Problem Set 8

Your answers will be graded by actual human beings (at least that's what we believe!), so don't limit your answers to machine-gradable responses. Some of the questions specifically ask for explanations; regardless, it's **always a good idea to provide a short explanation for your answer**.

Please read Chapters 16, 17 and 18 **before** solving these problems (and you may find it useful to consult these while solving them too). Please also solve the online practice problems on network routing and the problems at the end of Chapters 16 (but you can ignore Little's law for now; we'll pick that up in the next PSet), 17 and 18.

Problem 1: Hunting in (packet) pairs

This question will assess how well you understand the different sources of delay on a network link (see the first part of Section 16.4 in Chapter 16). A sender S and receiver R are connected using a link with an unknown bit rate of C bits per second and an unknown propagation delay of D seconds. At time t=0, S schedules the transmission of a **pair of packets** over the link. The bits of the two packets reach R in succession, spaced by a time determined by C. Each packet has the same known size, L bits.

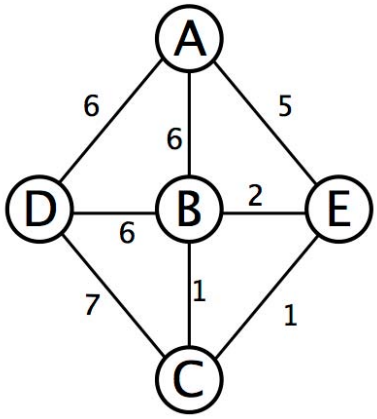
The last bit of the first packet reaches R at a known time t=T_1 seconds. The last bit of the second packet reaches R at a known time t=T_2 seconds. This packet pair method allows us to estimate the unknown parameters, C and D, of the path. Determine C and D in terms of L,

T_1, and T_2.

(points: 2)

Problem 2: Link-state in Alyssa's network

Alyssa P. Hacker runs the 6.02 link-state routing protocol in the network shown below. Each node runs Dijkstra's algorithm to compute minimum-cost routes to all other destinations, breaking ties arbitrarily. Answer the following questions, explaining each answer in the space provided.



A. In what order does C add destinations to its routing table in its execution of Dijkstra's algorithm? Give all possible answers.

(points: 1)

B. Suppose the cost of link CB increases. What is the largest value it can increase to, before **forcing** a change to any of the routes in the network? (On a tie, the old route remains.)

(points: 1)

C. Assume that no link-state advertisement (LSA) packets are lost on any link. When C generates a new LSA, how many copies of that LSA end up getting flooded in total over all the links of this network, using the 6.02 link-state flooding protocol?

(points: 1)

Problem 3. ChainNet

Each node in the chain-like network topology shown below runs a distance vector protocol, sending routing information to its neighbors, according to these rules:

Rule 1. Send the first distance-vector advertisement immediately after power up.

Rule 2. Every 200 seconds thereafter, send a distance-vector advertisement containing the cost for each known destination. That is, if a node powers on at time t, it sends advertisements at times t, t+200, t+400, t+600, ...

Rule 3. As soon as a route advertisement is heard, update the route and corresponding cost if, and only if: (a) the new cost is smaller than the current one, OR (b) the cost of the current route changes.

Once sent, the delay before a packet is received at a neighbor is negligible. Unless mentioned otherwise, assume that no route advertisements are lost between nodes that are on. Each link has cost 1.

In this network, there is no HELLO protocol in place. Each node knows the identity of each of its neghbors, but **does not** know if the neighbor is alive or not. Each node learns whether a neighbor is alive or not only by receiving an appropriate distance-vector advertisement.

Initially, all nodes are off. Node A powers on at time t=0, node B at t=3, node C at t=6, and node D at t=9 seconds.

A. At what time will node A have correct routing information about all the other nodes in the network? And at what time will node D have correct routing information for all the other nodes in the network? Give a brief explanation.

(points: 1)

To handle failures in this network, each node implements the following rule in addition to the three mentioned earlier:

> **Rule 4.** Immediately upon detecting the inability to reach a directly connected neighbor, update the cost to each destination in the routing table that uses the corresponding link to INFINITY (set to 100).

Suppose every node has found a correct route to every other node. The link between C and D fails and C detects the failure at some arbitrary time T. There are no other failures and the link doesn't come back up. Each packet takes zero time to be sent along a link and be processed at a node. Packets don't carry a ``hop limit'' (``time to live'') field.

