Distance Array {5001: 109, 5002: 99, 5003: 120, 5004: 0, 5005: 121}
```

Figure 2: Link State Routing for the static graph described

6.2 Link State Routing with Varying link costs

Here we have varied the link costs over time randomly. Note how the algorithm is dynamically updating the costs when changes occur.



```
Updated Distance Array {5000: 100000, 5001: 0, 5002: 10, 5003: 11, 5005: 12, 5004: 109}
Updated Distance Array {5000: 100000, 5001: 0, 5002: 10, 5003: 11, 5005: 12, 5004: 109}
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
```

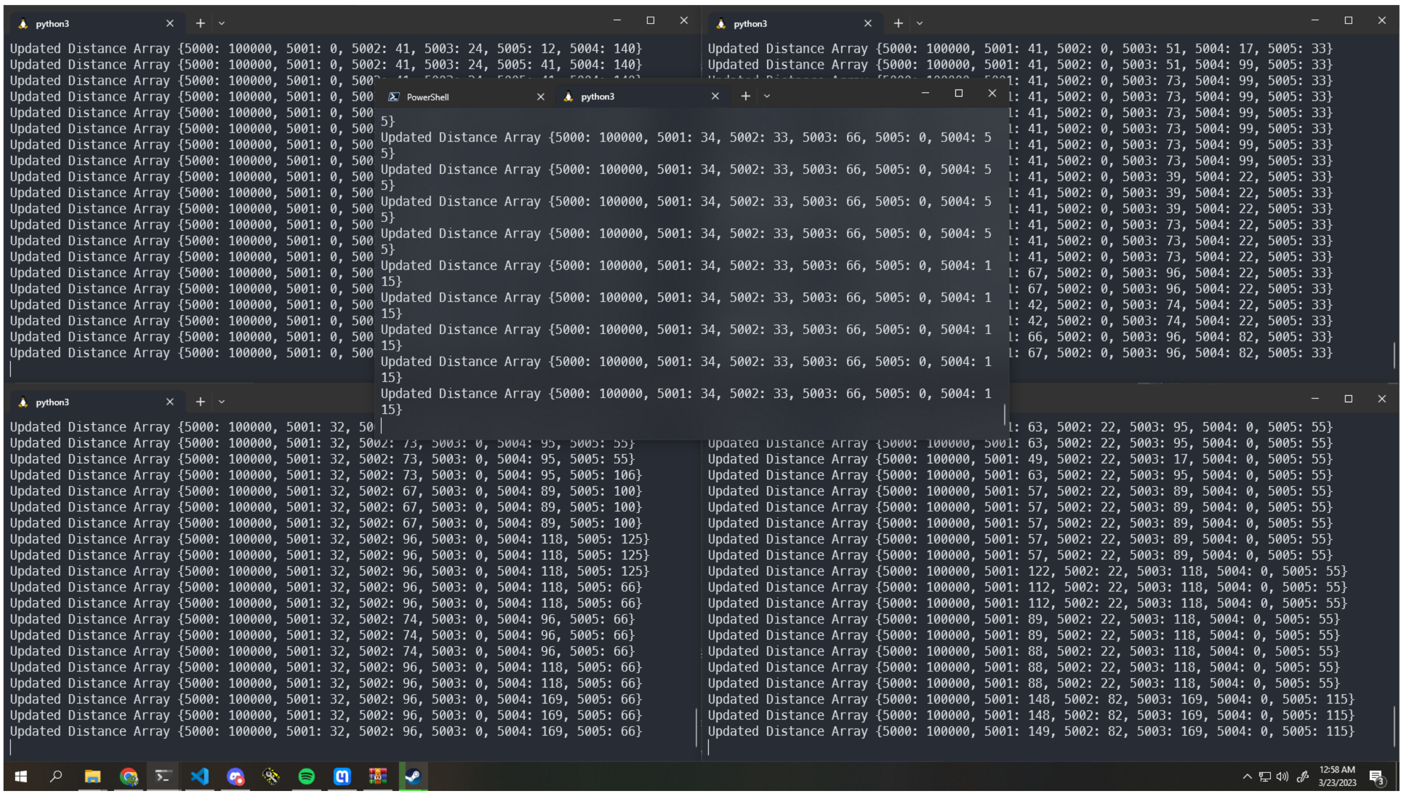
```
Updated Distance Array {5000: 100000, 5001: 41, 5002: 0, 5003: 51, 5004: 99, 5005: 33}
Updated Distance Array {5000: 100000, 5001: 41, 5002: 0, 5003: 51, 5004: 99, 5005: 33}
1: 41, 5002: 0, 5003: 51, 5004: 99, 5005: 33}
1: 41, 5002: 0, 5003: 51, 5004: 99, 5005: 33}
1: 41, 5002: 0, 5003: 51, 5004: 17, 5005: 33}
1: 41, 5002: 0, 5003: 51, 5004: 17, 5005: 33}
1: 41, 5002: 0, 5003: 51, 5004: 17, 5005: 33}
1: 41, 5002: 0, 5003: 51, 5004: 99, 5005: 33}
1: 41, 5002: 0, 5003: 73, 5004: 99, 5005: 33}
1: 41, 5002: 0, 5003: 73, 5004: 99, 5005: 33}
1: 41, 5002: 0, 5003: 73, 5004: 99, 5005: 33}
1: 41, 5002: 0, 5003: 73, 5004: 99, 5005: 33}
1: 41, 5002: 0, 5003: 73, 5004: 99, 5005: 33}
1: 41, 5002: 0, 5003: 73, 5004: 99, 5005: 33}
1: 41, 5002: 0, 5003: 39, 5004: 22, 5005: 33}
1: 41, 5002: 0, 5003: 39, 5004: 22, 5005: 33}
1: 41, 5002: 0, 5003: 39, 5004: 22, 5005: 33}
1: 41, 5002: 0, 5003: 73, 5004: 22, 5005: 33}
1: 41, 5002: 0, 5003: 73, 5004: 22, 5005: 33}
1: 41, 5002: 0, 5003: 73, 5004: 22, 5005: 33}
```

