Project 3 - Distance Vector Routing

In the lectures, you learned about Distance Vector routing protocols, one of the two classes of routing protocols. DV protocols, such as RIP, use a fully distributed algorithm that finds shortest paths by solving the Bellman-Ford equation at each node. In this project, you will develop a distributed Bellman-Ford algorithm and use it to calculate routing paths in a network. This project is similar to Project 2, except that we are solving a **routing** problem, not a **switching** problem. As we did in Project 2, we will be simulating a small network running the Bellman-Ford algorithm.

In "pure" distance vector routing protocols, the hop count (the number of links to be traversed) determines the distance between nodes. However, some distance vector routing protocols that operate at higher levels like BGP must make routing decisions based on business relationships in addition to hop count. These protocols are sometimes referred to as Path Vector protocols. We will explore this by using weighted links (including negatively weighted links) in our network topologies.

We can think of Nodes in this simulation as individual Autonomous Systems (ASes), and the weights on the links as a reflection of the business relationships between ASes. Links are directed, originating at one Node and terminating at another.

Getting Started

You should review some materials on Bellman-Ford. Some resources include:

- Wikipedia
- YouTube

Download and unzip the Project Files for Project 3 from Canvas in the Assignments section.

Project Layout

In the `Project-3` directory, you can see quite a few files. They're described below:

- `DistanceVector.py` - This is where **all** your work will be. It is a specialization (subclass) of the Node class that represents a network node (i.e., router) running the Distance Vector algorithm. You will have to implement the algorithm in this file. This is the *only* file that you need to make any changes to.
- `Node.py` - Represents a network node, i.e., a router.
- `Topology.py` - Represents a network topology. It's a container class for a collection of DistanceVector nodes and the network links between them.
- `run_topo.py` - A simple "driver" that loads a topology file (see `*Topo.txt` below), uses that data to create a Topology object containing the network Nodes, and starts the simulation.
- `helpers.py` - This contains logging functions that implement that majority of the logging code for you.

- `*Topo.txt` - These are valid topology files that you will pass as input to the `run.sh` script (see below).
- `BadTopo.txt` - This is an invalid topology file, provided as an example of what not to do, and so you can see what the program says if you pass it a bad topology.
- `output_validator.py` - This script can be run on the log output from the simulation to verify that the output file is formatted correctly. It does not verify that the contents are correct, just that the *format* is correct. If your output contains formatting errors, they will be announced by the validator. If it completes without announcing an error, then your output matches what is expected!
- `run.sh` - Helper script that launches the simulation on a specified topology and automatically runs the output validator on the log output when the simulation finishes. This script is basically a convenient wrapper for `run_topo.py` and `output_validator.py`.

To Dos:

There are a few TODOs in DistanceVector.py that you will have to complete:

1. Review the methods already implemented in Node.py. Because DistanceVector subclasses Node, consider how you might use these existing methods to complete the rest of the ToDos in this list. Do NOT modify Node.py
2. Decide on how each node will represent its distance vector. Consider what might be the simplest data structure to keep track of path weights (i.e., the distance vector). The distance vector variable should be local to the node, i.e., defined in the init function as a variable accessible via the self object (i.e. self.mylist).
3. Implement the Bellman-Ford algorithm. Similarly to Project 2, each Node will send out an initial message to its neighbors, and develop its view of the rest of the network (weighted distance to other nodes in the topology) by processing messages received from other Nodes (and sending out updates as needed). At the initial state you should assume that the node only knows itself and that it is reachable at cost 0 when it sends out initial messages. To determine the cost to reach a Neighbor and the neighbor's reachable destinations, each node will access the weights of the links connecting a Node to its Neighbor.
4. Write a logging function that is specific to your particular distance vector structure. We have provided much of the framework for you to use, along with logging helper files to take care of the bulk of the logging. As with your Bellman-Ford implementation, you should assume that the logging function only knows itself. Do not access the topology for logging; logging should happen at the Node level.
5. Observe Spirit of the Project guidelines in your Bellman-Ford implementation and your logging function.