For each of the following three statements (B, C, D), state whether it is true or not. Explain each response briefly.

B. Given enough time, the routing protocol converges, with every node having the correct cost for every destination.

(points: 1.5)

C. There exists some time T such that a packet sent from A to D at some time after T bounces back-and-forth between B and C.

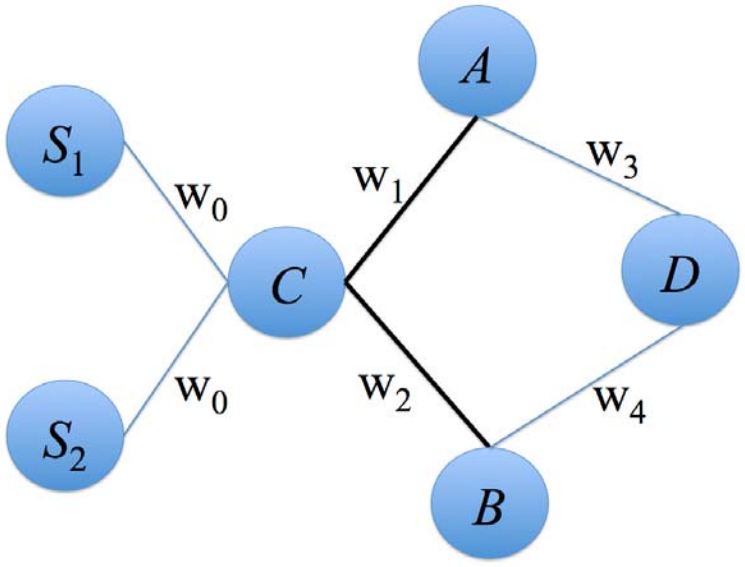
(points: 1)

D. There exists some time T such that a packet sent from A to D at some time after T bounces back-and-forth between A and B.

(points: 1.5)

Problem 4. FishNet

Ben Bitdiddle is responsible for routing in FishNet, shown below. He gets to pick the costs for the different links (w0, w1, w2, w3, and w4 shown near the links). All the costs are non-negative integers.



Goal: To ensure that the links connecting C to A and C to B, shown as darker lines, carry *equal traffic load*. All the traffic is generated by S1 and S2, in some unknown proportion. The rate (offered load) at which S1 and S2 together generate traffic for destinations A, B and D are R_A, R_B, and R_D, respectively. Each network link has a bandwidth higher than R_A + R_B + R_D. There are no failures.

Protocol: FishNet uses link-state routing; each node runs Dijkstra's algorithm to pick minimum-cost routes. If there two nodes at any step that have the same min cost, assume the choice is made arbitrarily.

Suppose R_A + R_D = R_B. For the following questions, choose weights that *guarantee* that there will be equal traffic on LinkCA and LinkCB, regardless of how ties are resolved when picking the minimum cost routes. (Note that all link costs are non-negative integers.)

A. Given the following link costs, what is the largest possible value for w1?
   w0 = 1
   w2 = 4
   w3 = 2
   w4 = 2

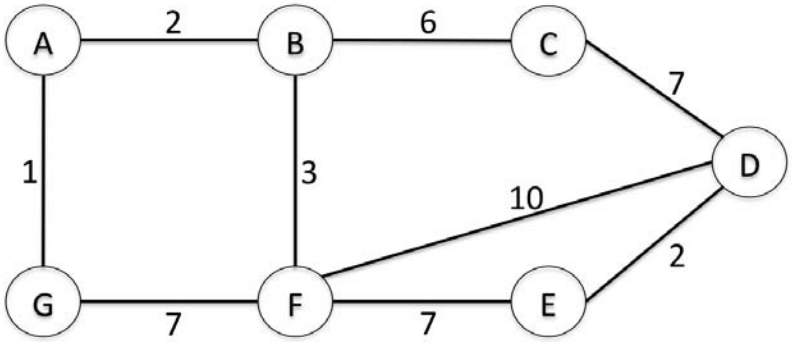
(points: .5)

B. Given the following link costs, what are the smallest and largest possible values for w2?

w0 = 1
w1 = 5
w3 = 3
w4 = 3

(points: .5)

Problem 5. Help Ben Bitdiddle answer these questions about the distance-vector protocol he runs on the network shown below. The link costs are shown near each link. Ben is interested in minimum-cost routes to destination node D.



