Project #3: Part 2: ModivSim

Aslı Ağılönü, Doruk Taneli, Onur İskenderoğlu

Introduction

In this project we have simulated a basic network to observe how the routing is decided. Our networks consist of nodes that act as a router. Each node keeps a forwarding table to decide where to forward an incoming packet. The routing table is constructed through a distance vector algorithm that uses Bellman-Ford algorithm. We used Java as our object-oriented language choice, and used Eclipse IDE and IntelliJ IDEA.

Part 1: Implementing Class Structures and Processing Inputs

For the project we have three main classes: Node, Message and ModivSim. ModivSim class is our main class where the network is created and the distance vector algorithm is executed. Initially the user is prompted to provide an input in .txt format that contains the node information. Each file contains the node's ID, its neighbors, the cost of the link to them and the link's bandwidth. The information regarding each node is stored in a node object described by the Node class. Each node object communicates by sending messages to each other where each message is a Message object. The details about each class will be explained in more detail below.

Node.java

Each router node is represented by a Node object. Node class primarily has linkCost and linkBandwidth hash tables that store the link cost and the link bandwidth of a node's direct neighbors, neighbors hash table that hold neighboring node objects defined by their IDs, a distanceTable 2-D array that stores information regarding the link cost of all other nodes, a bottleNeckBandwidth table which is an array list that holds the bottleneck bandwidths of the node's direct neighbors and a boolean value that holds the node's convergence status determined by the distance vector algorithm.

When a node object is first initialized, it only has information about its direct neighbors thus all the cost values regarding the other nodes are set to infinity, in this case to 999. Each node's cost to itself is 0 and the initial cost to its neighbors is what is provided in the input file. As the distance vector algorithm executes, the nodes send and receive information about the rest of the network through sendUpdate and receiveUpdate methods. sendUpdate method returns true if the update is successfully sent to neighboring nodes. To send an update first, a distance vector for the given node is created. Distance vector for the node contains the shortest path to a given destination node, determined through traversing the distance table. In this case,

distanceVector[i] denotes the shortest cost to reach node i from the current node. When this vector is constructed, the node's direct neighbors are notified through calling their receiveUpdate methods. A message object that contains the distanceVector is created and sent to the neighbors by passing it as a parameter to their respective receiveUpdate method.

A node's receiveUpdate method is called when one of its neighbors execute their sendUpdate method as described above. When receiving an update, the node receives a message object that contains the ID of the node that sent the message and that node's distanceVector. The node that receives the update then compares the received distanceVector with its distance table and updates the values if one of the costs in the received distance vector is lower than the node's currently stored values. Also, it runs a piece of code which works like the Bellman-Ford algorithm in each node. It compares the current node's distance in the table with the incoming distance vector's source's distance to the current node added to the source's distance to the destination node. Lastly, we understand from this method whether the algorithm converged or not by setting a ceiling value for every time this method is called but no changes were made.

Finally when the node converges based on the distance vector algorithm, it constructs a forwarding table. The forwarding table maps the string value of a node's ID to the string value of another node's ID. This mapping denotes that the current node should forward packets with destination represented in the key field through the nodes represented in the value field. The map holds two nodes in the value field indicating the first two nodes with the lowest cost so in the case that the lowest cost link is in use, the node can still forward the message using the second lowest cost hop. To construct the forwardingTable the method traverses through the distanceTable and decides on the lowest cost node hops for the given nodes.

Message.java

A message object holds the sender and receiver node's IDs, the link bandwidth between them and the distance vector that is being sent. It is used as an object that holds the information to be sent between nodes. A message object is called and passed between nodes through sendUpdate and receiveUpdate methods found in Node class.

ModivSim.java

This is the main class of our project where the simulation is executed. Initially the user is asked to provide inputs to create the network topology. Each file that contains the single node information is read, a node object is created and stored in the nodeList list. After processing all inputs each node object is called to create their neighbors list. After the network is successfully created the distance vector algorithm is executed. The nodes are created as a thread, and they terminate when everything is done, meaning that the topology converged, and results are printed out. The results are printed out to a Swing package UI, which allows for better output printing than terminal. In this window, starting and final distance table, starting and final forwarding table, every message sent by the window's owner node, every message the window's owner receives and every change made to the distance table is printed.

