Assignment 3

Anirudh Sivaraman

Minor modifications have been made by Kyoungjun Park and Venkat Arun to adapt it for UT CS356

This document is quite long, but please review this document, the starter code, and notes from lecture 5 before starting on this assignment. Doing so will save you time later on once you get to writing code for this assignment.

1 Coding routing protocols

The goal of this coding question is to implement the link-state (LS) and distance-vector (DV) routing protocols within a tick-by-tick simulator in which time progresses in discrete "ticks". This will also teach you how to use simulators to understand network protocols.

We'll start with the prerequisites for this assignment (§1.1) and discuss some simplifying assumptions that make this assignment tractable (§1.2). We'll then do a code walkthrough (§1.3) so that you understand each of the files in the starter code. We'll then discuss how you should go about doing this assignment in a gradual manner (§1.4), while checking your work at every step. We'll then discuss a few common error messages you might see when running this assignment and how to interpret them (§1.5). We'll discuss debugging strategies (§1.6).

The overall code that you will be writing on your own will be quite small, but debugging it might present some challenges because both LS and DV are distributed algorithms. Unlike coding questions in the last assignments, we have fewer TODOs here (only 3 in total), but each TODO requires 10–20 lines of code. This is intentional: we want to give you more freedom with the implementation as long as it is correct. You will be able to determine if your implementation is correct or not using the test cases we have provided.

1.1 Prerequisites

The VM image we have provided already has all of the prerequisites installed. So we recommend you use that. If not, we have provided instructions here on what you should be installing. Please ensure you have Python 3 installed on your machine (version 3.5 or higher). You'll need to install the Python packages numpy and scipy if you don't already have them on your computer. You can install both on Ubuntu using `sudo apt-get install python3-scipy python3-numpy`. You can install both on Mac using MacPorts by typing `sudo port install py35-scipy py35-numpy`. If neither approach works for you, or if you're using Windows, consult the installation page at `http://www.scipy.org/install.html`. If that also doesn't work, let us know, but please do so early!

1.2 Simplifying assumptions for this assignment

We are making several simplifying assumptions here to keep the assignment manageable.

1. This simulator features no end hosts for simplicity. All routing is done between routers and with the goal of reaching other routers. So your goal is to find a route from each router to every other router.

2. We'll also assume no link-state (LS) or distance-vector (DV) advertisements are dropped in this assignment. So you won't have to worry about advertisements getting dropped and reliable transmission of advertisements.

3. We also won't deal with links or routers failing, or routers dynamically being added to the topology. In other words, all our simulation experiments are one-shot: the routers all boot up together, run LS or DV, and then the experiment is over.

4. We'll also assume that the graph over which you are running the routing algorithm is connected, i.e., every node is reachable from every other node.

5. 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.

1.3 Code walkthrough

There are 5 Python files in the starter code for this assignment: `graph.py`, `simulator.py`, `router.py`, `ls_router.py`, `dv_router.py`. Three of them provide supporting infrastructure for the assignment: `graph.py`, `simulator.py`, and `router.py`. The remaining two correspond to the code for a router running the link-state (`ls_router.py`) and distance-vector (`dv_router.py`) routing protocols. You will be adding your code to `ls_router.py` and `dv_router.py` at three locations marked with TODOs. You do not need to modify the code in the other three files, but you need to understand all 5 files for the assignment.

In addition, there are a few `.graph` files that you can use to manually create graphs and feed them as test inputs to the simulator to test your routing protocols. These files have a simple format and can be used to create small test graphs for your protocols, where the test graphs are simple enough that you understand what the right output of any routing protocol should be on these small graphs. We'll now go over each of the 5 files.

1.3.1 `graph.py`

`graph.py` provides functions to represent and manipulate graphs. We represent graphs as objects of `class Graph`, which supports the following member functions.

1. `__init__`: This is the graph constructor which creates an empty graph object with no nodes or edges. It initializes `self.adj_list`, a dictionary representing the graph as an adjacency list. The key in this dictionary is a node ID and the value is a list of tuples denoting edges from this node. Each tuple has two elements: the other end of the edge from this node and the weight of the edge.

2. `__str__`: This provides the ability to print out graphs in the `.graph` format (see below for details of the `.graph` format), which can be helpful during debugging.

