ECE/CS 438: Communication Networks **Spring 2024**

Machine Problem #3: Routing Protocol

Released: March 31st, 2024 *Due: 11:59pm, April 14th, 2024*

Lead TA: Dachun Sun

1 Introduction

In this MP, you will implement your own routing protocol. The basic approach you use can be **either the link state (LS) or distance vector (DV) protocol – your choice**. Implementing both is also encouraged. Regardless of which you use, your program will read a file to get the network's topology and what messages to send, and write to an output file a log of what happened to those messages (the path along which they were delivered).

2 The Router Program

Your program should contain a collection of imaginary routing nodes that carry out their routing protocol (using your choice of protocol: link state or distance vector). These imaginary nodes are just data structures in your program. **There is no socket programming involved. In other words, you will be simulating what happens in a distributed network, in a way that the simulation can run on a single machine**.

We have provided a simple starter program for the MP along with the Makefile for compiling the programs. You are free to modify the code or add additional files, but make sure that the Makefile generates the appropriate executables (for more details see submission instructions at the end of this document).

Your program should first read the input, which consists of network topology, the messages (and their source/destination) that have to be sent across the network, and the topology changes to the network. Your program should create the appropriate data structures (e.g. node, forwarding table, forwarding table entry, routing update, etc.). You should update these data structures when you create the topology or when any of the changes are applied to the topology.

The core of the program is a simulation of the routing protocol. Here's roughly what you need to implement for the two protocols.

- **Distance Vector:** In a distance vector protocol, nodes gossip with each other about their distances with each other. When a link comes up or an existing link either goes offline or has its cost changed, the nodes for that link register that change and send updated distances to their neighbors who then update their forwarding table entries based on it. They then further propagate their distance information to their neighbors and so on. Your program should have data structures representing the

routing tables of each node. It should update the tables at a node when a new route has been found or a previously available route is no longer available, and should correctly propagate these changes across the (simulated) network.

- **Link State:** In a link state protocol, nodes gossip with each other about the state of the topology. When a node detects that the topology has changed, it runs the shortest path algorithm to find the new routes. In a real network, topology updates (called Link State Advertisements) would be propagated among nodes, until eventually all nodes have a complete and up-to-date view of the topology after a change. For this MP, you do not have to simulate the message propagation. Instead, your program is given the topology change, and you can assume that *all nodes have the complete up-to-date view of the topology,* initially and after each change. Your program should simulate what happens at each node after it has learned of the initial topology or a change – specifically, the recomputation of routes for the current topology.

Once the routing tables have converged for the initial topology, for each node, write out the node's forwarding table (in ascending order of node ID, see "Output format" section for details). Then, have some of your nodes send messages to certain other nodes, with the data forwarded according to the nodes' forwarding tables. The sources, destinations, and message contents are specified in the message file; see below for the format.

Then, one at a time, apply each line in the topology changes file (see below) in order. After each change, once the routing tables have re-converged, print out the forwarding tables and resend all of the messages. More specifically, because the topology has changed, your program will have to follow the (DV or LS) routing protocol to change its routing decisions and arrive at a correct forwarding table for the new network topology. Then, you should re-send the same set of messages, which now may follow updated paths.

3 Tie breaking

In general, you can follow a standard version of the routing algorithm, such as what we have discussed in the lecture. However, there is an important detail. We would like everyone to have consistent output even on complex topologies, so we ask you to follow specific tie-breaking rules.

1. **Distance Vector:** when two equally good paths are available, your node should choose the one whose next-hop node ID is lower.
2. **Link State:** If a current-best-known path and newly found path are equal in cost, choose the path whose last node before the destination has the smaller ID.
   Example: The source is 1, and the current-best-known path to destination 9 is $1 \rightarrow 4 \rightarrow 12 \rightarrow 9$. We are currently processing node 10 (i.e., considering routes through 10 to its neighbors). $1 \rightarrow 2 \rightarrow 66 \rightarrow 4 \rightarrow 5 \rightarrow 10 \rightarrow 9$ costs the same as path $1 \rightarrow 4 \rightarrow 12 \rightarrow 9$. We will switch to the new path, since $10 < 12$.

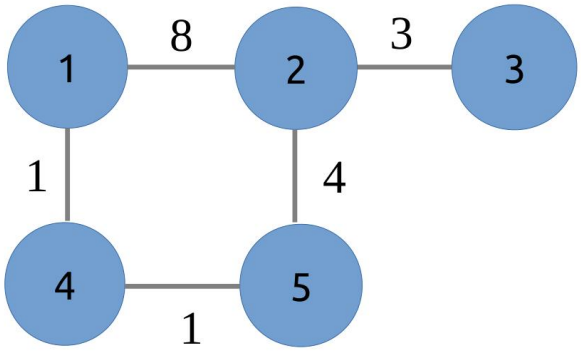

Figure 1: Sample Topology

4 Input Formats

The program reads three files: the *topology* file, the *message* file, and the *topology changes* file.

Your program will be run using the following commands:

```
./linkstate topofile messagefile changesfile
```

```
./distvec topofile messagefile changesfile
```

You only have to implement one of these. The Autograder will attempt to run both and take the highest score of the two programs. Be sure that `make` does not produce a compilation error even if you do not plan to implement one of them.

