COMP 416
Spring 2020

Project 3 Part 2: ModivSim Modified Distance Vector Routing Simulator

Ahmet Uysal

İpek Köprülülü

Furkan Şahbaz

Contents

1 Main Classes

ModivSim is a modified distance vector algorithm simulator that utilizes multiple classes to implement its logic and overall simulation flow. The main classes we created to implement ModivSim are Node, Link, Message, and Flow classes.

1.1 Node

Each node instance has the following fields:

- `int nodeID`
- `Map<Integer, Integer> linkCost`
- `Map<Integer, Integer> linkBandwidth`
- `HashMap<Integer, Integer> distanceVector`
- `Map<Integer, Map<Integer, Integer>> distanceTable`
- `List<Integer> neighbors`
- `Map<Integer, Integer> bottleNeckBandwidthTable`
- `boolean isUpdateRequired`

The `linkCost` and `linkBandwidth` fields store the initially provided cost and bandwidth values for each of a given Node's neighbors. During initiation, the `distanceTable` and the `distanceVector` are initiated with the initially given costs as well. During simulation, when a message is obtained from the concurrent message queue (that is contained within the main simulation class ModivSim) has a certain node ID as its destination node, the `receiveUpdate` method of the corresponding node is invoked, in which any message causing an update within the node updates the `isUpdateRequired` boolean field accordingly. Note that `receiveUpdate` is a `synchronized` method to prevent concurrency related complications. Another periodically invoked method of a node is the `sendUpdate` method, which, if any update has occurred prior to being called, prepares a message containing the node's updated situation in order to be sent to each one of its neighbors, and places it in the concurrent message queue.

```
...
neighborDistanceVector.forEach((nodeId, cost) -> {
    if (nodeId == this.nodeId)
        return;

    if (!this.distanceTable.containsKey(nodeId)) {
        this.distanceTable.put(nodeId, new HashMap<>());
        this.distanceTable.get(nodeId).put(neighborID, cost + costToNeighbor);
        this.distanceVector.put(nodeId, cost + costToNeighbor);
        isVectorUpdated.set(true);
    } else if (cost + costToNeighbor < this.distanceVector.get(nodeId)) {
        this.distanceVector.put(nodeId, cost + costToNeighbor);
        this.distanceTable.get(nodeId).put(pathToNeighbor, cost + costToNeighbor);
        isVectorUpdated.set(true);
    } else if (!this.distanceTable.get(nodeId).containsKey(pathToNeighbor)) {
        this.distanceTable.get(nodeId).put(pathToNeighbor, cost + costToNeighbor);
    } else if (cost + costToNeighbor < this.distanceTable.get(nodeId).get(pathToNeighbor)) {
        this.distanceTable.get(nodeId).put(pathToNeighbor, cost + costToNeighbor);
    }
});
...
```

Code snippet 1: Updating distanceTable and distanceVector in the `receiveUpdate` method of Node class.

As Code snippet 1 shows, the `receiveUpdate` method is responsible for updating the `distanceTable` and the `distanceVector` of a node, with information gathered from an updated neighbor's message. In order to check for any updates between the current node and any destination nodes, the distance vector obtained from message from the neighbor node is traversed. If the newly calculated total cost to a node is less than the cost that is currently stored in the `distanceTable` or the `distanceVector`, the cost is updated accordingly. This check should be done symmetrically within the distance table, since if the shortest path to a node passes through another, the cost updates should be done accordingly. Furthermore, if a node is currently not in the `distanceVector`, it is added with the newly calculated cost.

1.2 Message

Each message instance has the following fields:

- `int senderID`
- `int receiverID`
- `int linkBandwidth`
- `Map<Integer, Integer> senderDistanceVector`

The Message class plays a crucial role in transferring information between nodes, as the information regarding receiver node and updated `distanceVector` are contained in Message objects. During simulation, the ModivSim class is mainly responsible for obtaining messages from the concurrent message queue, and delivering them to the corresponding nodes by simply invoking their `receiveUpdate` methods.

