University of Dhaka

Department of Computer Science and Engineering

CSE-3111 : Computer Networking Lab

Lab Experiment 8 : Implementation of Distance Vector Algorithm

Submitted By:

Name: Muztoba Hasan Sinha

Roll No : 15

Name: Bholanath Das Niloy

Roll No : 22

Submitted On :

April 6, 2023

Submitted To :

Dr. Md. Abdur Razzaque

Md Mahmudur Rahman

Md. Ashraful Islam

Md. Fahim Arefin

Contents

1 Introduction

We were tasked with implementing a network of routers to simulate the Distance Vector Algorithm. This was to improve our understanding of the distance vector routing protocols and further develop the practical understanding required to use this algorithm in distributed systems.

2 Theory

2.1 Overview

Distance vector routing algorithm is a type of routing protocol used in computer networks to calculate the optimal path for transmitting data packets from the source to the destination. It is iterative, asynchronous, distributed and self-synchronizing as opposed to the other dominant routing algorithm, Link State Algorithm.

It is so because -

1. **Distributed**: Each node receives some information from it's neighbours but all nodes independently calculate *distributes* the results to all of it's neighbours.
2. **Iterative**: The process continues until all nodes have calculated the least cost paths and thus have no need for updates, thus no information is exchanged between the nodes.
3. **Self-terminating**: Due to the *Iterative* nature of the algorithm, when there is no information exchanged, the calculations at each nodes just stop.
4. **Asynchronous**: Does not require the nodes to be in sync with each other.

Distance-vector routing protocols rely on the Bellman-Ford algorithm, and in this type of protocol, routers don't have complete information about the entire network topology. Instead, they broadcast their calculated distance values (DV) to other routers and receive similar advertisements from their neighbors. Each router uses these advertisements to update its routing table, and this process repeats until all the routing tables converge to stable values, with updated information being advertised in each cycle. Changes made to the local network or by neighboring routers can trigger this process.

2.2 Routing Information Maintained at Each Node

Each node x in the network N maintains the following information.

1. For each neighbor v, the cost $c(x, v)$ from x to directly attached neighbor, v
2. Node x's **distance vector**, that is, $D_x = [D_x(y) : yinN]$, containing x's estimate of its cost to all destinations, y, in N
3. The distance vectors of each of its neighbors, that is, $D_v = [D_v(y) : yinN]$ for each neighbor v of x

2.3 Information exchange/Messaging among nodes

In DV algorithm, the nodes send a copy of their distance vector to each of it's neighbours periodically and after each change to it's own distance vector, who in turn update their own distance vectors. After receiving each Distance vector the locally maintained information is updated and it updates it's routing table.

2.4 Distance-Vector Algorithm

The Distance Vector Algorithm is basically a derivation of the Bellman-Ford Algorithm, which in turn is derived from the Bellman-Ford equation.

2.4.1 Bellman-Ford Equation

$$d_x(y) = min_v c(x, y) + d_v(y)$$

Where,

- $d_x(y)$ is the least cost path from node x to node y.
- min_v is the minimum taken over all of x's neighbours v
- $c(x, v)$ the cost from x to v
- $d_v(y)$ is the least cost path from node v to node y.

This is simply invoking a triangle inequality to see if a path through a neighbour is lower than the current path computed by the node at that instant.

2.4.2 Bellman-Ford Algorithm/Distance Vector Algorithm

We have modified the Bellman-Ford Algorithm slightly to suit our demands of the Distance vector algorithm. It runs at each node x and considers x as source

