CS/ECE 438: Computer Networks **Spring 2026**

Machine Problem 2

Handed Out: February 17th, 2026 *Due: 11:59pm, March* 29th, 2026

TA: Xiaojuan Ma

1 Introduction

In this MP, you will implement the link state and distance vector routing protocols. You will write two separate programs: one that implements the link state protocol, and one that implements the distance vector protocol. Both programs will read the same file formats to get the network's topology and what messages to send.

2 The Router Program

Your program should contain a collection of imaginary routing nodes that all carry out the same routing protocol (link state or distance vector, depending on the program). 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 Makefiles 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 Notes at the end of this document).

Your program should first read the input, which consists of a 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. Like the distance vector implementation, your link state implementation must explicitly propagate link information across the network. Your program then 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 format.

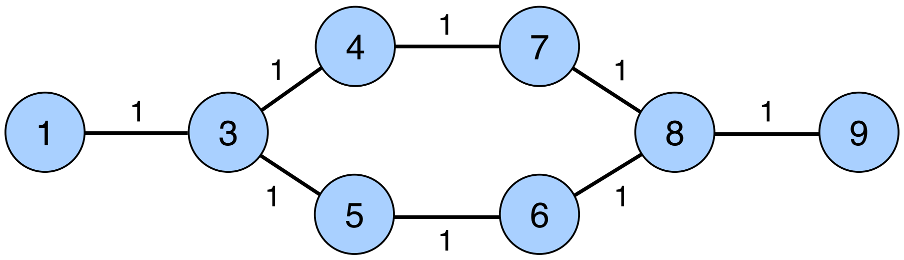
Then, one at a time, apply each line in the topology changes file (see below) in order. After each change, once the routing tables have re-converged, print out the forwarding tables and resend all of the messages. More specifically, because the topology has changed, your program will have to follow the (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 standard versions of routing algorithms, such as what we have discussed in the lecture videos. These algorithms choose a path to the destination with the smallest cost. However, we would like everyone to have a consistent output even on complex topologies, so we ask you to follow a specific tie-breaking policy. When a router R is determining how to select among multiple equal-cost paths to destination D:

1. **Distance Vector:** Choose the path for which the next-hop node (i.e., the node to which R will directly send packets) has the smallest node ID compared to the other possible next-hop nodes.
2. **Link State:** Choose the path with the lexicographically smaller sequence of node IDs from destination D to R.

For example, consider the following topology:



For a route from node 1 to 9, your Distance Vector routing implementation should select path $1 \rightarrow 3 \rightarrow 4 \rightarrow 7 \rightarrow 8 \rightarrow 9$ because while traversing the path from source to destination, $4 > 5$. In contrast, your Link State routing implementation should select path $1 \rightarrow 3 \rightarrow 5 \rightarrow 6 \rightarrow 8 \rightarrow 9$ because while traversing the path from destination to source, $6 < 7$.

Our DV tiebreaking rule is commonly used in real implementations of DV and Path Vector. (Deterministic outcomes are useful not only for our autograder, but also in real deployments!) In this assignment, we've chosen a different tiebreaking rule for LS because it better matches LS's computation of paths.

More specifically, DV builds paths from destination outward to sources, so to implement the tiebreaking rule, when updating the routing table, you can compare the next-hop node IDs of the equal-cost paths and choose the one with the smaller ID. LS, on the other hand, when using Dijkstra's algorithm, works from the source router R outward to all destinations. Thus, to implement the LS tiebreaking rule, when extending the shortest path tree, you can compare the previous-hop node IDs of the equal-cost paths and choose the one with the smaller ID.

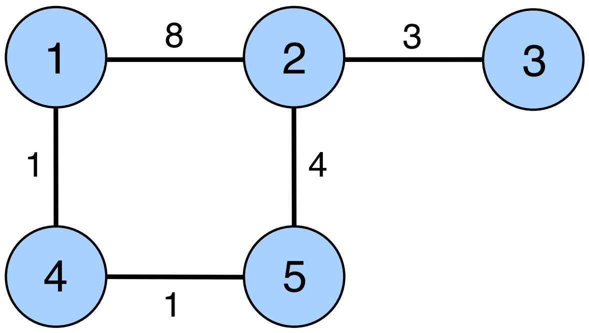


Figure 1: Sample Topology

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
```

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 structued 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, second 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

We require two files, detailed as follows:

5.1 output.txt

Write all output described in this section to a file called `output.txt`. The output file will have the general layout as follows. ***Note that the grayed out text is just commentary, not part of the output file.***

```
<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, 1st 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, 2nd 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 destination's packets to, and pathcost is the total cost of this path to 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 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.