3. `add_node`: Adds a node to the graph.

4. `add_edge`: Adds an edge to the graph. This function assumes that both ends of the edge have already been added to the graph using calls to `add_node`.

5. `adj_mat`: Returns an adjacency matrix representation of the graph. This is required by scipy to compute (centralized) shortest paths for an input graph; we use scipy as the reference to check that the shortest paths that you compute using your implementation of the routing algorithm are actually correct.

In addition, `graph.py` contains two non-member functions `graph_from_file` and `gen_rand_graph`, which are used to create a graph from a `.graph` file and generate a random graph respectively.

`graph_from_file` is relatively straightforward; it reads a `.graph` file (e.g., `line.graph`, `ring.graph`, and `clique.graph` in the starter code) and creates a graph object based on it. The first line in a `.graph` file specifies the number of nodes (num_nodes) in the graph. The node ids, which identify the nodes in the graph, run from 0 to num_nodes - 1. The second and subsequent lines in a `.graph` file specify the edges in the graph. Each edge line has three numbers separated by a space: the from node, the to node, and the edge weight (or

link cost). All edges are assumed to be bidirectional, so you should not specify an edge from both 0 to 1 and 1 to 0.

`gen_rand_graph` generates a random graph with a certain number of nodes (`num_nodes`) and a certain probability of an edge between any two nodes (`edge_probability`). The edge weights are chosen from a uniform distribution between 1 and a certain maximum edge weight (`max_edge_weight`). These three parameters are passed to `gen_rand_graph` using three variables in `simulator.py`: `args.num_routers`, `args.link_prob`, and `MAX_LINK_COST` respectively.

1.3.2 `simulator.py`

`simulator.py` runs your routing algorithm (DV or LS) on either a randomly generated graph or a graph that you specify using a `.graph` file. 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 your 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.**

`simulator.py` does three main things.

1. It reads inputs that the user supplies on the command line and runs either the DV or LS algorithm on either a randomly generated graph or a graph read in from a `.graph` file. To run the simulator on a randomly generated graph, you should type:

   ```
   python3 simulator.py DV rand_input --link_prob 0.5 --seed 1 --num_routers 10
   ```

   This will run the DV protocol on a randomly generated graph with 10 nodes, with a probability of 0.5 of an edge between any two nodes, and a random seed of 1. The seed is a mechanism to make the output from a random number generator deterministic and repeatable. Different seeds will produce different random-looking choices. A true pseudo-random number generator starts with a truly random seed, for example generated from physical sources of randomness such as temperature or user input. However, picking a constant, non-random, seed will allow you to debug your protocol under identical conditions (i.e., it will generate the same random graph) between two runs of the simulator. You can use any seed; just remember that using the same value of the seed should give you the same output because the seed controls all the randomness in the simulator.

   Similarly, to run either DV or LS on a graph read in from a `.graph` file, you should type:

   ```
   python3 simulator.py LS file_input --graph_file line.graph
   ```

   This will run the LS protocol on the graph represented by the `line.graph` file, which is provided as part of the starter code.

2. `simulator.py` runs the DV or LS protocol tick by tick on the graph that was either randomly generated or supplied through a `.graph` file. This is accomplished by the for loop:

   ```
   # Now simulate
   for tick in range(0, NUM_TICKS):
     self.clock.set_tick(tick)
     for i in range(0, num_nodes):
       routers[i].run_one_tick()
   ```

   This for loop assumes that time starts at tick 0, and on every tick, a router can send a DV/LS advertisement to its neighbors. This advertisement is received by the neighbors at the end of that tick, which can then run any protocol-specific computation and advertise the DV/LS advertisement to *their* neighbors. These neighbors receive the DV/LS advertisement at the end of the next tick and so on.

3. `simulator.py` computes the correct shortest paths on the test graph (either randomly generated or read in from the `.graph` file) using an offline shortest paths algorithm supplied by the scipy library. This algorithm is centralized. It takes in the whole graph as input and runs something akin to Dijkstra's, Bellman-Ford, or Floyd-Warshall on the graph. `simulator.py` then compares the shortest paths computed by scipy with the shortest paths computed by your DV or LS implementation.

