Introduction

The next two labs, Lab 10 and Lab 11, will use a simple network simulator written in Python. Python is an easy-to-learn interpreted programming language that has a nice selection of data types including support for object-oriented programming, interesting control structures, and a huge (and growing) set of libraries that make it simple to implement almost any processing task.

If you’d like to try out the simulator on your own machine you’ll need to install:

Python from [http://www.python.org/download/](http://www.python.org/download/)

There are pre-built binaries for Windows, Mac OS X and Linux.

For a comprehensive introduction to Python, we recommend the Python Tutorial written by Pythons architect Guido von Rossum:

[http://docs.python.org/tut/tut.html](http://docs.python.org/tut/tut.html)

The following sections of the Python Tutorial are particularly useful for what well be doing:

1. Section 3: An Informal Introduction to Python
2. Section 4.1: if Statements
3. Section 4.2: for Statements
4. Section 4.6: Defining Functions
5. Section 5.1: More on Lists
6. Section 5.5: Dictionaries
7. Section 9.3: A First Look at Classes

Lab Setup

In the two labs we’ll use the IDLE integrated programming environment to develop and run our Python programs.\(^1\) You can start IDLE by running setup 6.082 and then idle2.5 at the Athena prompt.\(^2\) Use File—Open to open a source file for editing; hit the F5 key to execute the current editor buffer.

Let’s first setup our Lab 10 environment:

\(^1\)IDLE isn’t required; it’s just a bit more convenient.
\(^2\)Our version of IDLE is currently only available under Linux; you will need to come into lab to run it.
In what follows, we will refer to your version of the code as `lab10.py`. After running `setup 6.082`, `idle2.5` and `python2.5` should both be in your path.

1 Overview of the Network Simulator

The simple network simulator is implemented by the class definitions found in `network.py` and includes a graphical front end. If you read `lab10.py` into IDLE and hit F5, you'll get a window that looks like:

The top part of the window shows a map of the network (i.e., the graph), which contains:

- **Nodes (black squares)**: These are instances of the Node class and represent named network nodes.

- **Links (black lines between nodes)**: These are instances of the Link class and represent bidirectional communication links between the nodes at either end of the link. Each link has two queues to hold the packets flowing in each direction.

- **Packets (colored circles)**: These are instances of the Packet class and represent the messages that will be sent from a source node to a destination node.

Underneath the map is a row of clickable buttons that control the simulation:

- **Reset**: return the network to its initial state. As we’ll see below, one usually provides a reset method for the network that performs the desired initialization.

- **Step N**: advance the simulation by 1, 10 or 100 time steps.

A status bar appears at the bottom of the window:

- **Time**: Current simulated time.

- **Pending**: A count of packets in the link queues and the transmit queues of the nodes. When this goes to 0 the simulation is over.

- **Total**: The total number of packets that have been created during the simulation. This is a useful measure of the total amount of message traffic since the simulation began.

- **Status**: When you move the mouse near a node or packet on the map the appropriate status message appears here. You can provide a status method for these objects that will display any information that you deem useful.
Notice that the network being simulated is actually an instance of the MyCostNetwork class, which is also defined in lab10.py. Actually we define two new classes: MyNode, which is a subclass of Node, and MyCostNetwork, which is a subclass of Network.

By defining our own classes we can modify the default behaviors of the Node and Network classes. More specifically:

1. The MyNode class defines its own method (forward) for forwarding packets, overriding the default forwarding behavior provided by the Node class. By editing this method you can change how a node chooses which link to use to send a packet towards its destination.
2. A new make_node method is defined for MyCostNetwork—when asked to create a new node instance, it will create instances of MyNode.
3. A new reset method is defined for MyCostNetwork, which, after calling the reset method for the parent class to perform the usual initializations, inserts a single packet into the network, choosing a random source and destination.