1.3 Link

Each link instance has the following fields:

- `int node1Id`
- `int node2Id`
- `int cost`
- `int bandwidth`
- `int bandwidthInUse`

Note: `node1Id` and `node2Id` fields are symmetric, i.e, two link objects are also equal if link1.node1Id == link2.node2Id and link1.node2Id == link2.node1Id.

`Link` objects are mainly created to keep track of used bandwidth amount. In our simulation, we allowed concurrent usage of a link by more than one flow similar to real world networks. In order to implement this, we needed a way to keep track of the amount of bandwidth used rather than just whether it is used. `Link` class also helped us keep track of dynamic links.

1.4 Flow

Each flow instance has the following fields:

- `String name`
- `int sourceId`
- `int destinationId`
- `int totalDataMbits`

- `double completionTime`
- `double sentDataMbits`
- `int usedBandwidth`
- `List<Link> assignedPath`
- `boolean isDone`

`Flow` objects are used in `FlowRouting` class to store the information read from the flows.txt file. It helps us to keep track of the mapping between the flows' information read from the file and available paths assigned to them. Beside keeping the path as `Link` objects, it is also responsible for keeping the bandwidth information to use while releasing the links when the flow is finished.

2 Interactions

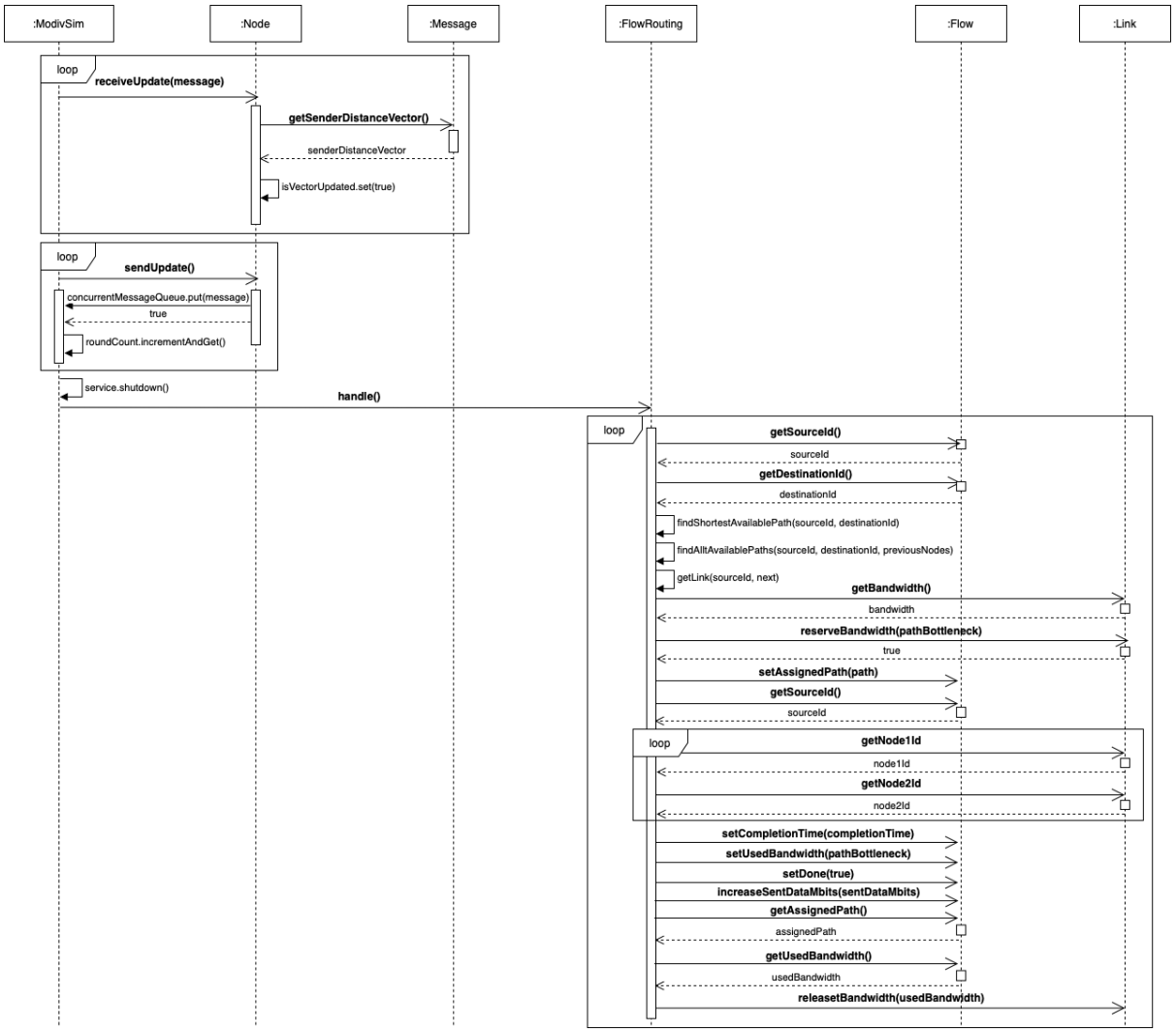

Figure 1: Sequence Diagram for ModivSim Execution

2.1 ModivSim

ModivSim is the class that is responsible for performing the modified distance vector algorithm simulation, the interactions of which can be observed in Fig. 1, for an example execution of the simulation. The program starts by reading the files that contain the initial node information, and generating the corresponding nodes. During initialization, special consideration is given to the links since the program needs to keep track of dynamic links. After the initializations are completed, the round mechanism is implemented by initiating a `ScheduledExecutorService`, which periodically sends updates from each

node, and also updates dynamic link by generating a random boolean for each one in each round.

```
while (true) {
    try {
        Message message = concurrentMessageQueue.poll(timeoutMilliseconds, TimeUnit.MILLISECONDS);
        if (message == null) {
            break;
        }
        int curNode = message.getReceiverId();
        nodes.get(curNode).receiveUpdate(message);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

Code snippet 2: Transferring messages between nodes.

During the rest of the simulation, the ModivSim class invokes the `receiveUpdate` method of each node that is the correspondent of the message obtained from a concurrent message queue (in order to preserve concurrency).