If the shortest paths between any two nodes are different between the scipy algorithm and your implementation (i.e., the paths include different nodes), `simulator.py` will check that the length of the shortest paths are the same. For any pair of nodes, if the length of the shortest path computed by your DV/LS implementation is the same as the length of the shortest path computed by scipy, then your implementation is correct on this particular test graph. If the shortest path lengths of your implementation and scipy are different, you will get an error about it, and there is a bug you need to fix.

If your implementation matches scipy on several randomly generated test graphs, then it is very likely your implementation is correct.[1] On the other hand, if there is disagreement between your routing algorithm and the scipy algorithm on any test input, it is very likely there is a bug in your code.

1.3.3 `router.py`

`router.py` is the base class for a router in our simulator. Both `ls_router.py` and `dv_router.py` derive (inherit) from this class (called `Router`). It consists of four member variables:

1. `self.neighbors`: A list consisting of references to neighbors of a router. These references are to objects of type Router as well (either `LSRouter` or `DVRouer` as the case may be).

2. `self.links`: A dictionary that maps a router's neighbor's router ID to the link cost of the link from a router to this neighbor. As we said earlier, all graphs in this assignment are undirected. So link costs from and to a neighbor are identical.

3. `self.router_id`: An integer that uniquely identifies a router. You can think of this as the router's address. We won't complicate this assignment by using real, hierarchical, IP addresses. Instead we'll use these router_ids to represent a router's address using a single integer.

4. `self.fwd_table`: This is a dictionary representing a forwarding table that maps a destination address (identified by the destination's `router_id`) to the next hop router for that destination (again identified by the next hop's `router_id`). This is the final output of the routing protocol and is how you communicate the result of your routing protocol to `simulator.py`. You must populate this correctly to make sure `simulator.py` can correctly compute shortest paths based on your routing protocol implementation.

`router.py` also includes two helper member functions to add neighbors (`add_neighbors`) and links (`add_links`) to a router. These functions need to be called once all routers have been constructed because they take a reference to a router object as an argument.

1.3.4 `ls_router.py`

`ls_router.py` is the file that implements the class `LSRouter`. It derives from Router, and hence includes all the member variables that Router includes. In addition, it includes the following member variables.

1. `self.broadcast_complete`: This is a boolean variable that indicates whether the link state advertisement (LSA) broadcast has been completed, i.e., whether it is time to run Dijkstra's (or some other single source shortest path algorithm). For now, we use a very simple heuristic to determine if the broadcast has completed: if it's been 1000 ticks since the beginning of the simulation, we assume that the broadcast is done. **This means you should keep your graph sizes under 1000 nodes so that the length of any path in the graph does not exceed 1000.**

---

[1]Very likely and not certain because we are randomly sampling the space of test inputs to the simulator and not exhaustively testing against every possible input.

2. `self.routes_computed`: This is a boolean variable that indicates if routes have been computed using Dijkstra's algorithm. This just ensures that routers do not repeatedly run Dijkstra's algorithm once the broadcast has completed.

3. `self.lsa_dict`: This is a dictionary data structure that maps from an integer representing the router ID to the LSA advertised by that router. Each router LSA is itself a dictionary that maps the neighbor's router ID to the link cost to that neighbor. `self.lsa_dict` is initialized to hold a single entry (`{self.router_id : self.links}`). This reflects the fact that at tick 0, each router knows about the LSA (`self.links`) for itself alone (`self.router_id`). As the LSA broadcast proceeds, each router gets to know about the LSAs of all other routers in the network. The `self.links` member variable is inherited from the Router class that `LSRouter` derives/inherits from. You can think of `self.lsa_dict` as a collection of adjacency lists, where there is one adjacency list for each router in the network. These adjacency lists correspond to the `self.links` of each router object. Once the LSA broadcast is complete, you have the adjacency lists for all routers in the network, giving you all the information you need to run Dijkstra's algorithm.

4. `self.broadcasted`: This is a dictionary that maps a router ID to a boolean variable. Given a router ID (R), it tracks if the LSA originated by the router with ID R has been broadcasted by this router (self) to all of self's neighbors. This is required to ensure that a router that receives an LSA for a particular router multiple times does not broadcast it out on its links more than once.[2]

`LSRouter` contains the following member functions.

