Using traceroute to understand Internet paths (20 points)

Traceroute is a tool to determine the Internet router-level path between two end hosts on the Internet. To determine the path to a given destination address, say 8.8.8.8, run traceroute as follows:

```
traceroute 8.8.8.8
```

1. How does the traceroute program work? We did not cover this in lecture, but running “man traceroute” or the Wikipedia page on traceroute will help you with this. Please answer this in your own words after referring to either source. In other words, don’t quote directly from Wikipedia or the man page. The goal is to get you to actually understand how a new network program works because you’ll likely do a fair bit of that if you use networking in a job.

2. What does an asterisk in the output of traceroute mean?

3. Run traceroute to three destinations: www.cs.nyu.edu (a server in NYC), gaia.cs.umass.edu (a server in Amherst, MA), and 139.130.4.5 (a server in Australia). Look at the traceroute output in each case, and report the IP addresses of the routers on the path between your computer and each of these destinations.

4. For each of the three destinations, use a tool like https://geoiptool.com/ to find the approximate geographic location of the routers between your computer and the destination. In many cases, the hostname of the router is indicative of its geographic location as well. You can restrict yourself to about 5 routers per traceroute, but pick these 5 routers so that you can get a sense of the geographic path taken by your packets.

Interdomain vs. intradomain routing (10 points)

1. What is interdomain routing?

2. What is intradomain routing?

3. What is the goal of an interdomain routing algorithm like BGP?

4. How is it different from the goal of an intradomain routing algorithm such as link-state or distance-vector routing?

5. Why are these goals different?

6. What is the most important factor when deciding between different alternative paths to the same destination in BGP? Why?

7. How is this different from the most important factor in picking between alternative paths in an intradomain routing algorithm?
3 Transit and peer relationships (10 points)

Consider two autonomous systems/domains/ISPs $A$ and $B$.

1. What does it mean for $A$ to be in a transit relationship with $B$?
2. What does it mean for $A$ to be in a peering relationship with $B$?
3. In the boomerang routing example that we discussed in class, what could be one reason the Kenyan ISP forces its customers to take a circuitous path through Europe even though the server is in South Africa?

4 Coding question: Distance vector routing (30 points)

Look at the attached starter code for instructions.

5 Coding question: Link state routing (30 points)

Look at the attached starter code for instructions.

General advice for coding questions

1. You’ll need to install the Python packages numpy and scipy if you don’t already have them on your computer. You can do this on most OSes using `pip install scipy` and `pip install numpy`.

2. To keep the assignment manageable, we’ll simplify several aspects of a routing protocol. We’ll assume no link-state (LS) or distance-vector (DV) advertisements are dropped in this assignment. We also won’t deal with links or routers failing, or routers dynamically being added to the topology. All our simulation experiments are one-shot: the routers all boot up together, run LS or DV, and then the experiment is over.

3. The coding questions have three TODOs, one in `dv_router.py`, and two in `ls_router.py`. But, we’ll still split points equally (30 and 30) between the DV and LS questions. This is because much of the LS question is about applying Dijkstra’s algorithm, which you already know from CSCI-UA.0310-001. Nevertheless, if you need a refresher, the Wikipedia page on Dijkstra’s algorithm is a pretty good reference, and you can always ask the course staff for help.

4. Unlike coding questions in the last two assignments, we have far fewer TODOs here, but each TODO requires 10–20 lines of code as opposed to the single-line TODOs from the previous assignments. This is intentional: we want to give you more freedom with the implementation and expect you to require less hand holding.

5. The overall code that you will be writing on your own will be quite small, but debugging it might present some challenges because these are distributed algorithms. Please review the starter code and lecture notes before starting to write your own code. We have also provided some more specific debugging advice below.

6. We’ll provide partial credit if your solution works on some graphs but not other graphs. A portion of your grade on the coding questions will be proportional to the fraction of our test cases that your code correctly handles. The remaining portion will depend on how well we think your code reflects an understanding of the LS and DV algorithms. Please leave explanatory, but concise, comments in your code to help us evaluate your understanding of the code—especially if your code doesn’t run correctly.
Running code

1. The coding questions will be carried out in the context of a simulator as with assignment 2. Running `python3 simulator.py --help` should tell you the parameters you can pass to the simulator.

2. When you initially run the code, it should throw an UnimplementedCode exception that says that there isn’t a next hop to a particular destination at a particular router’s forwarding table. To implement the LS/DV algorithms, you must fill out each router’s forwarding table (self.fwd_table in router.py) with next hop information for every destination.

3. The simulator works by generating a random graph, where you can specify the number of nodes in the graph and the probability of a link between any two nodes. The simulator then runs your routing algorithm (DV or LS) on this random graph. It also runs an offline shortest path routing algorithm on the graph, provided by the scipy library. The simulator then compares the output of the shortest paths produced by both the offline and routing algorithms and tells you if they disagree. For your solution, however, you are not permitted to use a turnkey implementation of a shortest path algorithm from scipy or any other graph library.

4. You can generate as many test cases as you want by passing different random seeds and graph sizes to the simulator. Use this to gain confidence that your code works.

5. The offline algorithm is the ground truth. It is also how we’ll be testing your code. So if there is any disagreement between your routing algorithm and the offline algorithm, it is very likely there is a bug in your code. On the other hand, if the offline algorithm and your routing algorithm match up on the output of their shortest paths (i.e., for every pair of graph nodes, they produce shortest paths that have the same total path cost) on several randomly generated test cases, then your code is very likely correct.

6. We are looking for clear and correct code, but we don’t care about the efficiency of your DV or LS algorithms—as long as it doesn’t take hours to run a simulation. This means it is OK, for instance, to implement Dijkstra’s algorithm without a heap-based priority queue and instead just scan a list to find the element with minimum priority. It is also OK to use a different algorithm for shortest paths (e.g., Bellman-Ford or Floyd-Warshall) if you find that easier to implement.

Debugging code

1. For reference, on my Macbook Air, my somewhat inefficient implementations of DV and LS take about 24 seconds on a 200-node graph and about 5 seconds on a 100-node graph. If your code is taking much longer, there is something wrong. Most likely, you are stuck in an infinite loop somewhere.

2. Start with small graphs so that you can isolate why your code is wrong. Use print statements liberally. Even after decades of computer systems research, we don’t seem to have a better tool for debugging most programs than printing our way out of problems :)

3. You can also print graphs as an adjacency list because we have implemented the `__str__` method in graph.py.

4. Use version control to your advantage. This will allow you to checkpoint versions of the code that partially work on some randomly generated graphs, before you attempt to fix problems with the code. Without version control, you run the following risk: (1) you make some changes to fix a problem, (2) but the problem actually gets worse, and (3) you don’t know how to undo these changes because you don’t remember what changes you made.