To run your algorithm on a specific topology, execute the `run.sh` bash script like so:

`./run.sh *Topo`

This will execute your implementation of the algorithm in `DistanceVector.py` on the topology defined in `*Topo.txt` and log the results (per your logging function) to `*Topo.log`. NOTE: You should *not* include the full filename of the topology when executing the `run.sh` script. For example, to run the algorithm on `topo1.txt` you should only specify `topo1` as the argument to `run.sh`.

For this project, you may create as many topologies as you wish and share them on Piazza. We encourage that you do share new topologies, and your log outputs to confirm correctness of your algorithm. If there's a bug with a created topology, you will get an error back when you try to run it. We've included four good topologies for you to use in testing and one bad topology so that you can see what will happen with an invalid topology. A correct logging result for each of the provided topologies does not necessarily mean that you have a correct solution, so do create additional tests and run your solution against them.

What topologies will and won't look like

We will be using many topologies to test your project. This includes but is not limited to:

1. topologies with and without cycles (loops), including odd length cycles
2. topologies of varying sizes, including topologies with more than 26 nodes (meaning node names may be longer than 1 character)
3. topologies with multiple paths to different nodes
4. topologies that include any combination of positive weights, zero weight, and negative weight
5. topologies with negative cycles, meaning a node may reach another at infinitely low cost
6. topologies with Nodes that do not have incoming or outgoing links. That is, that while all nodes will be connected, some may have both incoming and outgoing links, some may only have incoming links, and some may only have outgoing links

We will NOT test your submission against the following topologies (which means your algorithm does not need to account for them):

1. topologies with more than one link from the same origin to the same destination (multi-graphs)
2. topologies with portions of the network disconnected from each other (partitioned networks)

If you have any other questions about what topologies will and won't include, please ask on Piazza.

Correct Logs for Provided topologies

Below are the correct final logs for the provided topologies. We are providing them in order to help you identify correct behavior with respect to negative cycles and the assumptions in the instructions. We are only providing the final round; each topology should produce at least 2 rounds of output.

SimpleTopo:
A:A0,C3,B1,D3
B:A1,C2,B0,D2
C:A3,C0,B2,D0
D:A3,C0,B2,D0
E:A2,C-1,B1,E0,D-1

SingleLoopTopo:
A:A0,C16,B6,E6,D5
B:A2,C10,B0,E0,D7
C:C0
D:A3,C11,B1,E1,D0
E:A2,C10,B0,E0,D7

SimpleNegativeCycle:
AA:AA0,CC-99,AB0,AE-1,AD-2
AB:AA-1,CC-99,AB0,AE-2,AD-3
AD:AA1,CC-99,AB2,AE1,AD0
AE:AA0,CC-99,AB1,AE0,AD-2
CC:CC0,AA-1,AB0,AE-2,AD-3

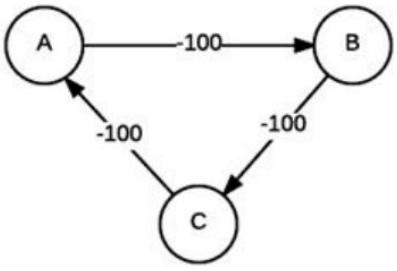
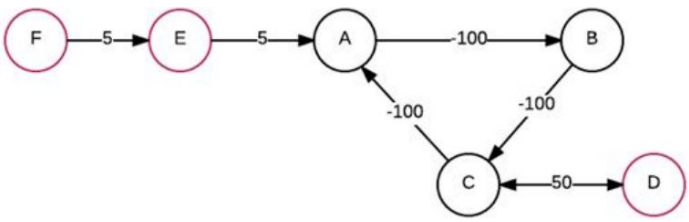
ComplexTopo:
ATT:TWC-99,GSAT-8,UGA-99,ATT0,VZ-3,CMCT-99,VONA-11
CMCT:TWC-99,GSAT-7,UGA-99,ATT1,VZ-2,CMCT0,VONA-10
DRPA:TWC-99,GT-1,GSAT5,UGA-99,PTGN1,OSU-1,ATT13,VONA2,EGLN1,VZ10,DRPA0,CMCT-99,UC-1
EGLN:TWC-99,GT-2,GSAT5,UGA-99,PTGN0,OSU-2,ATT13,VONA3,EGLN0,VZ11,DRPA1,CMCT-99,UC-2
GSAT:TWC-99,GSAT0,UGA-99,ATT7,VZ5,CMCT-99,VONA-3
GT:TWC-99,GT0,GSAT7,UGA-99,PTGN2,OSU0,ATT15,VONA5,EGLN2,VZ13,DRPA3,CMCT-99,UC0
OSU:TWC-99,GT0,GSAT7,UGA-99,PTGN2,OSU0,ATT15,VONA5,EGLN2,VZ13,DRPA3,CMCT-99,UC0
PTGN:TWC-99,GT-1,GSAT5,UGA-99,PTGN0,OSU-1,ATT13,VONA3,EGLN1,VZ11,DRPA2,CMCT-99,UC-1
TWC:TWC0,GSAT-7,UGA-99,ATT1,VZ-2,CMCT-99,VONA-10
UC:TWC-99,GT0,GSAT7,UGA-99,PTGN2,OSU0,ATT15,VONA5,EGLN2,VZ13,DRPA3,CMCT-99,UC0
UGA:TWC-99,GSAT42,UGA0,ATT50,VZ47,CMCT-99,VONA39
VONA:TWC-99,GSAT2,UGA-99,ATT10,VZ8,CMCT-99,VONA0
VZ:TWC-99,GSAT-6,UGA-99,ATT2,VZ0,CMCT-99,VONA-9

