Project #3: Routing Lab

Northwestern CS340, Fall 2022

1 Before starting

- Project #3 should be done in groups of two or individually.
- Please carefully read the instructions including all references and footnotes.

2 Overview

In this project, your group of two will implement a distance-vector algorithm and a link-state algorithm in the context of a simple routing simulator. This will give you an understanding of how OSPF and BGP work. Your implementations can mirror the algorithms described in the book, and will consist of only a few dozen lines of code. It is important that you understand what is going on before you start, and that you write a pseudocode implementation or at least have a clear plan before trying to implement the code. While it will be tempting for you and your partners to each separately implement an algorithm, we believe it will be far easier if you collaborate on both algorithms. The experience of implementing one of the algorithms will make the implementation of the second algorithm much easier.

> *TIP: Techniques for broadcasting are covered in Lecture 12 and this is needed to implement the link-state algorithm (for sharing link states). Of the two algorithms, I think link-state is easier to implement, so we suggest you start there.*

3 Getting the Code

You will be able to work on any Python3 environment (including either on your own machine or on `moore.wot.eecs.northwestern.edu`, or on any of the Wilkinson lab machines).

> **Remote access tip***: The routing simulator has a graphical component, so if you choose to work on moore you will have to log in using a method that supports graphics. Actually, this a bit clunky, so I recommend that you run the code on your own machine if possible. That said, if you choose to run on moore, you should use the FastX remote login system by first connecting to the Northwestern VPN and then clicking this link: https://moore.wot.eecs.northwestern.edu:3300/*
>
> *Further instructions are here. FastX will give you a virtual Desktop on moore that you can disconnect from and reconnect to.*

Get a copy of the code using git

```
1 $ git clone https://github.com/northwestern-cs340/routesim2
```

This will create a routesim2 directory, containing a complete network routing simulator. For this project, you will just have to write the implementations for two different node classes in the files:

1. `distance_vector_node.py`
2. `link_state_node.py`

In other words, you will just have to write the routing algorithm. The network simulator itself is already written. As a prerequisite, you will have to:

```
$ pip install --user networkx matplotlib
```

If you still encounter error saying module not found, try this as well:

```
$ pip3 install --user networkx matplotlib
```

To execute routesim, do the following:

```
$ python3 sim.py GENERIC [eventfile]
```

For example:

```
$ python3 sim.py GENERIC demo.event
```

The parameter `GENERIC` will be replaced by `DISTANCE_VECTOR` or `LINK_STATE` when you are ready to test each of your two implementations.

We will say more about event files soon. The graphs are drawn automatically by the existing simulator code and if you implement the code correctly should see the routing path from a source node to a destination node in the displayed graph (in red). Routesim will pause every time it draws a graph, waiting for you to close the graph window.

4 Event-driven Simulation

Routesim is an event-driven simulator. What this means is that instead of simulating the passage of time directly, it instead jumps from event to event. For example, suppose a node decides to send a routing message to neighbor. If the current time is 100, and the link latency to the neighbor is 10, then instead of simulating time 100.1, 100.2, . . . , 109.9, 110, the simulator “posts” an event (the

arrival of the message at its neighbor) to occur at time 110. If there are no other events posted for times between 100 and 110, the simulator can jump ahead to 110.

Event-driven simulators are very powerful tools that are widely used in science and engineering. Routesim is implemented in the usual manner for event-driven simulators. There is a priority queue (implemented as a heap), called the event queue, which stores the events in time order. The simulator main loop simply repeatedly pulls the earliest event from the queue and passes it to a handler until there are no more events in the queue. The handler for an event may insert one or more new events into the event queue. For example, the handler for the routing message arrival may update the neighbor node's distance table and then post a new arrival event for its neighbor.

5 Events in Routesim

Events in routesim come from the event file, and from handlers that are executed in response to events. The events file contains both events that construct the network topology (the graph) as well as events that modify link characteristics in the graph, or draw the graph, a routing path, or a shortest paths tree. In the events files, lines that are blank or whose first character is a # are ignored.