```
32}
Updated Distance Array {5000: 100000, 5001: 23, 5002: 33, 5003: 55, 5005: 0, 5004: 1
32}
Updated Distance Array {5000: 100000, 5001: 23, 5002: 33, 5003: 55, 5005: 0, 5004: 5
5}
Updated Distance Array {5000: 100000, 5001: 23, 5002: 33, 5003: 55, 5005: 0, 5004: 5
5}
Updated Distance Array {5000: 100000, 5001: 23, 5002: 33, 5003: 55, 5005: 0, 5004: 5
5}
Updated Distance Array {5000: 100000, 5001: 74, 5002: 33, 5003: 106, 5005: 0, 5004:
55}
Updated Distance Array {5000: 100000, 5001: 68, 5002: 33, 5003: 100, 5005: 0, 5004:
55}
Updated Distance Array {5000: 100000, 5001: 68, 5002: 33, 5003: 100, 5005: 0, 5004:
55}
Updated Distance Array {5000: 100000, 5001: 68, 5002: 33, 5003: 100, 5005: 0, 5004:
55}
Updated Distance Array {5000: 100000, 5001: 93, 5002: 33, 5003: 125, 5005: 0, 5004:
55}
```

```
Updated Distance Array {5000: 100000, 5001: 32, 50
Updated Distance Array {5000: 100000, 5001: 32, 5002: 73, 5003: 0, 5004: 169, 5005: 73}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 73, 5003: 0, 5004: 53, 5005: 73}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 73, 5003: 0, 5004: 53, 5005: 73}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 73, 5003: 0, 5004: 53, 5005: 73}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 73, 5003: 0, 5004: 169, 5005: 73}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 73, 5003: 0, 5004: 169, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 73, 5003: 0, 5004: 169, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 73, 5003: 0, 5004: 95, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 73, 5003: 0, 5004: 95, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 39, 5003: 0, 5004: 17, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 39, 5003: 0, 5004: 17, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 39, 5003: 0, 5004: 17, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 73, 5003: 0, 5004: 95, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 73, 5003: 0, 5004: 95, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 73, 5003: 0, 5004: 95, 5005: 106}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 67, 5003: 0, 5004: 89, 5005: 100}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 67, 5003: 0, 5004: 89, 5005: 100}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 67, 5003: 0, 5004: 89, 5005: 100}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 96, 5003: 0, 5004: 118, 5005: 125}
```