Key assumptions and clarifications

In order to avoid confusion, there are some assumptions we will make about behaviors of Nodes in this project:

1. The direction of a link indicates how traffic will flow, and two nodes connected with a link may pass messages regardless of that direction. Specifically, if a node A has an incoming link from node B, but has no outgoing link to that node B, node A will still send its distance vector to node B to "advertise" other nodes it can reach. Thus, a Node's distance vector is comprised of the nodes it can reach via its outgoing links (and a vector to itself of size 0), and **it advertises this vector to nodes that can reach it via an incoming link**.
2. Negative cycles are defined as a series of directed links that originate and terminate at a single node, where the sum of the link weights is less than 0. When negative cycles are present in a topology, it is possible that we will encounter a version of the count-to-infinity problem: where the distance vector from one node to another will continue to decrease by the weight of the negative cycle every few rounds. Your code must be able to detect and label negative cycles in order to terminate. Any node that can reach a destination node and infinitely traverse a negative cycle en route will set the distance to that node to -99.
3. Assume that edge weights between any two nodes in the topology will be between -50 and 50.
4. Assume for this project that nodes will not advertise the distance to itself as any value less than zero. This reflects that Nodes will not intentionally route its own traffic externally. This is especially relevant when considering negative cycles - a Node will not intentionally route traffic destined for itself through a cycle.
5. Continuing point 3 above, a Node will send traffic bound for other destinations through a negative cycle. This means that if one node can reach a destination via an infinite cycle (its vector is -99) then it will advertise that vector to its upstream neighbors. Further, a Node seeing an advertised vector of -99 from a downstream neighbor can assume this means it can reach that same destination at infinitely low cost (-99).
6. Keep in mind that this project is a simulation and for this project you are not required to optimize the behaviour of the node/routers based on the existence of negative cycles. Your nodes are required to detect and mark them.

What is a Negative Cycle?

The topology below is NOT a negative cycle. As each node is part of the loop, you cannot get into a loop going to a particular node because the node is in the loop and the routing would stop once reached. 	The path from Node E to Node D contains a negative cycle. 

Although not specific to this course or project, Professor Vigoda here explains Negative Weight Cycles

Spirit of the Project

As with Project 2, the goal of this project is to implement a simplified version of a network protocol using a distributed algorithm. This means that your algorithm should be implemented at the network node level. Each network node only knows it's internal state, and the information passed to it by its direct neighbors. **Declaring global variables will be a violation of the spirit of the project**.

The skeleton code we provide you runs a simulation of the larger network topology. For simplicity, the `Node` class defines a link to the overall topology. This means it is possible using the provided code for one Node to access another Node's internal state. This goes against the spirit of the project, and is not permitted.