Each node sends a distance-vector advertisement to all its neighbors at times 0, T, 2T, ... Each node integrates advertisements at times T/2, 3T/2, 5T/2, ... You may assume that all clocks are synchronized. The time to transmit an advertisement over a link is negligible. There are no failures or packet losses. At each node, a route for destination D is **valid** if packets using that route will eventually reach D. At each node, a route for destination D is **correct** if packets using that route will eventually reach D along some minimum-cost path.

A. At what time will **all** nodes have integrated a valid route to D into their routing tables? What node is the **last one** to integrate a valid route to D? Answer both questions.

(points: 1)

B. At what time will **all** nodes have integrated a correct (minimum-cost) route to D into their routing tables? What node is the **last one** to integrate a correct route to D? Answer both questions.

(points: 1)

Problem 6.

Alyssa P. Hacker has set up a 6-node connected network topology in her home, with nodes named A, B, ..., F. Inspecting A's routing table, she finds that some entries have been mysteriously erased (shown with ``?'' below), but she finds some other correct entries in it. A's routing table is:

Destination	Path cost	Link
B	3	LinkAC
C	2	?
D	4	LinkAE
E	2	?
F	1	?

Each link has a cost of either 1 or 2 and link costs are symmetric (the cost from link X to link Y is the same as the cost from Y to X). The routing table entries correspond to minimum-cost routes.

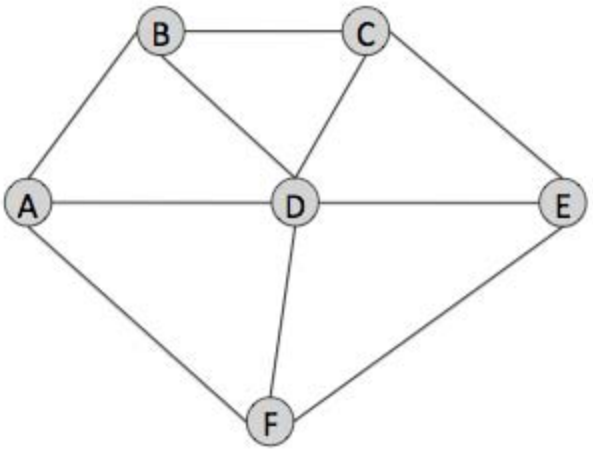
She knows that there could be other links in the topology. To find out, she inspects D's routing table, but it is mysteriously empty (and it shouldn't be).

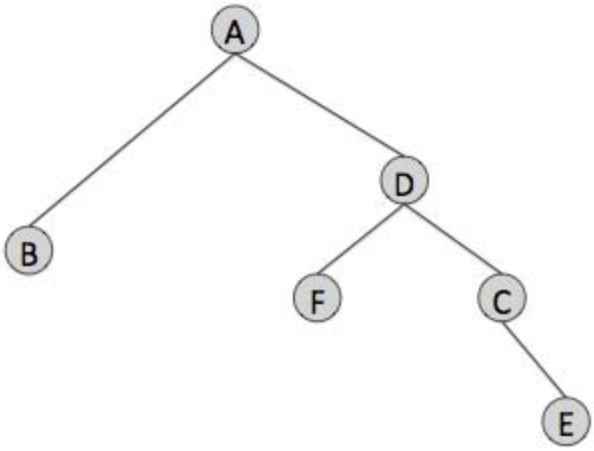
What is the smallest **possible** value for the cost of the path from D to F in Alyssa's network topology? Assume that any two nodes may possibly be directly connected to answer this question. (Hint: It will help to construct the topology on paper using the information given.)

(points: 1)

Problem 7.

Alyssa P. Hacker is trying to reverse engineer the trees produced by running Dijkstra's shortest paths algorithm at the nodes in the network shown in the picture on the left, below. She doesn't know the link costs, but knows that they are all positive. All link costs are symmetric (the same in both directions). She also knows that there is exactly one minimum-cost path between any pair of nodes in this network.





She discovers that the routing tree computed by Dijkstra's algorithm at node A looks like the picture on the right, above. Note that the exact order in which the nodes get added in Dijkstra's algorithm is not obvious from this picture.