1.1 How the Simulation Works

The simulator simulates how packets flow through the network, traveling from node to node via the interconnecting links. Here’s what happens in a single time step:

1. The simulator calls the phase1 method of each node, which in turn calls the receive method of each link connected to the node, which returns the first packet that was queued on the link for delivery to the node.
2. The simulator then calls the phase2 method of each node to process the packets collected from each link during phase 1. Processing proceeds in two sub-steps:
   (a) Call the node’s process method for each packet, which in turn calls either the receive method if the node is the packets destination, or the forward method if the packet should be passed on to a neighbor for eventual delivery to its final destination. The forward method chooses a link to use, then calls that links send method to actually add the packet to the appropriate queue within the link.
   (b) After all the incoming packets have been processed, call the node’s transmit method, which can produce new packets to be delivered by the network. The default transmit method maintains a queue of packets that are introduced into the network when the packets declared start time is reached. The transmit queue is filled by calls to the nodes add_packet method during network initialization.

In real-world networks, the only information a node has about other nodes in the network is information that has arrived in packets. The following features of the simulator are worth understanding for this lab:

- The source and destination fields of each packet contain addresses. An address is a string that uniquely identifies each node in the network.
- Link.receive returns a 2-tuple (link, packet), so each incoming packet can be associated with the link it arrived on.
- Node has some useful instance variables:
  1. self.neighbors: a dictionary that maps a link to a 2-tuple.
2. **self.LSA**: a dictionary that maps an address to an N-tuple.
3. **self.LSA_seqnum**: an integer used to uniquely identify LSA broadcasts.
4. **self.routes**: a dictionary that maps a (destination) address to a link, which is the link that will be used to forward packets to that destination.
5. **self.HELLO_INTERVAL**: the number of time steps between HELLO packets.
6. **self.LSA_INTERVAL**: the number of time steps between LSA broadcasts.

- **Node.process(self, p, link, time)** is called with the incoming packet, the current time, and the incoming link.
- **Node.forward(self, p)** uses the packets destination address to consult the **self.routes** dictionary to determine which outgoing link to use when forwarding the packet.
- You can click on a node to invoke that nodes **OnClick** method. **MyNodes** implementation for that method prints out the contents of the nodes neighbors, LSA, and routes dictionaries.
- You can click on a link to change its state from OKAY to BROKEN. Broken links are displayed with a big red “X” and drop any packets sent along the link.

Note that a nodes routing table only contains information about the next hop in a packets journey through the network. The packet is forwarded through the network by each node choosing the next link to be used; i.e., the packets path is determined incrementally and not by choosing a complete path from source to destination at the time the packet leaves the source. This approach works because of interesting property of shortest paths: if a shortest path from A to C passes through B, then it consists of a shortest path from A to B, followed by a shortest path from B to C. In other words once the packet has reached B, we dont need to know where it came from in order to have it continue along a shortest path, we only need to know where its going.

We will construct each node’s forwarding table using Dijkstra’s shortest path algorithm, which we discussed and analyzed in lecture.

Well use special packets to transmit the necessary neighbor information between nodes. Special packets can be identified by special values in their destination fields in the lab well be sending HELLO and LSA packets which will have the strings “HELLO” and “LSA” in their destination fields. Code in Node.process can check for the presence of the special destinations and process the packets accordingly.

Link states can change (in this lab we’ll do that by clicking on the links in the graph), and the routing tables will be need to be updated to reflect those changes. Well do the updates by periodically rebuilding the routing tables from scratch.

To summarize the pre-lab thus far: get familiar with the structure of the simulator.

In addition:

1. Read through the lab exercises below to get an idea of what the lab entails.

2. Write the pseudocode for a Breadth First Search (BFS) of a graph, \( G = (V, E) \), starting from node \( s \in V \). Step 3 in the lab exercises describes the procedure. Your pseudocode should use BFS to calculate the number of nodes, using the **visited_nodes** list to keep track of the BFS order.
3. Write the pseudocode for Dijkstra’s shortest path algorithm for a graph \( G = (V, E) \), from a node \( s \in V \). The pseudocode should produce the shortest path cost to every node in the graph. As a bonus, it should keep track of which of \( s \)’s neighbor is the first node along the shortest path to any node \( v \neq s \) in the graph.

The pseudocode above will help you write the code in lab more effectively.