Algorithm 1: Bellman-Ford Algorithm

```
Input: Network $N(V, E)$, Distance vector $D_x = [D_x(y) : y \in N]$,
Distance vectors of each neighbour $v$ of $x$, $D_x = [D_x(y) : y \in N]$,
Cost from $x$ to neighbour $v$ $c(x, v)$
Result: Shortest path from $x$ to all other vertices, Next hop to all other vertices from $x$, $\pi$
for each vertex $y \in N$ do
    if $y$ is a neighbour of $x$ then
        $D_x(y) \leftarrow c(x, y)$
        $\pi(y) \leftarrow y$
    else
        $D_x(y) \leftarrow \infty$
        $\pi(y) \leftarrow nil$
$D_x(x) = 0$
for each neighbour $v$ do
    send distance vector $D_x = [D_x(y) : \ y \in N]$ to $v$
while True do
    if link cost changes or a distance vector is received then
        for each $y$ in $N$ do
            $D_x(y) \leftarrow min_v\{c(x, v) + D_v(y)\}$
            if $D_x(y)$ changes for any destination $y$ then
                $\pi(y) \leftarrow v$
                for each neighbour $v$ do
                    send distance vector $D_x \leftarrow [D_x(y) : \ y \in N]$ to $v$
    else
        continue
```

This above algorithm gives us the least cost distance vectors for the node x also the list of next hops for the forwarding table π

2.5 Calculating the routing table

The DV algorithm tries to update the routing table in two cases for each node x.

- When it sees a cost change in one of it's directly attached nodes.
- When it receives a Distance Vector update from one of it's neighbours.

Then it uses the Bellman-Ford Equation for each node y in the network N That is, it runs the Bellman-Ford Algorithm If the node x's distance vector gets updated while running the Bellman-Ford Algorithm it sends the updated distance vector to all of it's neighbours. As long as

2.6 Calculating the forwarding table

To update the forwarding table of the node, we simply need to track the neighbouring node that is the next-hop router along the shortest path. This is done in the above algorithm by tracking the parent list π.

2.7 Link-Cost Changes and Link Failure

For Link Cost change we can have a cost increase or cost decrease. Link Failure can be modelled as an infinite cost increase. We can show the effects of these as follows

2.7.1 Link Cost Decrease

Incase of a link cost decrease the nodes directly attached will be alerted and the algorithm will automatically minimize the cost.

2.7.2 Link Failure or Cost increase

For a substantial increase in Cost if we communicate the increased cost to the 1st degree separated neighbours, they will compare it to their own tables and find that they know a path to the destination at a lower cost, which is erroneously through the parent from the previous state. When the directly connected node finds the message that the separated

node has a lower cost path it tries to route through the separated node. But the separated node will again try to route through the parent thus creating a **Routing Loop**. This goes on until the link cost computed at the separated node is greater than the increased link cost. But what about a Link Failure? it will continue looping the message infinitely this is why this problem is called the **Count to infinity problem**. Which has to be avoided.
From the above discussion we can understand the popular wisdom for DV protocols, "Good news travels fast, Bad news travels slow".

2.8 Poisoned reverse

To fix the Routing Loop we consult the forwarding table. If we find that we are sending a cost to a node that is the next hop router to the destination we simply send an infinite cost. This way there is no chance of getting an erroneous cost loop. But unfortunately this does not extend for 2nd degree neighbours and beyond. More advanced techniques like Split horizon is impelemented to avoid such cases.

3 Performance Metrics

3.1 Time Complexity Analysis

The time complexity of the distance vector routing algorithm is $\mathcal{O}(V^2)$, where V is the number of nodes in the network.

In the initialization phase, each node needs to initialize its distance vector to every other node in the network. This requires $\mathcal{O}(V^2)$ time.

In the update phase, each node broadcasts its distance vector to all its neighboring nodes. Each neighboring node then updates its own distance vector based on the received information. This process continues until all the nodes have updated their distance vectors. In the worst case, each node can have a maximum of $V - 1$ neighbors, which means that each node can potentially receive $V - 1$ distance vectors in each round. Thus, the update phase can take up to $\mathcal{O}(V^2)$ time as well.

In the worst case scenario where each node sends out its entire distance vector to all its neighbors in every iteration, the total time complexity can be $\mathcal{O}(V^3)$. However, in practice, this scenario is unlikely to occur because most networks have sparse connectivity, and the number of updates required will be much less than V^2. Here, we are assuming that the number of edges E in the network is much less than V^2.