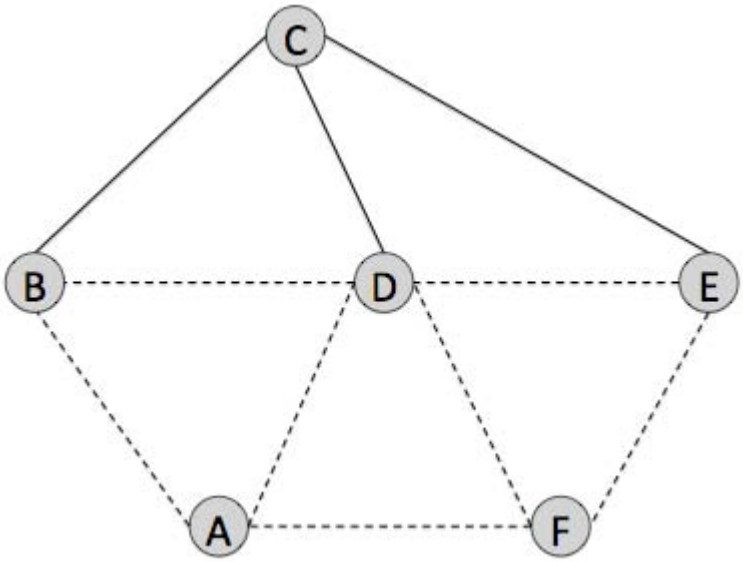
A. Which of A's links has the highest cost? If there could be more than one, tell us what they are.

(points: .5)

B. Which of A's links has the lowest cost? If there could be more than one, tell us what they are.

(points: .5)

Alyssa now inspects node C, and finds that it looks like the picture below. She is sure that the bold (not dashed) links belong to the shortest path tree from node C, but is not sure of the dashed links.



C. List all the dotted links that are *guaranteed* to be on the routing tree at node C.

(points: .5)

D. List all the dotted links that are *guaranteed* not to be (i.e., surely not) on the routing tree at node C.

(points: 1)

Introduction to the Python tasks

Useful download link for all the Python files: PS8.zip

This lab uses NetSim, a simple packet-level network simulator developed for 6.02. You will write the code for the main components of distance-vector and link-state routing protocols.

NetSim executes a set of steps every time slot; time increments by 1 each slot. During each time slot a link can deliver one packet from the node at one end of the link to the node at the other end of the link.

You can run the Python programs for this lab using python; this lab will not work in IDLE.

To understand the different parameters one can set in NetSim, go to a shell (terminal window)

and enter:

```
python PS8_dv.py -h
```

This command prints out the various options:

```
  -g, --gui                             show GUI
  -c, --checkoff                        checkoff the lab task
  -n NUMNODES, --numnodes=NUMNODES      number of nodes
  -t SIMTIME, --simtime=SIMTIME         simulation time
  -r, --rand                            use randomly generated topology
```

The `-r` option generates a random topology with the specified number of nodes. The default number of nodes is 12 and the default simulation time is 2000 time slots; you can change both using the corresponding command-line options.

This lab has two main tasks, each with a few sub-tasks. The first task is to implement a distance-vector (DV) routing protocol. The second task is to implement a link-state (LS) routing protocol. You may find it useful to debug your code using the `-r` option with randomly generated topologies, and it may be useful for you to use the `-g` option during the debugging process.

If you do not use the `-g` option, the code will run against a set of test topologies, which include static and changing costs as well as link failures. Each test carries a certain number of points.

Both routing protocols should construct the *minimum cost path* from the Router to all the other destination addresses that are currently reachable in the network. The destination is derived from the Router class and has an `address` field that will be used as an index into the routing table and the cost table. Unless explicitly mentioned otherwise, we will use the term "minimum-cost path" and "shortest path" interchangeably.

Useful classes and data structures

The Router class: The logic for DV resides in the DVRouter class; the logic for LS is in the LSRouter class. Both these classes are derived from the Router class, defined in PS8_netsim.py. The main goal of your software is to construct and maintain two key pieces of information in the DVRouter and LSRouter classes: *the routing table* and *the cost of the minimum-cost path* from the Router to the various destination addresses. That is, your code should correctly produce and maintain:

- `self.routes`, the routing table: a dictionary that maps from a destination address to a `Link`. This link is the link that the Router will use to forward any packet destined for the corresponding destination address.
- `self.spcost`, the table of costs of the shortest (i.e., minimum cost) paths: A dictionary that maps from a destination address to the current estimate of the cost of the path to get there.