Here are events that can occur in an events file:

```
arrival_time ADD_NODE node_num
arrival_time ADD_LINK src_node_num dest_node_num latency
arrival_time CHANGE_LINK src_node_num dest_node_num latency
arrival_time DELETE_LINK src_node_num dest_node_num
arrival_time DELETE_NODE node_num
arrival_time DRAW_TOPOLOGY
arrival_time DRAW_TREE src_node_num
arrival_time DRAW_PATH src_node_num dst_node_num
arrival_time DUMP_NODE node_num
```

Note that any DRAW event will cause a window to pop up with a drawing of the topology. The simulation will stall until you close the window.

We've included two demonstration event files (`demo.event` and `test1.event`) and you can generate random test examples using the *generate_simulation.py* script. Note that these files are very easy to write, and you may wish to do so on your own.

6 A Node of One's Own

To implement a routing algorithm in Routesim, you will write Node implementations in *distance_vector_node.py* and *link_state_node.py*. This code will be run for each and every node in the network. Thus you will be implementing two different distributed algorithms.

It's important that you do not add any additional "imports" in your code, except to import standard Python libraries. In other words, you may not import code from the simulator that would allow your node code to "cheat" and access global simulator data.

7 The Node Class

Node has four functions that you must implement for each algorithm:

1. `link_has_been_updated(neighbor, latency)` → `None`: is called to inform you that an outgoing link connected to your node has just changed its properties. It tells you that you can reach a certain neighbor (identified by an integer) with a certain latency. In response, you may want to update your tables and send further messages to your neighbors. This function does not have to return anything.
2. `process_incoming_routing_message(m)` → `None`: is called when a routing message "m" arrives at a node. This message would have been sent by a neighbor (more about how to do that later). The message is a string. In response, you may send further routing messages using `self.send_to_neighbors` or `self.send_to_neighbor`. You may also update your tables. This function does not have to return anything.
3. `get_next_hop(destination)` → `int`: is called when the simulator wants to know what your node currently thinks is the next hop on the path to the destination node. You should consult your routing table or whatever other mechanism you have devised and then return the correct next node for reaching the destination. This function should return an integer.
4. `__str__()` → `str`: is called to when the simulation wants to print a representation of the node's state for debugging. This is not essential, but it may be helpful to implement this function so that `DUMP_NODE` events print sensible information. This function should return a string.

Your implementation will consist of implementations of these four functions, as well as any additional functions that you need to help implement these. All of your code for this assignment should be in the two files *distance_vector_node.py* and *link_state_node.py*, and you should not change any other files.

8 Programming interface

Your implementations are subclasses of the Node class defined in *simulator/node.py*. In that file you will find a few simulator functions that you will need to use in your node subclass implementation.

1. `send_to_neighbor(neighbor, message)-> None`: sends a string routing message to a neighbor.
2. `send_to_neighbors(message)-> None`: sends a string routing message to **all** neighbors.
3. `get_time()-> int`: returns the current simulation time. This may or may not be needed.

Notice that link-state works by flooding, meaning that you'll want to flood a link update to all of your neighbors. Similarly, when a path gets updated in a distance vector algorithm, all neighbors need to be informed.

For the Link State algorithm, you will also need to implement Dijkstra's Algorithm to find the shortest path to nodes from a source node.

9 Representing routing messages

Generally, your code will represent the state of the network known to each node as Python objects, lists, dictionaries, sets, or a combination of the above. It's up to you to decide what data to store at each node and what data structures to use. Some of this information will be transmitted to neighbors in routing messages, and these routing messages must be plain strings.

We recommend that you the JSON format[1] for your routing messages. You may simply "import json" and then use `json.loads()` and `json.dumps()` to convert between JSON strings and Python objects.

You need to carefully think what will go inside these routing messages depending on the routing algorithm you are implementing. `LINK_STATE` routing messages are used to flood link information whereas `DISTANCE_VECTOR` routing messages communicate distance vectors.