3.2 Memory Complexity Analysis

The memory complexity of the distance vector routing algorithm can be represented as $\mathcal{O}(VE)$, where V is the number of nodes and E is the number of edges in the network. This is because each node in the network needs to store its distance vector, which contains information about the cost of reaching every other node in the network. Therefore, the memory required for storing the distance vector at each node is proportional to the number of edges in the network. Since there can be at most $V - 1$ edges per node (in a fully connected network), the overall memory complexity can be expressed as $\mathcal{O}(VE)$.

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.

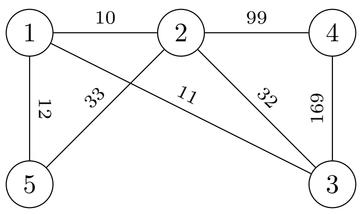


Figure 1: Network Topology

4.2 APSP

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

Table 1: Bellman Ford'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. The graph is not a multigraph. That is two nodes are not directly connected by more than one edge/link.
2. 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 multigraph would not make sense since Bellman-Ford will only route through the least cost path. And the last consideration is inherent to computer networks.

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 init

Here we are initializing the adjacency matrices for each node in our network. We update the costs for all the links adjacent to the current node and we initialize all the distance vector arrays with ∞.

5.2.2 Distance-Vector Algorithm Implementation

This is a function *update_distance_vector()* that updates the distance vector of the current node based on the new distance vector received from a neighboring node. Here's what it does:

1. It extracts the sender node from the *new_dist* array, which is a list of updated distances for each node in the network.
2. It stores the old distance vector of the sender node in the old variable for comparison later.
3. It updates the distance vector of the sender node in the *dist_all* dictionary with the new distances received for each node.
4. It then iterates over all nodes in the network and checks if the current node (ME) has a cost associated with it in the cost dictionary and if it is not in the *broken_links* list.
    - If both conditions are met and the distance from the current node to node i is infinite, it sets the distance from node i to the sender node to infinity in the *dist_all* dictionary. This is done to prevent routing through the sender node to reach the current node if the sender node has no information about the current node, this is also known as poisoned reverse as it quite literally poisons the reverse path with an ∞.
5. Finally, it compares the updated distance vector of the sender node with the old distance vector. If they are not equal, it triggers the *bellman_ford()* function to recalculate the distance vectors for all nodes in the network.

Overall, this function updates the distance vector for the current node based on the information received from a neighboring node and ensures that the new information is propagated to other nodes in the network as necessary.

5.2.3 send

This function is used to build and send a Distance Vector from the node to all of it's neighbours. It has 2 distinct parts. Firstly it builds the Distance Vector 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.

5.2.4 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.

5.2.5 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_distance_vector* function.

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 sleep for a bit to help synchronize the first dry run. Then we call the *bellman_ford* method for the first time prompting packet sending to neighbours. 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 Poisoned Reverse