The Link class: This class is defined in PS8_netsim.py; you don't need to modify it. But you should know that you can obtain the cost of a link, `L`, using `L.cost` (a variable in class `Link`). The other place you'll use the Link class is to populate the routing table, which (by definition) stores the Link to be used to reach any destination. We have

provided a useful function in the Router class, `self.getlink(n)`, which takes a neighboring Router, `n`, and returns the Link connecting `self` to `n`. This function is often useful in constructing the routing table. **Note that getlink(n) will not work as such, since getlink is a method of the class Router and not a usual function. You must call it using self.getlink()**

The HELLO protocol and maintaining live neighbors

We have already implemented the HELLO protocol for you in PS8_netsim.py (you should not need to modify this file); each Router sends a `HELLO` packet every `HELLO_INTERVAL` time slots (10 by default in NetSim). Whenever a node hears a `HELLO` packet along a link, it adds the current time, the address of the Router at the other end of the link, and the cost of that link, to `self.neighbors`, which is a dictionary mapping a Link to a `(timestamp, address, linkcost)` tuple.

If a node does not hear a `HELLO` for more than `2*HELLO_INTERVAL` time slots on a link, it removes that node from `self.neighbors` and calls the Router's `link_failed(link)` function, giving the "failed" link as argument. In response, your routing protocol implementation *may* take suitable actions.

Debugging and Testing Procedures Writing distributed protocols, even in simulation, can be a challenge. To help you a bit, we have some utilities that are accessible via the GUI (`-g` command-line option). By clicking on any node while the simulation is running, you can look at its routes and shortest path costs to every destination. For link-state routing, clicking on a node also prints out the last LSA information available at that node from each of the other nodes.

Please note that clicking on any link toggles the state of the link between "working" and "failed" states, letting you test your protocol under link failures. We will want to ensure that your protocol works properly in the face of failures and recoveries.

The "Step 1", "Step 10", and "Step 100" buttons are the way in which you should step through the operation of your protocol and see what is happening by clicking on various nodes.

In any given time-slot, colored squares may appear on a link. These are packets. The packets are color-coded: green ones are HELLO packets, red are advertisements (type ADVERT), and blue are data packets. (The blue data packets aren't relevant to this lab.)

For both the routing protocols you will implement in this lab, you should be prepared to demonstrate the correctness of the routing tables at the various nodes in the following scenarios listed below. Test your protocols with the `-r` option, which will generate random topologies. And make sure to test it on the test topologies and conditions we have provided (accessible when you don't use the `-g` option).

1. *No link failures.* All routes computed at the various nodes must actually correspond to the shortest paths in the graph. In addition, every node must have a routing table entry for all other nodes in the topology.
    - *Metric topology:* The link costs are for a topology where the costs satisfy the

metric property, i.e., one that satisfies the triangle inequality (cost(AB) + cost(BC) > cost(AC) for all triples of links AB, BC, AC.
    - *Non-metric topology*. The topology does not satisfy the triangle inequality.

2. *One or more link failures, but a connected network.* The routing protocol should be able to adjust its routes to route packets "around" the failed link. You should be able to give a rough estimate of how long link state and distance vector protocols take to find the new routes. You should also bring the failed link back up and verify that the routes go back to what they looked like in the base case.

3. *Dynamic cost changes and failures.* After the previous test has converged, change some link costs and introduce new failures to see if the protocol computes routes correctly.

4. *Convergence time.* This test configures some changes to link costs (both increase and decrease of costs) and sees whether the protocol converges fast enough.

5. *Disconnected topology.* We break multiple links in the topology such that some subset of the nodes cannot reach the other subset. You must show that your routing protocol eventually converges to the correct routing in this case -- that is, a node must have a routing table entry to every node it is connected to by a path in the underlying topology, and must not have a routing table entry for any node it cannot reach. You should also be able to provide a rough estimate of how long this convergence process takes for the two routing protocols by thinking about it and from your observations of the protocol in simulation. For distance vector routing, you should also demonstrate the "count to infinity" problem during convergence. Finally, when you "heal" one or more links to produce a connected toplogy, the protocol should eventually ensure that all nodes find the correct routes to all destinations.

Python Task #1: Distance vector (DV) routing

Useful download links:

PS8_netsim.py -- network simulator
PS8_tests.py -- testing functions
PS8_dv.py -- template file for this task

The file you will have to extend is PS8_dv.py.