When we grade your code, we will use a special version of `Node.py` that will have a randomly generated variable name for the topology object, and if you access it directly in order to generate your distance vectors in `DistanceVector.py`, your code will throw a runtime error, and receive no credit. **If you have questions about whether your code is accessing data it should not, please ask on Piazza or during office hours!**

What you can (and cannot) share

Do **not** share the content of your `DistanceVector.py` file with your fellow students, on Piazza, or elsewhere publicly. You **may** share any log files for any topology, and you may also share new topologies. Additionally, code that you write that is not required for turn-in, like testing suites may be shared. It may be a good idea to share a "correct" logs for a particular topology, if you have one, when you share the code for that topology. When sharing log files, you should leave alphabetization on. This allows your classmates to use the `diff` tool to see if you are getting the same log outputs as they are.

What to turn in

You only need to turn in the file you modify, `DistanceVector.py`, but you need to zip the file with your GTID you use to log into Canvas and the project number. This will take the form `GTLogin_p3.zip` where GTLogin should be replaced with your ID you use to log into Canvas (e.g., `smith7_p3.zip`). Do not modify the name of `DistanceVector.py` and do not place `DistanceVector.py` into a folder before zipping or else grading will be affected. DistanceVector.py must be at the top level when extracted from the GTLogin_p1.zip file.

There are some very important guidelines for this file you must follow:

1. **Your submission must terminate!** If your submission runs indefinitely (i.e. contains an infinite loop) or throws an error at runtime, it will not receive full credit. Termination here is defined as self-termination by the process. Manually killing your submission via console commands or interrupts is NOT an acceptable means of termination.
2. **Remove any print statements from your code before turning it in!** Print statements left in the simulation, particularly for inefficient but logically sound implementations, have drastic effects on run-time. Your submission should take less than 10 seconds to process a topology. If your leave print statements in your code and they adversely affect the grading process, your work will not receive full credit. Feel free to use print statements during the project and during debugging, but remove them before you submit to Canvas.
3. **Logging is important for this project!** It is the only way that we can verify that your algorithm is running correctly. The output validator will catch most formatting mistakes, but you should inspect your output by hand to make sure it matches the requested format just in case the script missed something. The proper logging format is specified in the TODO comment for logging located in `DistanceVector.py`.
4. **Grading Note**: Zero credit will be given if logging is incorrect. Incorrectly formatted logs will not match properly and fail the autograder, and we will not be manually inspecting incorrectly named/formatted/etc. logs due to the number of students in the class. Additionally, partial credit will not be awarded to distance vector logs that are *mostly* correct. If the output contains some paths that are between nodes that are not the shortest available route, then there is an issue with the algorithm with respect to that test case.

Grading

10 pts	Correct Submission	For turning in the correct file, with the correct name, and significant effort has been made towards completing the project.
50 pts	Provided Topologies	For correct Distance Vector results (log file) on the provided topologies. 0 or 8 pts. could be awarded for each topology.
90 pts	Unannounced Topologies	For correct Distance Vector results (log file) on topologies that you will not see in advance. They are slightly more complex than the provided ones, and test the cases described above. 0 or 12.5 pts. could be awarded for each topology.

FAQs

Q: May I import a python module into DistanceVector.py? For example, May I use `import collections`
A: Your DistanceVector.py submission must run on the provided course VM without requiring the installation of any additional packages or software. All submissions will be tested using the commands and files described in this document, as well as with several additional unannounced topology files. This means that any modules included in the standard Python installation of the VM are fine for import. However, most students complete this project without doing so.

Q: What is the best way to format and process node messages?
A: There is no right or wrong way to format messages. For best results keep things simple.

Q: Is it required that the distance vectors displayed in my log files be alphabetized?
A: Take a look at the code in helpers.py. Note how the DVs are alphabetized each round, and this is reflected in the provided correct output logs. The nodes within individual vectors are not required to be sorted.

Q: Should my solution include an implementation of split horizon?
A: That is *not* a requirement for this project.

Q: What if there really is a valid path between two indirectly linked nodes with no cycle and the total cost is -99 or less?
A. We will not test your submission against a topology that does this. However, point 4 from the Key assumptions and clarifications holds: "a Node seeing an advertised vector of -99 from a downstream neighbor can assume this means it can reach that same destination at infinitely low cost (-99)."