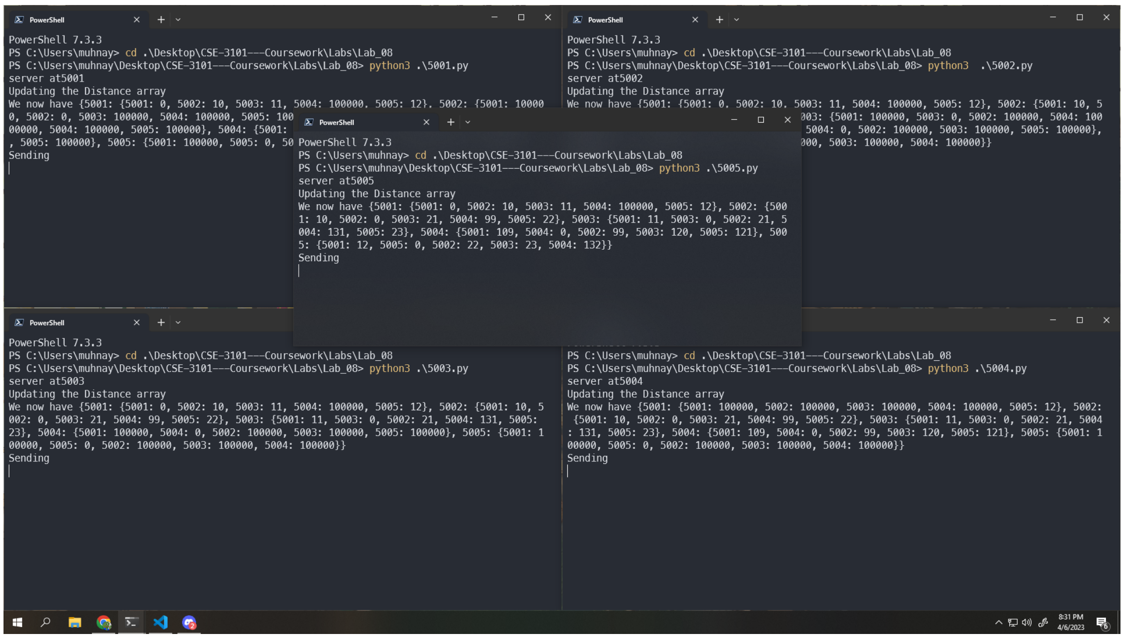
We have handled poisoned reverse as explained in the Distance vector explanation above.

5.3.2 Handling Link Failures

We have handled this by updating all the distance vectors with an ∞ whenever we see that we aren't able to reach to a node that is present in the network.

6 Screenshots of working code

6.1 Distance-Vector Routing with static link costs



```
PowerShell 7.3.3
PS C:\Users\muhnay> cd .\Desktop\CSE-3101---Coursework\Labs\Lab_08
PS C:\Users\muhnay\Desktop\CSE-3101---Coursework\Labs\Lab_08> python3 .\5001.py
server at5001
Updating the Distance array
We now have {5001: {5001: 0, 5002: 10, 5003: 11, 5004: 100000, 5005: 12}, 5002: {5001: 10000
0, 5002: 0, 5003: 100000, 5004: 100000, 5005: 100
00000, 5004: 100000, 5005: 100000}, 5004: {5001:
, 5005: 100000}, 5005: {5001: 100000, 5005: 0, 50
Sending
```

```
PowerShell 7.3.3
PS C:\Users\muhnay> cd .\Desktop\CSE-3101---Coursework\Labs\Lab_08
PS C:\Users\muhnay\Desktop\CSE-3101---Coursework\Labs\Lab_08> python3  .\5002.py
server at5002
Updating the Distance array
We now have {5001: {5001: 0, 5002: 10, 5003: 11, 5004: 100000, 5005: 12}, 5002: {5001: 10, 5
                                         003: {5001: 100000, 5003: 0, 5002: 100000, 5004: 100
                                          5004: 0, 5002: 100000, 5003: 100000, 5005: 100000},
                                         000, 5003: 100000, 5004: 100000}}
```