```
1: 140, 5002: 99, 5003: 150, 5004: 0, 5005: 132}
Updated Distance Array {5000: 100000, 5001: 140, 5002: 99, 5003: 150, 5004: 0, 5005: 132}
Updated Distance Array {5000: 100000, 5001: 58, 5002: 17, 5003: 68, 5004: 0, 5005: 50}
Updated Distance Array {5000: 100000, 5001: 140, 5002: 99, 5003: 150, 5004: 0, 5005: 132}
Updated Distance Array {5000: 100000, 5001: 140, 5002: 99, 5003: 150, 5004: 0, 5005: 132}
Updated Distance Array {5000: 100000, 5001: 140, 5002: 99, 5003: 169, 5004: 0, 5005: 132}
Updated Distance Array {5000: 100000, 5001: 85, 5002: 99, 5003: 53, 5004: 0, 5005: 126}
Updated Distance Array {5000: 100000, 5001: 140, 5002: 99, 5003: 169, 5004: 0, 5005: 132}
Updated Distance Array {5000: 100000, 5001: 63, 5002: 22, 5003: 95, 5004: 0, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 63, 5002: 22, 5003: 95, 5004: 0, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 63, 5002: 22, 5003: 95, 5004: 0, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 63, 5002: 22, 5003: 95, 5004: 0, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 49, 5002: 22, 5003: 17, 5004: 0, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 63, 5002: 22, 5003: 95, 5004: 0, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 57, 5002: 22, 5003: 89, 5004: 0, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 57, 5002: 22, 5003: 89, 5004: 0, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 57, 5002: 22, 5003: 89, 5004: 0, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 57, 5002: 22, 5003: 89, 5004: 0, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 57, 5002: 22, 5003: 89, 5004: 0, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 122, 5002: 22, 5003: 118, 5004: 0, 5005: 55}
```

Figure 3: Link State Routing with changing edge costs



```
Updated Distance Array {5000: 100000, 5001: 0, 5002: 41, 5003: 24, 5005: 12, 5004: 140}
Updated Distance Array {5000: 100000, 5001: 0, 5002: 41, 5003: 24, 5005: 41, 5004: 140}
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
Updated Distance Array {5000: 100000, 5001: 0, 500
```

```
Updated Distance Array {5000: 100000, 5001: 41, 5002: 0, 5003: 51, 5004: 17, 5005: 33}
Updated Distance Array {5000: 100000, 5001: 41, 5002: 0, 5003: 51, 5004: 99, 5005: 33}
1: 41, 5002: 0, 5003: 73, 5004: 99, 5005: 33}
1: 41, 5002: 0, 5003: 73, 5004: 99, 5005: 33}
1: 41, 5002: 0, 5003: 73, 5004: 99, 5005: 33}
1: 41, 5002: 0, 5003: 73, 5004: 99, 5005: 33}
1: 41, 5002: 0, 5003: 73, 5004: 99, 5005: 33}
1: 41, 5002: 0, 5003: 73, 5004: 99, 5005: 33}
1: 41, 5002: 0, 5003: 39, 5004: 22, 5005: 33}
1: 41, 5002: 0, 5003: 39, 5004: 22, 5005: 33}
1: 41, 5002: 0, 5003: 39, 5004: 22, 5005: 33}
1: 41, 5002: 0, 5003: 73, 5004: 22, 5005: 33}
1: 41, 5002: 0, 5003: 73, 5004: 22, 5005: 33}
1: 41, 5002: 0, 5003: 73, 5004: 22, 5005: 33}
1: 67, 5002: 0, 5003: 96, 5004: 22, 5005: 33}
1: 67, 5002: 0, 5003: 96, 5004: 22, 5005: 33}
1: 42, 5002: 0, 5003: 74, 5004: 22, 5005: 33}
1: 42, 5002: 0, 5003: 74, 5004: 22, 5005: 33}
1: 66, 5002: 0, 5003: 96, 5004: 82, 5005: 33}
1: 67, 5002: 0, 5003: 96, 5004: 82, 5005: 33}
```