1. `__init__`: This is the constructor for `LSRouter` and initializes all the member variables to their correct default values.

2. `initialize_algorithm`: This function is called by `simulator.py` after a router's neighbors and links have been added to each router. We need two separate functions `__init__` and `initialize_algorithm` because we first need to create all the router objects corresponding to all graph nodes in `simulator.py` using the `__init__` function. We then add each router's neighbors using references to router objects that we just created (`add_neighbors`), and then add links (`add_links`) to these neighbors. Only after the neighbors and links are up, can a routing algorithm be initialized (`initialize_algorithm`).

3. `run_one_tick`: This function specifies what happens on each tick of the algorithm. For the LS algorithm, if the tick is under the `BROADCAST_INTERVAL` we simply broadcast any LSA that we have received at this router so far to this router's neighbors—if we have not already done so. To do so, use the list of keys in `self.lsa_dict` to determine all the LSAs that have been received at this router and use the `self.broadcasted` dictionary to determine if a particular LSA has been broadcasted so far or not. This broadcasting logic in tick is something you need to implement on your own.

4. `send`: This is a helper function for a router to send an LSA to one of its directly connected neighbors. It works by adding the LSA to the `self.lsa_dict` dictionary of the neighbor, even if it already exists in that dictionary. A dictionary does not have any duplicate keys, so rewriting a key with the same value (because LSAs don't change in our assignment) is harmless.

5. `dijkstras_algorithm`: This is where you implement Dijkstra's algorithm to complete the LS algorithm. If you need a refresher on Dijkstra's algorithm, the Wikipedia page on Dijkstra's algorithm is a pretty good reference (`https://en.wikipedia.org/wiki/Dijkstras_algorithm#Pseudocode`). You do not need to use a priority queue or heap to implement Dijkstra's algorithm. You can simply find the minimum by traversing a list if that's easier. Your final output should be a valid configuration of `self.fwd_table` that contains the next hop for every destination in the network. You can use another single-source shortest paths algorithm like Bellman-Ford or even another all-source shortest path algorithm like Floyd-Warshall if you find that easier and can still compute the correct routes.

---

[2]This is because broadcasting multiple times is wasteful unless these broadcasts get dropped, which we aren't concerned with for this assignment.

6. `next_hop`: This is an optional helper function that you can use to recursively find the next hop towards a destination by using a dictionary (`prev`) that maps every destination router ID to the penultimate router on the path to that destination router ID. By recursively looking up entries in `prev`, you can use `prev` to find the next hop for any destination once you have computed `prev`. You should be able to compute `prev` at the end of Dijkstra's algorithm. You don't *have* to use this. It is just provided if you find it helpful.

1.3.5 `dv_router.py`

`dv_router.py` implements the class `DVRouter`. Similar to `LSRouter`, `DVRouter` also inherits/derives from Router. In addition to the member variables of Router, `DVRouter` contains the following member variables.

1. `self.dv_change`: This is a variable that tracks if the distance vector maintained by a router has been modified or not. If it has been modified, it needs to be sent out again to all neighbors of this router.

2. `self.dv`: This is a variable that stores the distance vector for a router. `self.dv` is a dictionary that maps from a router ID to the current known best distance for that router ID from self. To begin with, at time 0, you can assume every router R is initialized (as part of the `initialize_algorithm` member function) with a distance vector with the following entries: (1) if another router O is directly connected to R, the distance to O is the link cost of the link between R and O, (2) for all other routers, the distance is infinity. We don't explicitly represent a distance of infinity in the `self.dv` dictionary. Instead if a particular destination router ID is not present in `self.dv`, we will take this to mean that the distance to that router ID is infinity.

DVRouter contains the following member functions.

1. `__init__` constructs a `DVRouter` object and sets all its member variables to their initial values.

2. `initialize_algorithm` initializes the DV algorithm once the router's neighbors and links have been added to the router. It initializes two data structures, `self.dv` and `self.fwd_table`. `self.dv` is initialized such that a router knows the best distance to its neighbors alone; all other destinations have a distance of infinity. Similarly, `self.fwd_table` is initialized so that the next hops are known for neighbors alone. `self.dv` and `self.fwd_table` evolve hand in hand; every time `self.dv` changes, you need to update `self.fwd_table` as well.