Static Link Cost Scenario Test

Below is the manually calculated distance vector table including each iteration and the forwarding table. The algorithm will give this output if it works correctly. I finished this part before the distance vector algorithm was completed so I cannot comment on whether it works correctly or not. The results may differ from the output of the program.

Initial Distance Vectors:

	Dx0	Dx1	Dx2	Dx3	Dx4
D0	0	∞	∞	∞	∞
D1	∞	0	∞	∞	∞
D2	∞	∞	0	∞	∞
D3	∞	∞	∞	0	∞
D4	∞	∞	∞	∞	0

First iteration: (immediate neighbors)

	Dx0	Dx1	Dx2	Dx3	Dx4
D0	0	5	3	∞	∞
D1	5	0	9	∞	1
D2	3	9	0	3	∞
D3	∞	∞	3	0	7
D4	∞	1	∞	7	0

Second iteration: (2 hops)

	Dx0	Dx1	Dx2	Dx3	Dx4
D0	0	5	3	6,N2	6,N1
D1	5	0	8,N0	8,N4	1
D2	3	8,N0	0	3	10,N1/N3
D3	6,N2	8,N4	3	0	7
D4	6,N1	1	10,N1/N3	7	0

Third Iteration: (3 hops)

	Dx0	Dx1	Dx2	Dx3	Dx4
D0	0	5	3	6,N2	6,N1
D1	5	0	8,N0	8,N4	1
D2	3	8,N0	0	3	9,(N0,N1)
D3	6,N2	8,N4	3	0	7
D4	6,N1	1	9,(N1,N0)	7	0

The topology converges in 3 rounds.

Final Distance Table:

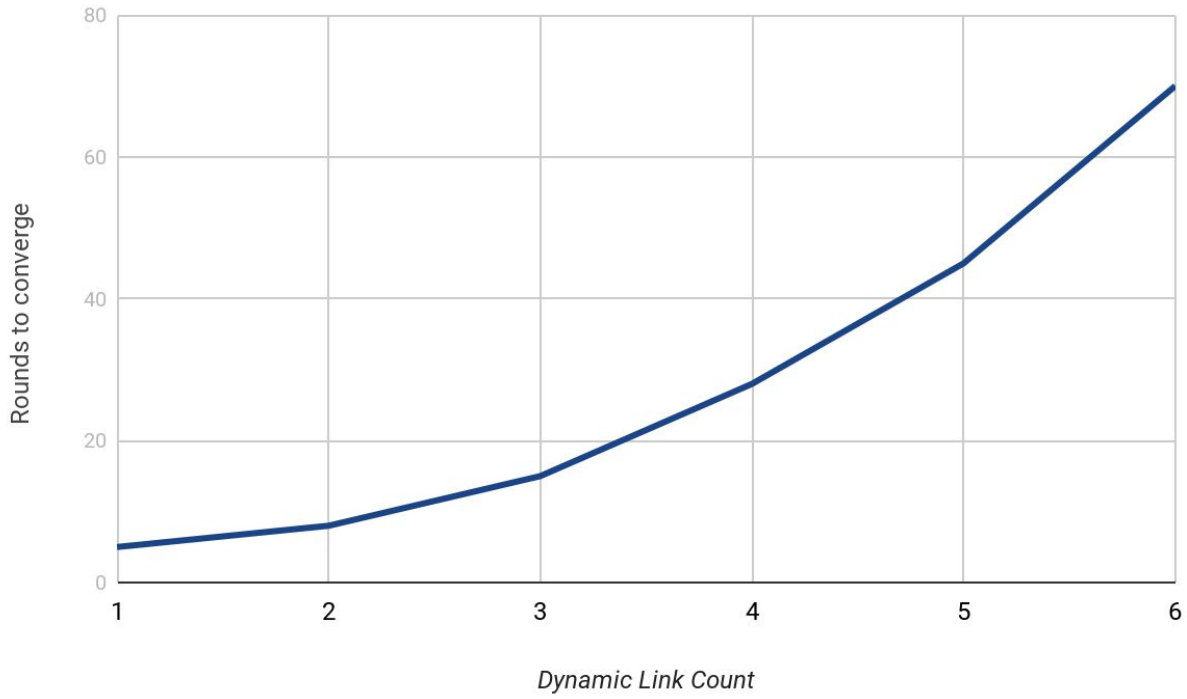
	Dx0	Dx1	Dx2	Dx3	Dx4
D0	0	5	3	6	6
D1	5	0	8	8	1
D2	3	8	0	3	9
D3	6	8	3	0	7
D4	6	1	9	7	0