This file contains the `DVRouter` class, which is derived from the `Router` class (which in turn derives from the `Node` class defined in PS8_netsim.py. Your first task for this lab is to write the following *three* functions, which are the core of any DV protocol:

1. `make_dv_advertisement()`: Scan the `self.routes` and `self.spcost` tables and construct a list of `[(dest1, cost1), (dest2, cost2) ...]`. Return this list.

   As explained in lecture and in Chapter 17, each router in a DV protocol periodically exchanges *routing advertisements* with its neighbors in the network topology, containing the information about destinations and their shortest-path costs

([(dest1,cost1), (dest2,cost2), ...]). This function will be called every ADVERT_INTERVAL time slots (50 by default in NetSim) by send_advertisement(), which will take care of constructing packets with this list as its contents, and sending one such packet to each neighbor in the topology.

2. link_failed(link): Called when the HELLO protocol determines a failure. When called, your code needs to take suitable action to recognize that the link is now "dead". For example, depending on how you design your DV protocol, you may: set self.spcost for all destinations whose routing table entries currently use that link to self.INFINITY (note: self.INFINITY, not just INFINITY), delete that route from your table, or anything else.

   We are intentionally not specifying the precise behavior, leaving it to you to design it. As long as what you do in this step is consistent with what you do in make_dv_advertisement(), your protocol will work correctly. Conversely, an inconsistency will likely cause the protocol to be incorrect.

3. integrate(link, adv): This function is where the actual distributed computation occurs. It takes as input two arguments: the link which deliverted the advertisement (link), and the advertisement itself (adv), which you constructed as a list in make_dv_advertisement. (You can ignore the marshalling of the advertisement into a packet and the corresponding unmarshalling back to the list, but if you're curious, you can see how send_advertisement() and process_advertisement() do these tasks.) You can use link.cost to determine the cost associated with the link.

   The result of integrate() is the current self.routes and self.spcost tables, which as mentioned before, are the routing table and table of shortest path costs. The underlying rule you should use is the Bellman-Ford update rule: remember to add the link cost to the cost reported by the advertisement that comes from the neighbor at the other end of the link.

   The integrate() step should take care to update the current route for a destination under the following conditions:

   a. If the cost in an advertisement for the destination plus the link cost is smaller than the cost of the current route.

   b. If the cost in an advertisement for the destination changes, and the advertisement comes on the link corresponding to the current-best route.

   c. Depending on how you design your DV protocol, you may have to take care of a subtle (but important) issue in integrate(): if you find that a previous advertisement that came from fromnode contained a destination, and you are using the corresponding link as the route to the destination, and the current advertisement *does not* mention the destination, then you have to assume that the destination is *no longer reachable via* `fromnode`. Otherwise, it is likely that your protocol may not be correct. You should also note that not every design requires this check; it all boils down to how you send your routing advertisements.

Please note that if your protocol decides that there is no route to a destination, then the route to that destination must be set to `None` and the `spcost` for that destination should be set to `self.INFINITY`. Failure to do so may cause some of the tests to fail.

You are NOT required to implement mechanisms to alleviate or eliminate the "counting-to-infinity" problem, such as split-horizon, poison-reverse, or even path vector, but are welcome to do so if you like!

When first debugging your code, it's useful to use the `-g` option so you can run the simulation for a small number of timesteps and then click on nodes to see their routing tables. When you're ready to see if your code passes the tests, run the program without the `-g` option. Please note that the verification is not exhaustive: passing these tests does not necessarily imply that your code is 100% correct!

Upload your code here. During check-off, you will be graded on how well your code handles the test cases, so you will run and explain it to your interviewer.

Upload code for Task 1: Browse...

(points: 5)

Python Task #2: Link-state (LS) routing

Useful download link:

PS8_ls.py -- template file for this task

The LS protocol uses the `LSRouter` class, which is derived from the `Router` class. The `self.routes` and `self.spcost` tables are identical to the DV case. The `LSRouter` class adds two new variables to Router:

- `self.LSA`, a dictionary that maps a Router address to a list `[seqnum, (n1,c1), (n2,c2) (n3,c3), ...]`, where the Router address is the *originator* of the link-state advertisement (LSA), `seqnum` is the current sequence number of the LSA from that node, and the `(n_i,c_i)` tuples are the currently live neighbor address and link cost from the LSRouter sending the LSA. Note that this LSA does not have a "origin_address" field; we simply get that from the source field in the packet, and you don't need to worry about it (in practice, implementations will explicitly include such a field in the LSA). Note that this dictionary contains the set of nodes and links in the graph known to the node.