3. `run_one_tick` specifies what happens on each tick. This is very straightforward for DV. If the distance vector has changed since the last time it was sent out to the router's neighbors, it needs to be sent out again.

4. `send` allows a router to send out a distance vector to one of its directly connected neighbors. It does this by calling the neighbor's `process_advertisement` member function.

5. `process_advertisement` is the core part of the DV algorithm that you will need to implement. You need to check if the DV that you just received from your neighbor allows you to find a better path to any destination through your neighbor. This also includes destinations for which your current distance is infinity. For such destinations, if the neighbor contains any non-infinite distance in its distance vector, the neighbor trivially has a better path to the destination. To complete the implementation of DV correctly, ensure that you populate `self.fwd_table` correctly so that it has a next hop for every possible destination.

1.4   Getting started with this assignment

To start with, ensure you have the prerequisites (scipy and numpy) for this assignment. You can check this by opening up a python3 interpreter, typing `import scipy` and `import numpy`, and verifying that there are no errors.

Once this is done, do a quick sanity check that you can run the starter code by feeding the `clique.graph` file as the test graph to the simulator. This is a 4-node clique where every router is connected to every other

router with a link cost of 1. On a clique with identical link costs for all links, a DV router has the shortest path to every other router at time t=0 based on the initial contents of its forwarding table created as part of the `initialize_algorithm` member function. In other words, the starter code should already compute the shortest path for DV on `clique.graph`. Verify this by running the following command and checking that the output is similar to what is shown below.

```
Anirudhs-Air:asg3_starter_code anirudh$ python3 simulator.py DV file_input --graph_file clique.graph
Namespace(graph_file='clique.graph', input_type='file_input', rt_algo='DV')
num_nodes is  4
Graph created from file has 4 nodes, 6 edges

SUCCESS: Routing and offline algorithm agree on shortest paths between all node pairs
```

Now run LS on the same `clique.graph`. For LS, the `initialize_algorithm` function doesn't populate the forwarding table (`self.fwd_table`) and hence you should get the following error message.

```
Anirudhs-Air:asg3_starter_code anirudh$ python3 simulator.py LS file_input --graph_file clique.graph
Namespace(graph_file='clique.graph', input_type='file_input', rt_algo='LS')
num_nodes is  4
Graph created from file has 4 nodes, 6 edges
Traceback (most recent call last):
  File "simulator.py", line 133, in <module>
    raise UnimplementedCode(str(j) + " isn't in router " + str(i) + "'s fwd table")
__main__.UnimplementedCode: 1 isn't in router 0's fwd table
```

This tells you that the forwarding table at router 0 does not have an entry for router 1. Once you implement LS correctly, this error message should disappear and be replaced with:

```
SUCCESS: Routing and offline algorithm agree on shortest paths between all node pairs
```

If on the other hand, there is a bug in your routing algorithm, you will get an error message like this.

```
Anirudhs-Air:asg3_starter_code anirudh$ python3 simulator.py DV file_input --graph_file triangle.graph
Namespace(graph_file='triangle.graph', input_type='file_input', rt_algo='DV')
num_nodes is  3
Graph created from file has 3 nodes, 3 edges

NOTE: Routing algorithm computed shortest path from  0  to  1  as  [0, 1]
Offline algorithm computed shortest path from  0  to  1  as  [0, 2, 1]
Distance computed by offline algorithm =  3.0
Distance computed by routing algorithm =  5.0
ERROR!!!: Distances differ

NOTE: Routing algorithm computed shortest path from  1  to  0  as  [1, 0]
Offline algorithm computed shortest path from  1  to  0  as  [1, 2, 0]
Distance computed by offline algorithm =  3.0
Distance computed by routing algorithm =  5.0
ERROR!!!: Distances differ

ERROR: There was at least one path on which routing and offline algorithm did not agree
```

As the message indicates, this means that there was at least one path where your routing algorithm was incorrect and did not agree with the output of the scipy (offline) algorithm. In the example above, this is because after DV's `initialize_algorithm` function has run, a DV router has *some* distance to every destination when the input graph is `triangle.graph` (because every router is directly connected to every other router in `triangle.graph`). But this is not the *shortest* distance to the destination.