Forwarding Table:

Keys↓	N0	N1	N2	N3	N4
0	-	(0,4)	(0,3)	(2,4)	(1,3)
1	(1,2)	-	(0,1)	(4,2)	(1,3)
2	(2,1)	(0,2)	-	(2,4)	(1,3)
3	(2,1)	(4,2)	(3,0)	-	(3,1)
4	(1,2)	(4,0)	(0,1)	(4,2)	-

Dynamic Link Cost Scenario Test

Below is the graph for how many rounds it takes to converge for from 1 to 6 dynamic links. The dynamic link costs were not implemented in the time of writing this. So it is the expected graph with increasing rounds for increasing dynamic link count.



Flow Routing Simulation

For flow routing simulation, I created 2 new classes: Flow and FlowRouter.

- Flow is a relatively basic class that has 4 attributes: name, startNode, endNode, and flowSize; getters and setters, constructor and a toString function.
- FlowRouter has the methods required to run the flow routing simulation from ModivSim class.

FlowRouter class explanation:

- When calling the constructor, FlowRouter reads the flows from flow.txt, converts them to Flow class and adds them to a list.
- ProcessFlows function iterates over all the flows. It starts from the startNode, gets the forwarding table of the node, and tries to reach the endNode of the flow using the nodes of the forwarding table, prioritizing the first element of the value in the table.
- If the endNode can be reached, it sets the nodes as inUse for flowSize/bandwidth seconds.

- If the endNode of the flow cannot be reached as the nodes are already in use, the flow is queued.
- Every ‘period’ seconds which I arbitrarily chose as 2,
    - the inUse attribute of all nodes is decreased by period amount.
    - FlowSize of active flows is decreased by period*bottleneckBandwidth
    - Routes of flows are calculated again.
- If there is no route for the flow available, flow is added to the queue and will be tried to process in the next period.
- If all flows are completed, the algorithm stops.

I finished this part before the distance vector algorithm was created. So below is the theoretical output when the distance vector algorithm works correctly and the annotated parts in my code are uncommented when the algorithm is finished.

Example output:

```
t=0
Flow [name=A, startNode=0, endNode=3, flowSize=100]
Route: [0-2-3]          Bandwidth: 10
Flow [name=B, startNode=0, endNode=3, flowSize=200]
Route: [0-1-4-3]        Bandwidth: 5
Flow [name=C, startNode=1, endNode=2, flowSize=100]
Route: [1-2]            Bandwidth: 5

t=2
Flow [name=A, startNode=0, endNode=3, flowSize=80]
Route: [0-2-3]          Bandwidth: 10
Flow [name=B, startNode=0, endNode=3, flowSize=190]
Route: [0-1-4-3]        Bandwidth: 5
Flow [name=C, startNode=1, endNode=2, flowSize=90]
Route: [1-2]            Bandwidth: 5

t=4
Flow [name=A, startNode=0, endNode=3, flowSize=60]
Route: [0-2-3]          Bandwidth: 10
Flow [name=B, startNode=0, endNode=3, flowSize=180]
Route: [0-1-4-3]        Bandwidth: 5
Flow [name=C, startNode=1, endNode=2, flowSize=80]
Route: [1-2]            Bandwidth: 5
--------
t=10
Flow [name=B, startNode=0, endNode=3, flowSize=150]
Route: [0-2-3]          Bandwidth: 10
Flow [name=C, startNode=1, endNode=2, flowSize=50]
Route: [1-2]            Bandwidth: 5
--------
t=20
Flow [name=B, startNode=0, endNode=3, flowSize=50]
Route: [0-2-3]          Bandwidth: 10
```

Task Distribution

We used the suggested task distribution

Aslı: Reading and processing the inputs, implementing Node and Message classes.
Onur: The implementation of Distance Vector Routing algorithm within ModivSim class.
Doruk: Test of ModivSim, Flow Routing Simulation, graph (see Requirements section).