10 General approach to implementing a routing algorithm

1. Run routesim with type `GENERIC` to see how it works.
2. Read `demo.event` and `test1.event` to learn the event file format.
3. Read the `generic_node.py`, `distance_vector_node.py` and `link_state_node.py` to understand what functions you'll be implementing.
4. Decide what data must be stored by each node in the node class, for the particular algorithm you are implementing. It should also be updatable, as its contents willchange with the arrival of link updates and routing messages.
5. Implement `get_next_hop()` to return the appropriate answer, based on what the node currently knows.
6. Develop your routing message. Think about where your routing message will go. There are subtleties involved in flooding, in particular.
7. Implement `link_has_been_updated()`. You must record the new information within the node, and (depending on the message contents) you may need to send messages to neighbors.
8. Implement `process_incoming_routing_message()`. As above, you must record the new information within the node, and (depending on the message contents) you may need to send messages to neighbors.

11 Link-state flooding

For link-state, you will need to implement controlled flooding to propagate information about link updates. When a link update happens, the simulator will call the `link_has_been_updated` method only on the two nodes on either side of the link, and it is that node's responsibility to start propagating that information.

- Link state messages will include: (link source, link destination, cost, *sequence number*)
- A link's sequence number is incremented whenever that link changes (`linkHasBeenUpdated`), and this includes when the link has been deleted.

[1] JSON documentation: https://docs.python.org/3/library/json.html

- The adjacent nodes of each link are responsible for incrementing the link's sequence number.
- The adjacent nodes are also responsible for constructing the LS message and broadcasting it. The receivers will just relay the flooding message, unmodified.
- The flooding will start identically from two sources (the two adjacent nodes), with the same message, but that's OK.
- Each node keeps a record of the "latest" LS message it received for each link (the one with the highest sequence number). If it receives a LS message with a new sequence number, then it broadcasts that message on the other links (not on the link it received the message on). If the sequence number is "old" (smaller than the highest it already has for that link), then it sends the newer LS message back to the node who sent the older message.
- By following these rules, you will ensure that every node receives the LS broadcast, and the flooding eventually terminates when everyone has the latest information.
- If a new node is added later in the simulation (using `ADD_NODE`) it will have missed prior broadcasts. It will have to somehow be brought up to date by its neighbors.
- Your network might have two or more "islands" of nodes that are suddenly joined by a new edge. Actually, the case above describing a new node joining the network is just a special case of two islands being joined. The two newly-joined islands must share their information, including past broadcasts.

12 Avoiding loops in Distance Vector

The simulator allows links to be deleted, and this can lead to a "count to infinity situation" in your DV algorithm if you are not careful. The `demo.event` file actually exhibits this problem (`test1.event` does not have this problem because no links are deleted). The problem arises when you have two or more nodes that are suddenly "cut off" from the rest of the network. They each will try routing through the other until the distance literally reaches infinity. This is unacceptable. The simple solution called "poisoned reverse" will not help if there are three or more nodes in the cut-off island.

To truly solve this problem, your distance vectors should include the full routing path for each destination (similar to the `AS_PATH` in BGP). In other words, each entry in the DV should also include a list (or set) of nodes that are involved in the path. This will allow nodes to avoid choosing routes that would form loops and this will prevent count to infinity.

TIP: Remember that Python assigns lists by reference. The following code will modify "a":

```
>>> a = [1, 2, 3]
>>> b = a
>>> b.insert(0, 4)
>>> print(a)
[4, 1, 2, 3]
```

The following makes a deep copy:

```
>>> import copy
>>> a = [1, 2, 3]
```

```
>>> b = copy.deepcopy(a)
>>> b.insert(0, 5)
>>> print(a)
[1, 2, 3]
>>> print(b)
[5, 1, 2, 3]
```

So, you must use a deep copy if you are constructing an `AS_PATH` based on other `AS_PATH`.

13 Out of order delivery of DV messages

When you send a message, it is not delivered until after the link latency time elapses. This can actually create out-of-order delivery of DVs if the link suddenly becomes faster. For example, consider the case when you send a DV to a neighbor on a link that has latency 10 and then one second later the latency becomes 5. The first DV is still scheduled to be delivered nine seconds from now, but if we send un updated DV now, it will be received earlier (five seconds from now). This is a somewhat unrealistic limitation in the simulator. To deal with this issues, you should add a sequence number (or just a timestamp) to DVs, so that you always keep the one that was **sent latest**, not necessarily received latest.