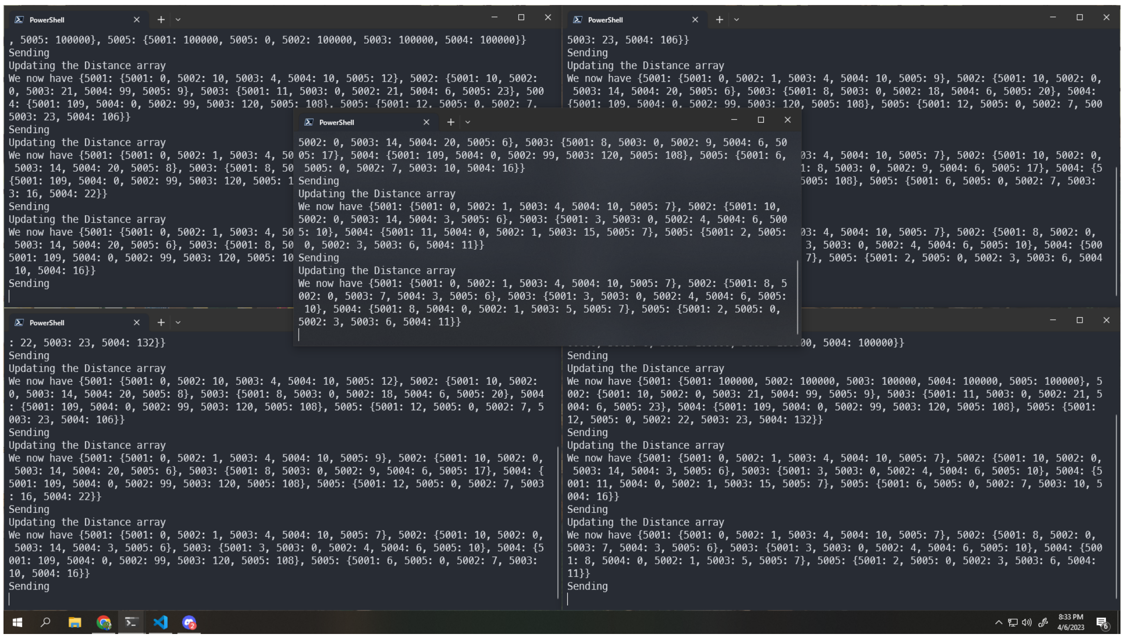
```
PowerShell 7.3.3
PS C:\Users\muhnay> cd .\Desktop\CSE-3101---Coursework\Labs\Lab_08
PS C:\Users\muhnay\Desktop\CSE-3101---Coursework\Labs\Lab_08> python3 .\5005.py
server at5005
Updating the Distance array
We now have {5001: {5001: 0, 5002: 10, 5003: 11, 5004: 100000, 5005: 12}, 5002: {500
1: 10, 5002: 0, 5003: 21, 5004: 99, 5005: 22}, 5003: {5001: 11, 5003: 0, 5002: 21, 5
004: 131, 5005: 23}, 5004: {5001: 109, 5004: 0, 5002: 99, 5003: 120, 5005: 121}, 500
5: {5001: 12, 5005: 0, 5002: 22, 5003: 23, 5004: 132}}
Sending
```

```
PowerShell 7.3.3
PS C:\Users\muhnay> cd .\Desktop\CSE-3101---Coursework\Labs\Lab_08
PS C:\Users\muhnay\Desktop\CSE-3101---Coursework\Labs\Lab_08> python3 .\5003.py
server at5003
Updating the Distance array
We now have {5001: {5001: 0, 5002: 10, 5003: 11, 5004: 100000, 5005: 12}, 5002: {5001: 10, 5
002: 0, 5003: 21, 5004: 99, 5005: 22}, 5003: {5001: 11, 5003: 0, 5002: 21, 5004: 131, 5005:
23}, 5004: {5001: 100000, 5004: 0, 5002: 100000, 5003: 100000, 5005: 100000}, 5005: {5001: 1
00000, 5005: 0, 5002: 100000, 5003: 100000, 5004: 100000}}
Sending
```

```
PS C:\Users\muhnay> cd .\Desktop\CSE-3101---Coursework\Labs\Lab_08
PS C:\Users\muhnay\Desktop\CSE-3101---Coursework\Labs\Lab_08> python3 .\5004.py
server at5004
Updating the Distance array
We now have {5001: {5001: 100000, 5002: 100000, 5003: 100000, 5004: 100000, 5005: 12}, 5002:
 {5001: 10, 5002: 0, 5003: 21, 5004: 99, 5005: 22}, 5003: {5001: 11, 5003: 0, 5002: 21, 5004
: 131, 5005: 23}, 5004: {5001: 109, 5004: 0, 5002: 99, 5003: 120, 5005: 121}, 5005: {5001: 1
00000, 5005: 0, 5002: 100000, 5003: 100000, 5004: 100000}}
Sending
```