*This completes the pre-lab. Turn in your pseudocode to a 6.082 staff member during the lab session on May 2, 2007.*

## 2 In the Lab (2pm – 5pm, Wednesday, May 2, 2007)

Our goal is to build each node’s routing table using link state information received from other nodes in the network. We’ll rebuild the routing tables periodically using updated link state information. We’ll tackle the job in three steps:

1. Have each node periodically figure out which of its links are currently operational and thus who its current neighbors are: we’ll do this by having each node periodically send HELLO packets on its links. The source addresses of incoming HELLO packets can be saved to make list active links and the neighbors’ addresses.

2. Have each node periodically broadcast an up-to-date list of its neighbors’ addresses. Every node saves the incoming broadcasts and thus learns about the current neighbors for all the other nodes in the network.

3. Have each node periodically rebuild its routing table by doing a *breadth first search* through the neighbors data collected in step 2, using the current node as the root of the search.

### Step 1: Maintain an up-to-date list of neighbors’ addresses

In this step our goal is to have each node keep an up-to-date list of its neighbors’ addresses. To help with this task, we will use `self.neighbors`, a dictionary that takes a link as a key and returns a 2-tuple (timestamp, address) containing the address of the neighbor at the other end of the link. The timestamp is used below to identify entries that have not been updated recently and should hence be removed from `self.neighbors`.

**Step 1(a).** First add some code to `MyNode.transmit` that periodically (every `self.HELLO_INTERVAL` time steps) sends out HELLO packets along each link. A HELLO packet is distinguished from other packets by having a destination of “HELLO”. It doesn’t have any content, since the neighbors will only be interested in the source address field of the packet that contains this node’s address. Set the color property of the packet to be ‘green’ so you can distinguish HELLO packets from other packets during simulation. You can make a packet by calling `self.network.make_packet` with the appropriate arguments.

`MyNode.process` detects arriving HELLO packets and uses their source address to update the `self.neighbors` dictionary using the arrival link as the key and setting the value to the tuple `(time, p.source, link.cost)`.
Step 1(b). Next, add some code to `MyNode1.transmit` that periodically checks each entry in `self.neighbors`, discarding entries that haven’t been updated in a while. We will define “a while” to be twice the `self.HELLO_INTERVAL`, so we need to delete all `self.neighbor[link]` entries that were sent at a time older than that long ago. These are entries that aren’t being updated by incoming HELLO packets, which means some link (or node) has gone down and were no longer receiving packets from that neighboring node.

Testing your code. Test out your modifications to `lab10.py` by running it and clicking on “Step 100”. You can now click on a node in the network diagram and another window will pop up and show the information stored in `self.neighbors`. Now click on a link—a big red ‘X’ should appear showing that the link has been broken. Click on “Step 100” again and check the neighbors listing for nodes at either end of the broken link to see that those nodes are no longer neighbors. Click on the broken link to repair it, click “Step 100”, and check that the neighbors lists have returned to their original state.

Demonstrate your working code to a staff member.

Step 2: Broadcast a list of current neighbors, process broadcasts from other nodes

The next step is for a node to tell every node in the network about who its neighbors are by broadcasting link-state announcement (LSA) packets. We will use another instance variable, `self.LSA`, to track the processing of incoming LSA packets. It’s a dictionary accessed using the LSA packets source address as a key, with values that are lists of the form (sequence_number neighbor1 cost1 neighbor2 cost2 ...). Here, cost1 is the cost from the node to neighbor1, cost2 the cost to neighbor2, and so on.

Add some code to `MyNode.transmit` that periodically (every `self.LSA_INTERVAL` time steps) sends out LSA packets along each link. These packets are distinguished by having a destination of “LSA”. The LSA packet should contain:

1. A sequence number that gets incremented each time a new set of LSA packets is sent.
2. A list of pairs corresponding to your neighbors’ addresses and the link cost to reach each neighbor.

The sequence number will be used by the other nodes to determine if they’ve already processed this LSA broadcast (for this lab, we will assume that the sequence number field doesn’t wrap around). Set the color property of LSA packets to ‘red’ so that they can be distinguished during simulation.