```
long roundPeriodMilliseconds = 1000;
service.scheduleAtFixedRate(() -> {
    boolean anyUpdates = false;
    for (Node node : nodes.values()) {
        anyUpdates |= node.sendUpdate();
    }
    if (anyUpdates) {
        roundCount.incrementAndGet();
    }
}, 0, roundPeriodMilliseconds, TimeUnit.MILLISECONDS);
```

Code snippet 3: `ModivSim` periodically calls `sendUpdate` method of all `Node` instances.

The scheduler then invokes the `sendUpdate()` method of each node contained within the simulation, which, as mentioned earlier, updates the current distanceVector, and notifies the node's direct neighbors by preparing messages for them and placing them in the queue.

The convergence of the simulations determined when no messages have been queued to the concurrent message queue for a certain duration, which is 5 seconds for this simulation. After the timeout occurs, the loop simply breaks and the service is shutdown. Once the simulation has converged, how many rounds it took for the simulation to converge is displayed, and then the simulation moves on to the flow routing part.

2.2 Testing

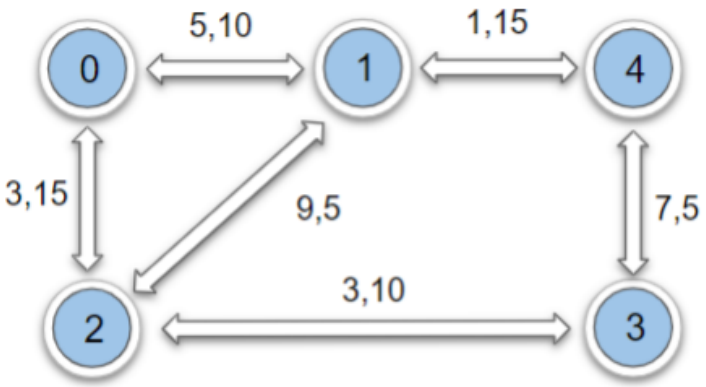

Figure 2: Topology that is used for testing.

2.2.1 Static Link Scenario

The topology was first simulated for a static link scenario, where the links in the topology are those in Fig. 2. This topology was also traced by hand, in order to confirm the correctness of the results.

The resulting distance vectors for hand iterations after initial placement are as follows:

1. Node 0: {Node 1: 5, Node 2: 3}
   Node 1: {Node 0: 5, Node 2: 9, Node 4: 1}
   Node 2: {Node 0: 3, Node 1: 9, Node 3: 3}
   Node 3: {Node 2: 3, Node 4: 7}
   Node 4: {Node 1: 1, Node 3: 7}

2. Node 0: {Node 1: 5, Node 2: 3, Node 3: 6, Node 4: 6}
   Node 1: {Node 0: 5, Node 2: 8, Node 3: 12, Node 4: 1}
   Node 2: {Node 0: 3, Node 1: 8, Node 3: 3, Node 4: 9}