All files have their items delimited by newlines. Each has a specific format and meaning, which we will describe next. The files we test your code with will follow these descriptions exactly, with nothing extraneous or improperly formatted.

4.1 Topology file

The *topology* file represents the initial topology. A line in the *topology* file represents a link between two nodes and is structured this way:

```
<ID of a node> <ID of another node> <cost of the link between them>
```

Example topology file:

```
1 2 8
2 3 3
2 5 4
4 1 1
4 5 1
```

This would correspond to a topology of Figure 1.

4.2 Changes file

The *topology changes* file represents a sequence of changes to the topology, to be applied one by one. The *changes* file has the same format as the topology file. The first line in the changes file is the first change to apply, the second line is the second, etc. Cost -999 (and only -999) indicates that the previously existing link between the two nodes is broken. (Real link costs are always positive; never zero or negative.) Example changes file:

```
2 4 1
2 4 -999
```

This would add a link between 2 and 4 with cost 1, and then remove it afterwards.

4.3 Messages file

The *message* file describes which nodes should send data to whom once the routing tables converge. (The tables should converge before any of the topology changes, as well as after each change). All messages in the *message* file should be sent every time the tables converge. So, if there are two message lines in the messagefile, and two changes in the changesfile, a total of 6 messages should be sent: 2 initially, 2 after the first change in changesfile has been applied, and 2 after the second line in changesfile has been applied.

A line in the *message* file looks like

```
<source node ID> <dest node ID> <message text>
```

Example message file:

```
2 1 here is a message from 2 to 1

3 5 this one gets sent from 3 to 5!
```

The message file would cause "here is a message from 2 to 1" to be sent from 2->1, and "this one gets sent from 3 to 5!" from 3->5. Note that node **IDs could go to double digits**.

5 Output Format

Write all output described in this section to a file called "`output.txt`". The output file will have the general layout as follows:

```
<forwarding table entries for node 1>
<forwarding table entries for node 2>
<forwarding table entries for node 3>
<forwarding table entries for node ...>
<message output line 1>
<message output line 2>
<message output line ...>
```

——— At this point, 1^{st} change is applied

```
<forwarding table entries for node 1>
<forwarding table entries for node 2>
<forwarding table entries for node 3>
<forwarding table entries for node ...>
```

<message output line 1>
<message output line 2>
<message output line ...>
——— At this point, 2^{nd} change is applied
<forwarding table entries for node 1>
<forwarding table entries for node 2>
<forwarding table entries for node 3>
<forwarding table entries for node ...>
<message output line 1>
<message output line 2>
<message output line ...>
——— And so on...

The format for a single forwarding table entry should be like:

<destination> <nexthop> <pathcost>

where nexthop is the neighbor we hand the destination's packets to, and path cost is the total cost of this path to the destination. The table should be sorted by destination.

Example of forwarding table entries for node 2 from the example topology:

```
1 5 6
2 2 0
3 3 3
4 5 5
5 5 4
```

There is one single space in between each number, with each row on its own line. As you can see, the node's entry for itself should list the nexthop as itself, and the cost as 0. If a destination is not reachable, do not print its entry.

Your program should print all nodes' tables, sorted according to the node ID. So, the example for node 2 would have been preceded by a similarly formatted table for node 1, and followed by the tables of nodes 3, 4, and 5.

When a message is to be sent, print the source, destination, path cost, path taken (including the source, but NOT the destination node), and message contents in the following format:
''from <x> to <y> cost <path_cost> hops <hop1> <hop2> <...> message <message>''
e.g.: ''from 2 to 1 cost 6 hops 2 5 4 message here is a message from 2 to 1''
Print messages in the order they were specified in the messages file. If the destination is not reachable, please say ''from <x> to <y> cost infinite hops unreachable message <message>''

Please do not print anything else; any diagnostic messages or the like should be commented out before submission. **However, if you want to organize the output a little, it's okay to print as many blank lines as you want in between lines of output**.

Both messagefile and changesfile can be empty. In this case, the program should just print the forwarding table.

6 Submission Instructions

For this programming assignment, you need to submit a zip file named `mp3.zip` to Gradescope. This zip file must contain a `Makefile`, whose default target produces executables named `distvec` or `linkstate`.

Notes:

1. For this assignment, the Git repository is NOT monitored by Autograder and needs active submission to Gradescope. Adding partner IDs is similar to homework submission.
2. During the compilation of your program, only the command `make clean && make` will be executed. You may modify your Makefile, for instance, if you wish to change the compilation arguments or the names of the source files. **However, ensure that your Makefile generates the executable named** `distvec` **or/and** `linkstate`, depending on which one you choose to implement. Generally, the default combination of Makefile and source files is sufficient. **If you submit code for both DV and LS, we'll take the maximum score of the two programs.**
3. The Autograder will use the following format for command line input to your program:
   ```
   ./distvec topofile messagefile changesfile
   ./linkstate topofile messagefile changesfile
   ```
4. When compressing your submission, you should zip all related files, not the directory containing these files. On Linux/macOS, you can use the following commands:
   ```
   cd /path/to/your_mp3_code/
   rm mp3.zip
   zip -r mp3.zip *
   ```