  `self.LSA.get(u)` returns the last LSA update originating from Router `u` that this Router (`self`) knows about. It has the format `[seqnum, (n1,c1), (n2,c2), ...]` where `seqnum` is the sequence number at `u` when it originated the LSA and each `n_i` is a neighbor of `u` (that `u`'s `HELLO` protocol considered to be "live" when it generated the LSA numbered `seqnum`), and `c_i` is the cost of the link from `u` to `n_i`.

- `self.LSA_seqnum`, which is the sequence number for the LSA generated by the Router. It increments by 1 on each successive advertisement.

See Chapter 17 (and 18) for the details of how the LS protocol works. Each Router periodically sends its currenly live links (which we maintain in the `self.neighbors` dictionary, as explained earlier when we discussed the HELLO protocol). Each Router also *re-broadcasts*, along all its links, an LSA packet that it receives via a neighboring Router, containing an LSA originating at some other Router. This re-broadcast is done only once per LSA; to ensure this property, the Router checks the sequence number of the incoming LSA to make sure that it is larger than the last LSA heard from that originating Router. These periodic LSA broadcasts are done every `ADVERT_INTERVAL` time slots (50 by default in NetSim); the re-broadcasts are done when a Router receives a new LSA (using the sequence number check).

For this task, the file you have to extend is PS8_ls.py:

Your task is to write the following two functions, which form the core of any LS protocol:

1. `make_ls_advertisement()`: consult the list of neighbors, `self.neighbors`, that are currently "live" and return a list of `[(neighbor1,linkcost1), (neighbor2,linkcost2), (neighbor3,linkcost3),...]`. This function is called by `send_lsa`, which marshalls this LSA into the packets, and then sends one packet each along each link.

   Note that the `self.neighbors` dictionary mapping a `Link` to a `(timestamp, address, linkcost)` tuple will be useful in constructing the LSA; as mentioned above in the discussion of the `HELLO` protocol, `neighbors` keeps track of currently "live" neighboring Routers.

2. `run_dijkstra(nodes)`: Use Dijkstra's algorithm to produce `self.routes[dest]` for all `dest` in `nodes`, as well as `self.spcost[dest]`. The topology information for the network (graph) is available in the `self.LSA`, whose format was described above. The set of `nodes` currently reachable from the Router is passed as the argument to `run_dijkstra()`.

   There is one important issue that you need to watch out for in the steps of `run_dijkstra()` that will set the routing table entries, `self.routes`, for various destinations. At the Router, as you go through the different destinations in non-decreasing order of shortest-path costs and set the route to a node to be that of its parent in the shortest path tree. If the parent is `self.address` (i.e., the Router running the algorithm), then you should remember to set the route to the link connecting the Router (`self`) to the destination. You can use the `self.getlink()` method for this purpose.

Please note that if your protocol decides that there is no route to a destination, then the route to that destination must be set to `None` and the `spcost` for that destination should be set to self.INFINITY. Failure to do so may cause some of the tests to fail.

When first debugging your code, it's useful to use the `-g` option so you can run the simulation

for a small number of timesteps and then click on nodes to see their routing tables. When you're ready to see if your code passes the tests, run the program without the `-g` option. Please note that the verification is not exhaustive: passing these tests does not necessarily imply that your code is 100% correct!

Upload your code here. During check-off, you will be graded on how well your code handles the test cases, so you will run and explain it to your interviewer.

Upload code for Task 2: Browse...

(points: 5)

Suppose that a link failure has just occurred in the network. Give a rough estimate for how long it will take for the distance vector (Task #1) and link state (Task #2) protocols to correctly update the routing tables in *all* the nodes. Briefly explain your answer. Give your answer in terms of the HELLO and ADVERT intervals, not a numeric value.

(points: 1)

If the link failure causes the network topology to become disconnected, will your answer to the question above change? Explain.

(points: 1)

In testing a network path with very high cost:

```
A-----B-------------------------C-----D
   1      self.INFINITY-2           1
```

a correct implementation of the distance-vector implementation says that there is no route at node A for destination D, and vice versa, but in fact there is a path between these two nodes. Why does the protocol say that?

(points: 1)

6.02 Introduction to EECS II: Digital Communication Systems

Fall 2012