   Node 3: {Node 0: 6, Node 1: 12, Node 2: 3, Node 4: 7}
   Node 4: {Node 0: 6, Node 1: 1, Node 2: 9, Node 3: 7}

3. Node 0: {Node 1: 5, Node 2: 3, Node 3: 6, Node 4: 6}
   Node 1: {Node 0: 5, Node 2: 8, Node 3: 8, Node 4: 1}
   Node 2: {Node 0: 3, Node 1: 8, Node 3: 3, Node 4: 9}
   Node 3: {Node 0: 6, Node 1: 8, Node 2: 3, Node 4: 7}
   Node 4: {Node 0: 6, Node 1: 1, Node 2: 9, Node 3: 7}

Therefore, the resulting forwarding tables for each node (according to the modified algorithm) is as follows:

Node 0: {Node 1: (1, 2), Node 2: (2, 1), Node 3 (2, 1), Node 4 (1, 2)}

Node 1: {Node 0: (0, 4), Node 2: (0, 2), Node 3: (4, 0), Node 4: (4, 0)}

Node 2: {Node 0: (0, 3), Node 1: (0, 1), Node 3: (3, 0), Node 4: (0, 1)}

Node 3: {Node 0: (2, 4), Node 1: (4, 2), Node 2: (2, 4), Node 4: (4, 2)}

Node 4: {Node 0: (1, 3), Node 1: (1, 3), Node 2: (1, 3), Node 3: (3, 1)}

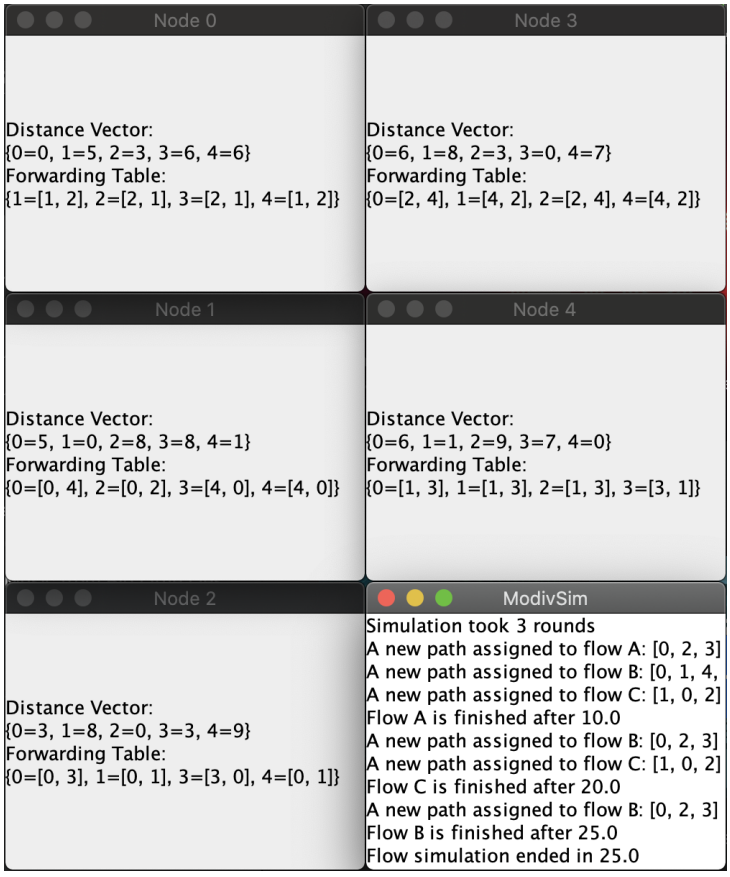

Figure 3: Static Link Results.

As Fig. 3 shows, the hand iterations and the simulation results agree, where distance vectors and forwarding tables for nodes at every round can be observed in the each node's window.

2.2.2 Dynamic Link Scenario

We tested our implementation with the topology provided in project description. We randomly selected sets of links that will dynamically change their costs and modified the topology files accordingly. You can find our results below.

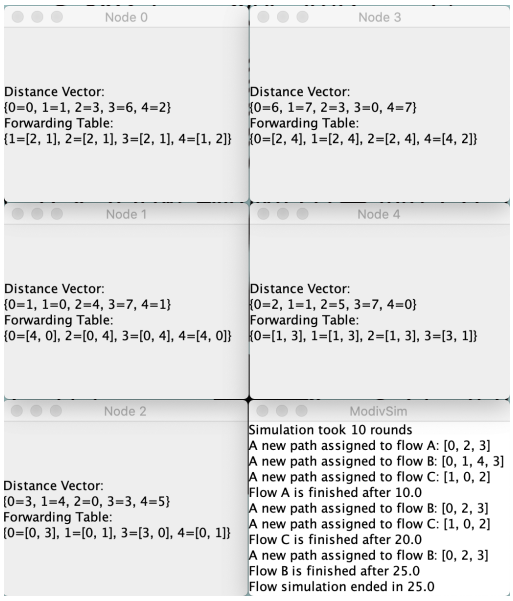

Figure 4: 1 Link: 0-1

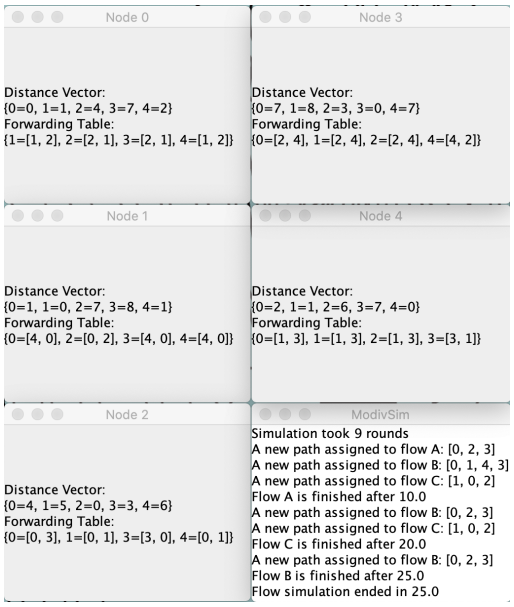

Figure 5: 2 Links: 0-1, 0-2

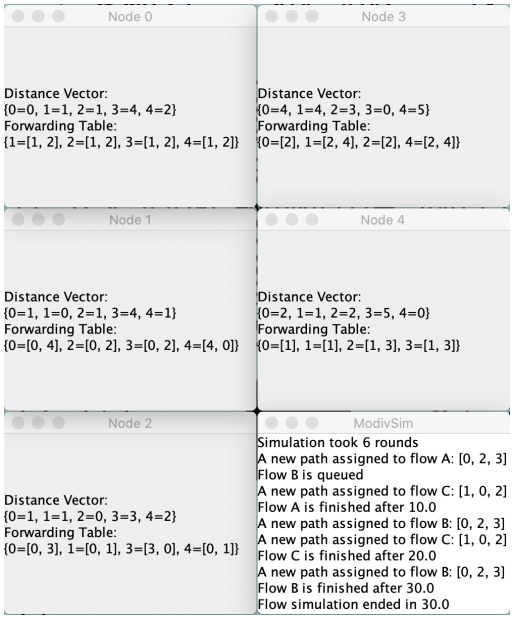

Figure 6: 3 Links: 0-1, 0-2, 1-2

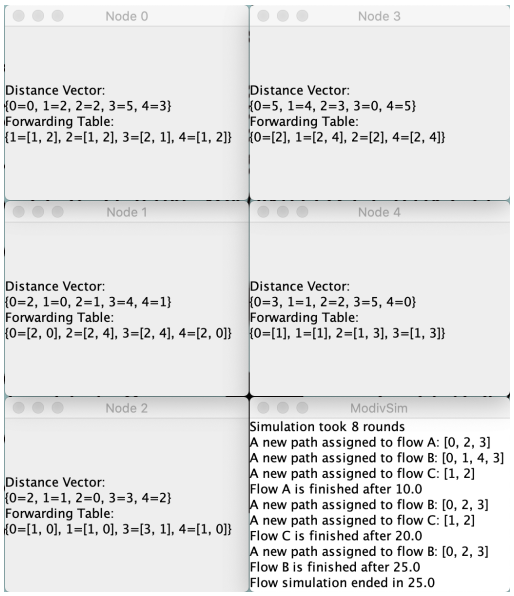

Figure 7: 4 Links: 0-1, 0-2, 1-2, 1-4

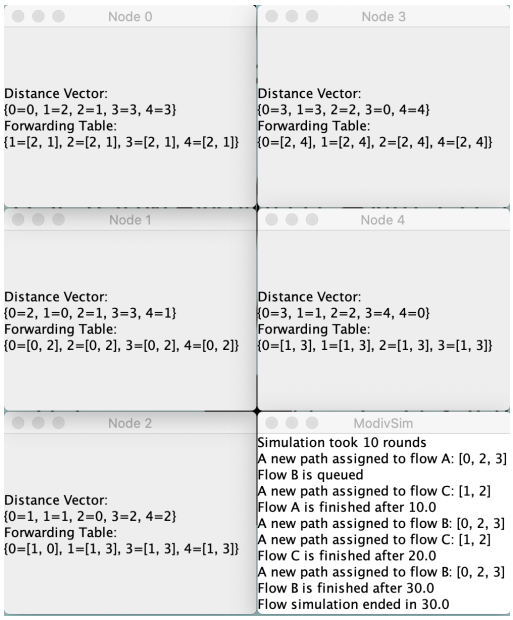

Figure 8: 5 Links: 0-1, 0-2, 1-2, 1-4, 2-3

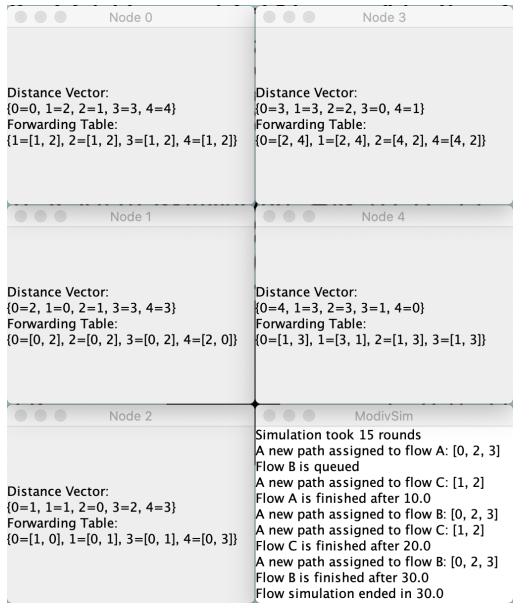

Figure 9: 6 Links: 0-1, 0-2, 1-2, 1-4, 2-3, 3-4