`MyNode.transmit` includes code to clear out older entries from `self.LSA`, which runs after each round of LSA broadcasts. In this case “old” might mean entries that have a sequence number thats less than `self.LSA_seqnum-1`. This ensures that old neighbor information is discarded; the dictionary will be updated as new LSA packets arrive from other nodes in the network. You don’t have to touch this part of the code.

Step 2(b). Add some code to `MyNode.process` that recognizes incoming LSA packets and checks the incoming sequence number against the stored sequence number for the LSA packet’s source address. If the incoming sequence number is less than or equal to the stored sequence number, the incoming LSA packet can be discarded. Otherwise update the stored sequence number and neighbor list from the incoming packet, and rebroadcast the packet. To rebroadcast the packet, you should send duplicates of the packet along each of the nodes links, using `self.network.duplicate_packet()`.

Once all the LSA broadcasts have made their way through the network, `self.LSA` for each node should have a list of neighbors’ addresses for every reachable node in the network. Test out your new code by running `lab10.py`, stepping 100 time steps and clicking on a couple of nodes to see the listing of their neighbors.
LSA dictionaries. Break a few links, step another 100 time steps, and check the LSA entries have been updated correctly.

Then, try breaking a set of links that partitions the network into two unconnected pieces: do the updated LSA tables reflect the new network topology?

*Demonstrate your working code to a staff member.*

**Step 3: Build routing table from LSA dictionary using Dijkstra’s shortest paths algorithm**

In this step, well periodically rebuild the routing table using the neighbor information collected from incoming LSA packets after a new round of LSA transmissions has updated the information. Look for the stub code in `MyNode.transmit` to see where your code should go.

This code, which each node in the simulator runs, uses another dictionary, `self.spcost`, which maintains the current shortest path cost to each node in the network that this node knows about. The code initializes a few variables, and then has two main parts.

1. **Count the number of nodes in the network.** We need to know how many nodes there are in the network. That’s because from round to round, the failure of links or nodes, the recovery of links or nodes, or the arrival of new links and nodes could change this number. The nice property of LSA advertisements (and good distributed routing protocols in general) is that they are able to handle these dynamic situations because of the periodic broadcasts and refreshes of local node state.

   We will count the number of nodes by traversing the information in the LSAs, emulating a breadth first search (BFS) of the network graph, as follows.

   We will use a local variable, `visited_nodes`, which contains a list of addresses for nodes we have visited and accounted for. In each iteration, we will scan all the neighbors of another node in `visited_nodes` by looking at its LSA information, and see if any new unvisited nodes ought to be added to `visited_nodes`. The termination condition is when every node has been added to `visited_nodes` and has been considered in the loop. During this calculation, we will set the `spcost` for each visited node to be INFINITY (except for the node itself, whose `spcost` is obviously 0).

2. **Implement Dijkstra’s algorithm.** We now turn to implementing the shortest-path computation. We’ll use the local variable, `nodes_done`, to maintain a list of the node addresses for which we already know the correct shortest path. The algorithm produces `self.routes`, a dictionary of routes at the node. This dictionary is keyed by destination node address, with the value being the `link` that must be used at the node to reach the corresponding destination.

   The LSA information for node $u$ is given by `self.LSA[u][1:]`, which is the list with the first element (the sequence number) removed. The elements of this list alternate between neighbor and cost, so you need to traverse this list in steps of 2. You can do that easily in Python using

   ```python
   for i in range(0, len(mylist), 2):
   ```

   The “2” above is the step size.

   As explained in class, each iteration of the list should find one node to add to `nodes_done`. That node should be the node with the smallest total shortest path cost among all nodes in the graph not
already in `nodes_done`. You’ll need to do some bookkeeping here to get this code to work, though the code itself can be as little as 15-20 lines of Python.

Test out your new code by running `lab10.py`, clicking on “Step 100” and then clicking nodes to examine their routing tables. Break a few links, click on “Step 100” again, and see if the routing tables have been recomputed correctly.

*Demonstrate your working code to a staff member.*

Congratulations—you’ve now completed Lab 10 and are now a route computation guru (well, at least you’re well on your way to getting there)!

*Don’t forget to comment your code and hand it in at the beginning of Friday’s lecture.*