```
5}
Updated Distance Array {5000: 100000, 5001: 34, 5002: 33, 5003: 66, 5005: 0, 5004: 5
5}
Updated Distance Array {5000: 100000, 5001: 34, 5002: 33, 5003: 66, 5005: 0, 5004: 5
5}
Updated Distance Array {5000: 100000, 5001: 34, 5002: 33, 5003: 66, 5005: 0, 5004: 5
5}
Updated Distance Array {5000: 100000, 5001: 34, 5002: 33, 5003: 66, 5005: 0, 5004: 5
5}
Updated Distance Array {5000: 100000, 5001: 34, 5002: 33, 5003: 66, 5005: 0, 5004: 1
15}
Updated Distance Array {5000: 100000, 5001: 34, 5002: 33, 5003: 66, 5005: 0, 5004: 1
15}
Updated Distance Array {5000: 100000, 5001: 34, 5002: 33, 5003: 66, 5005: 0, 5004: 1
15}
Updated Distance Array {5000: 100000, 5001: 34, 5002: 33, 5003: 66, 5005: 0, 5004: 1
15}
Updated Distance Array {5000: 100000, 5001: 34, 5002: 33, 5003: 66, 5005: 0, 5004: 1
15}
```

```
Updated Distance Array {5000: 100000, 5001: 32, 50
Updated Distance Array {5000: 100000, 5001: 32, 5002: 73, 5003: 0, 5004: 95, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 73, 5003: 0, 5004: 95, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 73, 5003: 0, 5004: 95, 5005: 106}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 67, 5003: 0, 5004: 89, 5005: 100}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 67, 5003: 0, 5004: 89, 5005: 100}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 67, 5003: 0, 5004: 89, 5005: 100}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 96, 5003: 0, 5004: 118, 5005: 125}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 96, 5003: 0, 5004: 118, 5005: 125}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 96, 5003: 0, 5004: 118, 5005: 125}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 96, 5003: 0, 5004: 118, 5005: 66}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 96, 5003: 0, 5004: 118, 5005: 66}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 74, 5003: 0, 5004: 96, 5005: 66}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 74, 5003: 0, 5004: 96, 5005: 66}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 74, 5003: 0, 5004: 96, 5005: 66}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 96, 5003: 0, 5004: 118, 5005: 66}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 96, 5003: 0, 5004: 118, 5005: 66}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 96, 5003: 0, 5004: 169, 5005: 66}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 96, 5003: 0, 5004: 169, 5005: 66}
Updated Distance Array {5000: 100000, 5001: 32, 5002: 96, 5003: 0, 5004: 169, 5005: 66}
```

```
1: 63, 5002: 22, 5003: 95, 5004: 0, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 63, 5002: 22, 5003: 95, 5004: 0, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 49, 5002: 22, 5003: 17, 5004: 0, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 63, 5002: 22, 5003: 95, 5004: 0, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 57, 5002: 22, 5003: 89, 5004: 0, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 57, 5002: 22, 5003: 89, 5004: 0, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 57, 5002: 22, 5003: 89, 5004: 0, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 57, 5002: 22, 5003: 89, 5004: 0, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 57, 5002: 22, 5003: 89, 5004: 0, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 122, 5002: 22, 5003: 118, 5004: 0, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 112, 5002: 22, 5003: 118, 5004: 0, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 112, 5002: 22, 5003: 118, 5004: 0, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 89, 5002: 22, 5003: 118, 5004: 0, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 89, 5002: 22, 5003: 118, 5004: 0, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 88, 5002: 22, 5003: 118, 5004: 0, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 88, 5002: 22, 5003: 118, 5004: 0, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 88, 5002: 22, 5003: 118, 5004: 0, 5005: 55}
Updated Distance Array {5000: 100000, 5001: 148, 5002: 82, 5003: 169, 5004: 0, 5005: 115}
Updated Distance Array {5000: 100000, 5001: 148, 5002: 82, 5003: 169, 5004: 0, 5005: 115}
Updated Distance Array {5000: 100000, 5001: 149, 5002: 82, 5003: 169, 5004: 0, 5005: 115}
```

Figure 4: Link State Routing with changing edge costs (changing)

References

[1] Computer network. https://www.javatpoint.com/link-state-routing-algorithm. [Online; accessed 2023-03-23].

[2] Contributors to Wikimedia projects. Link-state routing protocol. https://en.wikipedia.org/wiki/Link-state_routing_protocol, jan 29 2023. [Online; accessed 2023-03-23].

[3] Sushant Gaurav. Link state routing algorithm. https://www.scaler.com/topics/computer-network/link-state-routing-algorithm/, jul 13 2022. [Online; accessed 2023-03-23].

[4] Prepbytes. Link state routing algorithm. https://www.prepbytes.com/blog/miscellaneous/link-state-routing-algorithm/, jan 31 2023. [Online; accessed 2023-03-23].