5.2 all_messages.txt

Write all messages described in this section to a file called `all_messages.txt`. This file contains all messages nodes send among themselves to propagate topology changes. For distance vector routing, these are distance vector updates; for link state routing, these are link state advertisements. You are also required to write the actual data messages into this file.

The file will have the general layout as follows. ***Note that the grayed out text is just commentary, not part of the output file.***

```
<control message 1>
<control message 2>
<control message ...>
——— At this point, the routing tables have converged, start to send messages
<data message 1>
<data message 2>
<data message ...>
——— At this point, 1st change is applied
<control message 1>
<control message 2>
<control message ...>
——— At this point, the routing tables have converged, start to send messages
<data message 1>
<data message 2>
<data message ...>
——— At this point, 2nd change is applied
<control message 1>
```

```
<control message 2>
<control message ...>
——— At this point, the routing tables have converged, start to send messages
<data message 1>
<data message 2>
<data message ...>
——— And so on...
```

The format for a single message entry should be:
<message-type>: <timestamp (unix ns)>: <message content>
where **message-type** indicates the type of message: `CONTROL` for topology information propagation (distance vector updates or link state advertisements); `DATA` for actual messages that nodes send to each other after convergence.

Use system time in nanoseconds for timestamps. Messages should be logged with the timestamp at which they are sent.

CONTROL Messages

Control messages are used by nodes to learn the initial network topology and propagate topology updates. Log each control message that is sent during the protocol execution.

The format is: `CONTROL: <timestamp>: <your actual message content>`

where `<your actual message content>` is the actual messages being transmitted in your implementation. This should include at minimum:

- The sending node
- The receiving node
- The routing information being shared (distance vectors or link state data)

Example formats (your actual format may vary based on your implementation):

```
CONTROL: <timestamp>: <your actual control message>
```

The exact format of the message content depends on your implementation, but it should clearly show what routing information is being exchanged between nodes.

DATA Messages

Data messages are the actual messages sent between nodes from the messages file. The content after `DATA: <timestamp>:` must be **exactly** the same as what appears in `output.txt`.

The format is: `DATA: <timestamp>: <exact output.txt line>`

Example of data messages:

```
DATA: 1000000500: from 2 to 3 cost 6 hops 2 5 4 message hello
DATA: 1000000600: from 3 to 5 cost infinite hops unreachable message unreachable
```

Messages should be logged in chronological order based on their timestamps. Control messages appear during the initial topology learning phase and after each topology change. Data messages appear only after the routing tables have converged, between the convergence marker and the next change marker.

Both `messagefile` and `changesfile` can be empty. In this case, the program should only log the initial control messages for topology learning.

Please do not print anything else to this file; 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 blank lines in between sections.

6 Notes

All the notes for the previous MPs still apply. We are not repeating those here for brevity.

New information:

1. Your project must include a Makefile whose default target makes executables called `distvec` and `linkstate`
2. Command line format:
   ```
   ./distvec topofile messagefile changesfile
   ./linkstate topofile messagefile changesfile
   ```
3. You are required to implement both Link State (LS) and Distance Vector (DV). Your final score consists of 82 points from the autograder and 18 points from hidden tests, for a total of 100 points.

   Test cases evaluate different aspects of your implementation, including but not limited to: correctness over various topology sizes (simple ones, similar to examples in this document, or even larger networks), handling edge cost changes, link deletions (which may lead to unreachability), and link additions.

   The autograder tests both implementations separately on the same set of 9 test cases. The score breakdown is as follows:

   - **10 points:** Successful compilation producing both `linkstate` and `distvec` executables.
   - **4 points each $\times$ 9 tests (LS):** 36 points total for Link State implementation.
   - **4 points each $\times$ 9 tests (DV):** 36 points total for Distance Vector implementation.

   Note that if compilation fails, **the entire MP will receive a score of 0** as no tests will be run.

   Detailed per-test feedback including pass/fail results and your total autograder score will be available in the `_grades` branch of your GitHub repository after each submission.
4. You are expected to work in groups of 2 on this MP. The partner with the lexicographically earlier NetID should submit the group registration form via Microsoft Forms (one submission per group; see the Campuswire announcement for details). If you wish to work alone, you must first request an exception on Campuswire and receive approval from the course staff before proceeding.

5. We will grade the repository of the partner with the lexicographically earlier NetID. Both partners must work out of that single repository to avoid confusion. **If both partners submit separately, only the score from the lexicographically earlier NetID's repository will count.**
6. Your final grade will be based on your *last* submission.
7. You may form different groups for different MPs, but group changes within the same MP are not permitted after registration unless you explicitly request an exception on Campuswire and receive approval from the instructors.

Caution: Submission instructions are same as for prior MPs. During the last few hours leading of the submission deadline, queues could be multiple hours long. We advise you to get your work done early.

Please refer to prior MP instructions for other notes.