Once you have run the starter code's unimplemented DV and LS algorithms on `clique.graph`, you should start implementing one of the two algorithms. DV requires less code but is more distributed. LS requires more

code but is (mostly) centralized. I did DV first because it's less code to reason about, but it's up to you. When implementing these algorithms, it is a good idea to add print statements to print the relevant data structures (e.g., `self.lsa_dict`) even before writing the algorithmic logic because this lets you understand the flow of data through the simulator.

Whichever algorithm you pick, you should test it on simple test cases generated using `.graph` files first. We have provided three test files `ring.graph`, `line.graph`, and `clique.graph` to represent a ring, straight line, and clique graph where all links have unit costs. These are very simple topologies where you should be able to quickly check whether you have the right shortest paths.

Once you can successfully pass these test cases, you should move on to slightly harder, but still manually generated use cases using `.graph` files. `midterm.graph` is the example from the 2017 midterm. `triangle.graph` is an example that illustrates the DV algorithm based on Prof. Nick Feamster's YouTube video here: `https://www.youtube.com/watch?v=x9WIQbaVPzY`. These are both examples in which the shortest path may have a longer hop count than an alternative path.

Once you have passed these test cases, you can move on to the randomly generated test cases, where you can generate random graphs and compare your routing algorithm against the scipy algorithm. Restrict yourself to graphs with at most 100 nodes (the num_routers command line argument) for simplicity. 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. If you want to debug your code though, manually generated `.graph` files work better.

You can also use the following procedure combining random and manually generated graphs if it helps. First, generate a random graph. Most likely your DV/LS implementation fails on that graph the first time around. You can then print out the random graph (recall that the class Graph has a `__str__` function to write Graphs in the `.graph` format) into a file and use this file as a `.graph` file for a subquent run. You can then pare down the `.graph` file until you are able to understand exactly why your routing algorithm is failing. Once you have fixed it for this test case, it is likely it will work correctly on several more randomly generated graphs because the number of failure modes is quite small relative to the number of test cases.

When we test your code, we'll use a similar pattern to test your code with easy, hard, and random test cases.

1.5 Understanding error messages

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.

If you generate a random graph but your link probability is too low, the randomly generated graph may be disconnected. The same applies if you input your own `.graph` file and the `.graph` file represents a disconnected graph. We are not testing you on disconnected graphs, so you'll get the following error message.

```
Anirudhs-Air:asg3_starter_code anirudh$ python3 simulator.py LS rand_input --seed 1 --num_router 20 --link_prob 0.1
Namespace(input_type='rand_input', link_prob=0.1, num_routers=20, rt_algo='LS', seed=1)
Random graph has 20 nodes, 16 edges
Traceback (most recent call last):
  File "simulator.py", line 83, in <module>
    raise argparse.ArgumentTypeError("Input graph is disconnected.\nFor randomly generated graphs, use a different seed
 or increase the link probability.\nFor graphs generated from files, check the graph file.")
argparse.ArgumentTypeError: Input graph is disconnected.
For randomly generated graphs, use a different seed or increase the link probability.
For graphs generated from files, check the graph file.
```

1.6 Debugging strategies

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 5–10 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, both to understand the flow of code in the simulator, and to debug. 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 out into `.graph` files because we have implemented the `__str__` function in `graph.py`.

4. (**Optional**) If you are already familiar with git and have a personal repository we recommend using version control to your advantage. This will allow you to checkpoint versions of the code that partially work on some test 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.

1.7 Other important details

1. Do not invent your own notion of time for the assignment. Time in the simulator is captured by the clock. In particular don't use functions like time.time() to capture the current wall-clock time. Instead, fetch the time using `clock.read_tick()`

2. Pass statements are a Python placeholder for no-op statements (similar to the semicolon in Java and C++), which are required so that the code passes syntax checks. They have no effect on the program.

3. For your solution to the DV/LS algorithms, you are not permitted to use a turnkey implementation of a shortest path algorithm from scipy or any other graph library.

4. We use the terms routing protocols and algorithms interchangeably throughout this document.