14 Suggestions and Hints

- Since there are two routing algorithms, it will be tempting for you and your partner(s) to each implement an algorithm independently. WE STRONGLY SUGGEST THAT YOU DO ***NOT*** DO THIS. It'll take some time to understand the framework, and that will be made a lot easier if the two of you work together. Furthermore, after you implement the first algorithm (regardless of whether it's DV or LS), you'll find the second to be a lot easier.
- The description of the DV algorithm given in **Lecture 11** is much simpler (in my opinion) than the description given in the book. Please review this lecture.
- Your implementations should not assume limits on the number of nodes and edges. On the other hand, it does not have to be blindingly fast. You may find it very useful to use lists and dictionaries.
- You should represent links using the **frozenset** Python object. A set allows you to represent a pair of vertices where the ordering does not matter (`set([1,2])== set([2,1])`). A `frozenset` is an *immutable* set, which allows it to be used as a dictionary key.
- A node knows nothing ahead of time. It does not know how many nodes are in the universe, nor anything else about the topology. It knows only what it learns from `link_has_been_updated()` and `process_incoming_routing_message()`. This mimics a real-world distributed system.
- Do not assume that all node numbers are contiguous. For example, nodes may be numbered 1, 2, 4, 6. Remember that nodes and links may be deleted throughout the simulation.
- For the link-state algorithm, a simple implementation of Djikstra's algorithm without a priority queue is fine.

- Keep it simple. You won't be graded on CPU performance. It is possible to write each algorithm in 100 lines of code or so. However, do not sent more message than is necessary. The simulator reports the number of messages sent when it's done. Use the `DRAW_TREE` event to test your results. `DRAW_TREE` will show the correct solution in green and your solution in red. By the time your code is done, the green and red trees should match, or at least they should be equidistant alternatives!
- Whenever a graphical plot is displayed on your screen, it is also saved to a png image file in the output directory.
- `test2.event` is a large (100-node) test case. You should be able to run this in a reasonable amount of time. This test case ran in one minute on my laptop using the DV algorithm (sending about 90,000 messages total). On the same machine using the LS algorithm the runtime was 7 seconds (and it required about 200,000 messages). If your code is too slow but your number of messages is OK, then we suggest you use the profiler to determine where your code is spending the most time. Run your code on the command line with the following extra parameters:

```
python3 -m cProfile -s cumtime sim.py DISTANCE_VECTOR test2.
    event
```

  Scroll to the top of the output and look for the functions that have the most cumulative time. For example, this might reveal that a lot of time is spent in json serialization or in making unnecessary copies of lists.
- In addition to the test cases in the repo, please test your code on the following test cases, which are in the public files repository of course (click to go to the GitHub repo!):
  1. Download case_1.event
  2. Download case_2.event
  3. Download case_3.event
  4. Download case_4.event
  5. Download case_5.event
  6. Download case_6.event
  7. Download case_7.event
  8. Download case_8.event
  9. Download case_9.event
  10. Download case_10.event

15 Submission guidelines

- Your code must run on Python 3.6 on `moore.wot.eecs.northwestern.edu`.
- Your submission should be a **.tgz** file (a gzipped tarball) including three files: `distance_vector_node.py`, `link_state_node.py`, and a `README`. The README should include the names and net IDs of the project team, a brief description of work undertaken by each member:

```
$ tar -czvf project3_NETID1_NETID2.tgz README.txt
    distance_vector_node.py link_state_node.py
```

Please make sure to include both partners' net IDs in the filename!

- Remember that you should leave a comment explaining any bits of code that you copied from the Internet (just in case two groups copy the same code from the Internet, we won't think that you copied each other directly).
- Just one of the two teammates should submit. If your teammate submitted, then you can leave a comment identifying your teammate.