Figure 2: Link State Routing static edge costs

6.2 Distance-Vector Routing with Varying link costs



```
, 5005: 100000}, 5005: {5001: 100000, 5005: 0, 5002: 100000, 5003: 100000, 5004: 100000}}
Sending
Updating the Distance array
We now have {5001: {5001: 0, 5002: 10, 5003: 4, 5004: 10, 5005: 12}, 5002: {5001: 10, 5002:
0, 5003: 21, 5004: 99, 5005: 9}, 5003: {5001: 11, 5003: 0, 5002: 21, 5004: 6, 5005: 23}, 500
4: {5001: 109, 5004: 0, 5002: 99, 5003: 120, 5005: 108}, 5005: {5001: 12, 5005: 0, 5002: 7,
5003: 23, 5004: 106}}
Sending
Updating the Distance array
We now have {5001: {5001: 0, 5002: 1, 5003: 4, 50
 5003: 14, 5004: 20, 5005: 8}, 5003: {5001: 8, 50
{5001: 109, 5004: 0, 5002: 99, 5003: 120, 5005: 1
3: 16, 5004: 22}}
Sending
Updating the Distance array
We now have {5001: {5001: 0, 5002: 1, 5003: 4, 50
 5003: 14, 5004: 20, 5005: 6}, 5003: {5001: 8, 50
5001: 109, 5004: 0, 5002: 99, 5003: 120, 5005: 10
 10, 5004: 16}}
Sending
```

```
5003: 23, 5004: 106}}
Sending
Updating the Distance array
We now have {5001: {5001: 0, 5002: 1, 5003: 4, 5004: 10, 5005: 9}, 5002: {5001: 10, 5002: 0,
 5003: 14, 5004: 20, 5005: 6}, 5003: {5001: 8, 5003: 0, 5002: 18, 5004: 6, 5005: 20}, 5004:
{5001: 109, 5004: 0, 5002: 99, 5003: 120, 5005: 108}, 5005: {5001: 12, 5005: 0, 5002: 7, 500

03: 4, 5004: 10, 5005: 7}, 5002: {5001: 10, 5002: 0,
1: 8, 5003: 0, 5002: 9, 5004: 6, 5005: 17}, 5004: {5
5005: 108}, 5005: {5001: 6, 5005: 0, 5002: 7, 5003:

03: 4, 5004: 10, 5005: 7}, 5002: {5001: 8, 5002: 0,
3, 5003: 0, 5002: 4, 5004: 6, 5005: 10}, 5004: {500
7}, 5005: {5001: 2, 5005: 0, 5002: 3, 5003: 6, 5004
```

```
5002: 0, 5003: 14, 5004: 20, 5005: 6}, 5003: {5001: 8, 5003: 0, 5002: 9, 5004: 6, 50
05: 17}, 5004: {5001: 109, 5004: 0, 5002: 99, 5003: 120, 5005: 108}, 5005: {5001: 6,
 5005: 0, 5002: 7, 5003: 10, 5004: 16}}
Sending
Updating the Distance array
We now have {5001: {5001: 0, 5002: 1, 5003: 4, 5004: 10, 5005: 7}, 5002: {5001: 10,
5002: 0, 5003: 14, 5004: 3, 5005: 6}, 5003: {5001: 3, 5003: 0, 5002: 4, 5004: 6, 500
5: 10}, 5004: {5001: 11, 5004: 0, 5002: 1, 5003: 15, 5005: 7}, 5005: {5001: 2, 5005:
 0, 5002: 3, 5003: 6, 5004: 11}}
Sending
Updating the Distance array
We now have {5001: {5001: 0, 5002: 1, 5003: 4, 5004: 10, 5005: 7}, 5002: {5001: 8, 5
002: 0, 5003: 7, 5004: 3, 5005: 6}, 5003: {5001: 3, 5003: 0, 5002: 4, 5004: 6, 5005:
 10}, 5004: {5001: 8, 5004: 0, 5002: 1, 5003: 5, 5005: 7}, 5005: {5001: 2, 5005: 0,
5002: 3, 5003: 6, 5004: 11}}
```

```
: 22, 5003: 23, 5004: 132}}
Sending
Updating the Distance array
We now have {5001: {5001: 0, 5002: 10, 5003: 4, 5004: 10, 5005: 12}, 5002: {5001: 10, 5002:
0, 5003: 14, 5004: 20, 5005: 8}, 5003: {5001: 8, 5003: 0, 5002: 18, 5004: 6, 5005: 20}, 5004
: {5001: 109, 5004: 0, 5002: 99, 5003: 120, 5005: 108}, 5005: {5001: 12, 5005: 0, 5002: 7, 5
003: 23, 5004: 106}}
Sending
Updating the Distance array
We now have {5001: {5001: 0, 5002: 1, 5003: 4, 5004: 10, 5005: 9}, 5002: {5001: 10, 5002: 0,
 5003: 14, 5004: 20, 5005: 6}, 5003: {5001: 8, 5003: 0, 5002: 9, 5004: 6, 5005: 17}, 5004: {
5001: 109, 5004: 0, 5002: 99, 5003: 120, 5005: 108}, 5005: {5001: 12, 5005: 0, 5002: 7, 5003
: 16, 5004: 22}}
Sending
Updating the Distance array
We now have {5001: {5001: 0, 5002: 1, 5003: 4, 5004: 10, 5005: 7}, 5002: {5001: 10, 5002: 0,
 5003: 14, 5004: 3, 5005: 6}, 5003: {5001: 3, 5003: 0, 5002: 4, 5004: 6, 5005: 10}, 5004: {5
001: 109, 5004: 0, 5002: 99, 5003: 120, 5005: 108}, 5005: {5001: 6, 5005: 0, 5002: 7, 5003:
10, 5004: 16}}
Sending
```

```
00, 5004: 100000}}
Sending
Updating the Distance array
We now have {5001: {5001: 100000, 5002: 100000, 5003: 100000, 5004: 100000, 5005: 100000}, 5
002: {5001: 10, 5002: 0, 5003: 21, 5004: 99, 5005: 9}, 5003: {5001: 11, 5003: 0, 5002: 21, 5
004: 6, 5005: 23}, 5004: {5001: 109, 5004: 0, 5002: 99, 5003: 120, 5005: 108}, 5005: {5001:
12, 5005: 0, 5002: 22, 5003: 23, 5004: 132}}
Sending
Updating the Distance array
We now have {5001: {5001: 0, 5002: 1, 5003: 4, 5004: 10, 5005: 7}, 5002: {5001: 10, 5002: 0,
 5003: 14, 5004: 3, 5005: 6}, 5003: {5001: 3, 5003: 0, 5002: 4, 5004: 6, 5005: 10}, 5004: {5
001: 11, 5004: 0, 5002: 1, 5003: 15, 5005: 7}, 5005: {5001: 6, 5005: 0, 5002: 7, 5003: 10, 5
004: 16}}
Sending
Updating the Distance array
We now have {5001: {5001: 0, 5002: 1, 5003: 4, 5004: 10, 5005: 7}, 5002: {5001: 8, 5002: 0,
5003: 7, 5004: 3, 5005: 6}, 5003: {5001: 3, 5003: 0, 5002: 4, 5004: 6, 5005: 10}, 5004: {500
1: 8, 5004: 0, 5002: 1, 5003: 5, 5005: 7}, 5005: {5001: 2, 5005: 0, 5002: 3, 5003: 6, 5004:
11}}
Sending
```

Figure 3: Link State Routing with changing edge costs

References

[1] Computer network. https://www.javatpoint.com/distance-vector-routing-algorithm. [Online; accessed 2023-04-06].

[2] Distance vector routing DVR protocol. *GeeksforGeeks*, oct 4 2017. [Online; accessed 2023-04-06].

[3] Contributors to Wikimedia projects. Distance-vector routing protocol. https://en.wikipedia.org/wiki/Distance-vector_routing_protocol, oct 12 2022. [Online; accessed 2023-04-06].

[4] Snehal Jadhav. Distance vector routing algorithm. https://www.scaler.com/topics/computer-network/distance-vector-routing-algorithm/, oct 7 2022. [Online; accessed 2023-04-06].