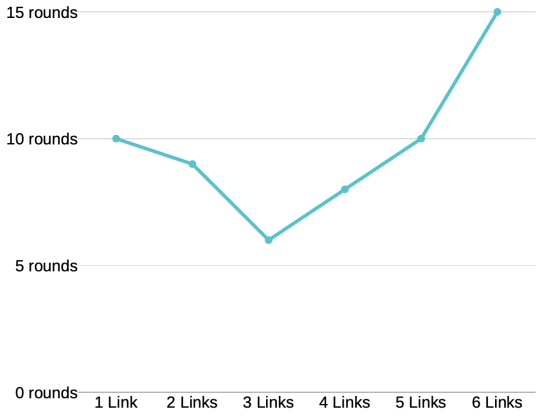

Figure 10: Dynamic Link Count/Round to Converge Graph for the given outputs.

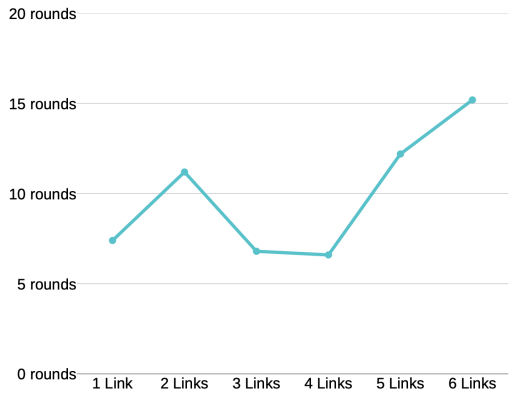

Figure 11: Average Dynamic Link Count/Round to Converge Graph over 3 runs of simulation

As Fig.s 4-9 and graphs in Fig.s 10-11 show, increasing the number of dynamic links has mostly caused more iterations to be needed. Possibly the choice of links for 3 links has sequantially made it easier for the simulation to converge, however larger topologies with more complex structures would certainly need more iterations. Furthermore, the tests were done multiple times, and the average number of iterations were reported, and the slight change in number of iterations are demonstrated in Fig.s 10-11.

2.3 Flow Routing Simulation

FlowRouting class is responsible for reading the flows from the flows.txt file and assigning the possible shortest paths to those flows for the corresponding time calculated with the bottleneck bandwidth. Flow routing simulation starts with creating a `FlowRouting` object in the ModivSim class and calling the `handle` method of it. When the object is created, it immediately reads the text file and keeps the information by creating new Flow objects and store them in a map. After `handle` method is called, the task starts and it checks the flows one by one in a loop which continues until there is no flow remained in the activated flows' map. Firstly, it checks if a flow is activated and finished. If not, it tries to create an available path by calling `findShortestAvailablePath` method which calls `findAllAvailablePaths` method as we can also observe from the Fig. 1. The method checks the forwarding tables of the nodes to create a path, possibly the shortest one. If a path is found, it is assigned to the corresponding flow in the `handle` method. It is deleted from the queue if it was queued and added into the `activeFlows` list. The bottleneck bandwidth for the path is found and completion time for the flow is calculated. The bandwidth value of the corresponding links is decreased by bottleneck bandwidth which was calculated. After each flow is checked once, it continues with checking if any of the activated flows fills its time. If it is, then the flow is deleted from the `activeFlows` list and the corresponding field of the `Flow` object is set as done. To be sure that the task works as efficient as possible, it deletes the path information of all the flows and starts from the beginning since the released links may offer a shorter path for them. The code below captures the finished flows and resets the path information of all others.

```
1  Flow flowToFinish = activeFlows.stream().min(Comparator.comparingDouble(Flow::getCompletionTime)).get();
2  flowToFinish.setDone(true);
3  double timeToNextStep = flowToFinish.getCompletionTime();
4  totalTimeSpent += timeToNextStep;
5  appOutput.append("\nFlow " + flowToFinish.getName() + " is finished after " + totalTimeSpent);
6  activeFlows.forEach(fl -> {
7      fl.increaseSentDataMbits(fl.getUsedBandwidth() * timeToNextStep);
8      fl.getAssignedPath().forEach(link -> link.releaseBandwidth(fl.getUsedBandwidth()));
9  });
10 activeFlows.clear();
```

Code snippet 4: `FlowRouting` checks if the time of a flow is finished and resets the path information of all other flows.

When all activated flows are finished and the `activeFlows` list is empty, it breaks the loop and stops working.

3 Task Distribution

Furkan Şahbaz: Reading and processing the inputs, implementing Node and Message classes

Ahmet Uysal : The implementation of Distance Vector Routing algorithm within ModivSim class

İpek Köprülülü: Test of ModivSim, Flow Routing Simulation, graph