Project 4 - Distance Vector Routing

In the lectures, you learned about Distance Vector (DV) routing protocols[1], 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 like Project 2, except that we are solving a **routing** problem, not a **switching** problem.

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 a hop count. These protocols are sometimes referred to as Path Vector protocols[2]. 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.

1. Getting Started

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

- Wikipedia (https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm)
- "Computer Networking: A Top-Down Approach" by Kurose and Ross
    - 7th edition discusses the algorithm on pages 384-385 in Chapter 5 ("The Network Layer: Control Plane")

Download and unzip the Project Files for Project 4 from Canvas in the Assignments section.

2. Project Layout

The `Project-4` directory contains the following files:

- `DistanceVector.py` - This is the only file you will modify. It is a specialization (subclass) of the Node class that represents a network node (i.e., router) running the Distance Vector algorithm, which you will implement.
- `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, only the format.
- `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; basically a convenient wrapper for `run_topo.py` and `output_validator.py`.

[1] https://en.wikipedia.org/wiki/Distance-vector_routing_protocol for more background information.
[2] https://en.wikipedia.org/wiki/Path_vector_routing_protocol for more background information.

3. To Dos:

There are a few TODOs in DistanceVector.py:

A. ***Review the methods already implemented in Node.py***.
   a. Because DistanceVector subclasses Node, consider how you might use these existing methods to complete the rest of the ToDos in this list.
   b. **Do NOT modify Node.py**.

B. ***Decide on how each node will represent its distance vector***.
   a. Consider what might be the simplest data structure to keep track of path weights (i.e., the distance vector).
   b. 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).

C. ***Implement the Bellman-Ford algorithm***.
   a. Similarly to Project 2, each Node will:
      i. send out an initial message to its neighbors
      ii. process messages received from other nodes
      iii. send updates to other nodes as needed
   b. Initially, a node only knows of:
      i. itself and that it is reachable at cost 0,
      ii. its neighbors and the weights on its links to its neighbors
   c. NOTE: a node's links are **unidirectional**.
   d. NOTE: the Bellman-Ford algorithm implementation should terminate naturally without external intervention.

D. ***Write a logging function*** that is specific to your distance vector structure.
   a. You can use the logging helper files to take care of the bulk of the logging.
   b. You should assume that the logging function only knows itself.
      i. **Do NOT access the topology for logging**; logging should happen at the Node level.

4. Testing and Debugging

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

```
./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 sharing new topologies with log outputs. Topologies with format errors will get an error back when you try to run them.

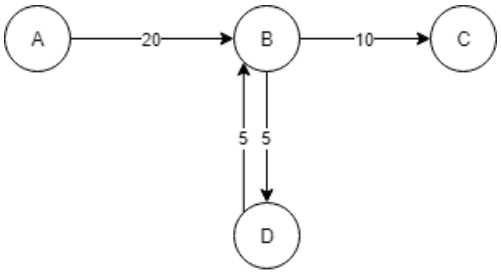
We've included four good topologies for you to use in testing and one bad topology to demonstrate invalid topology. **The provided topologies do not cover all the edge cases; your code will be graded against more complex topologies**.

5. Assumptions and clarifications

A. Node behavior:

a. The direction of a link indicates how **traffic** will flow; two nodes connected with a link ***may pass messages regardless of traffic direction***.

i. Example: Node B has an incoming link from Node A, but has no outgoing link to Node A, Node B will send its distance vector to node A to "advertise" other nodes it can reach (Nodes C and D).



b. A Node's distance vector is comprised of the nodes it can reach via its outgoing links (***including*** to itself at distance = 0).

i. A Node will never advertise a negative distance to itself. (Important for negative cycles.)

c. A Node advertises its distance vector to its ***upstream*** neighbors.

d. Nodes do **not** implement poison-reverse.

B. Edge and Path weights:

a. Edge weight values may be between **-50 and 50, inclusive**.

b. There is no upper limit for path weights.

c. The lower limit for path weights is "-99", which is equivalent to "negative infinity" for this project.

C. Negative cycles:

a. A Node can forward traffic through a negative cycle.

b. Negative cycles are a series of directed links that originate and terminate at a single node, where the sum of the link weights is less than 0.

i. This can lead to a negative "count-to-infinity" problem. Therefore, your implementation must be able to detect negative cycles in order to be able to terminate on its own.

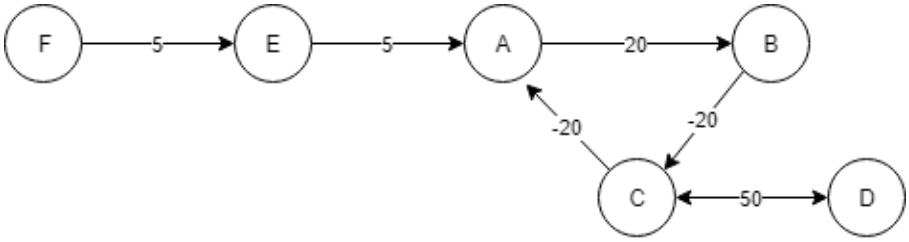
ii. Any node that can reach a destination node and infinitely traverse a negative cycle en route will set the distance to that node to -99.

1. Your implementation only needs to detect and record these traversals appropriately; it does not need to mitigate them.
2. Extra resource: Professor Vigoda explains Negative Weight Cycles and how to detect them, which is Lecture 4, Parts 2-7 of GATech's "Introduction to Graduate Algorithms" course on Udacity.[3] (Parts 6 & 7 are Bellman-Ford specific.)

iii. A Node can advertise a negative distance for other nodes (but not for itself).

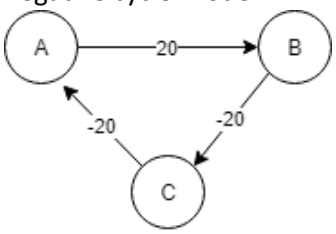
iv. A Node that receives an advertisement with a distance of -99 from a downstream neighbor should also assume that it can reach the same destination at infinitely low cost (-99).

v. *Example*: Traffic from Node F to Node D can route through A->B->C->A indefinitely to reach an extremely low (very negative) value.



[3] https://classroom.udacity.com/courses/ud401/lessons/10046800612/concepts/37924e09-63b6-4357-ab2c-824ed7c89838

c. A Node will ***not*** forward traffic destined to itself.
    i. Example: The below topology will ***not*** result in a count-to-infinity problem, as there are no possible pairs of source and destination nodes where traffic could indefinitely traverse a negative cycle. Node A will not forward traffic for Node A, and similarly for Nodes B and C.



D. Topologies used in grading:
    a. We will be using many topologies to test your project. This includes but is not limited to:
        - topologies with and without cycles (loops), including odd length cycles
        - topologies of varying sizes, including topologies with more than 26 nodes
        - topologies with nodes with names longer than one character
        - topologies with multiple paths to different nodes
        - topologies that include any combination of positive weights, zero weight, and negative weight
        - topologies with negative cycles, meaning a node may reach another at infinitely low cost
        - topologies with Nodes that do not have incoming or outgoing links
            - All nodes will be connected but:
                - some may have both incoming and outgoing links
                - some may only have incoming links
                - some may only have outgoing links
    b. We will NOT test your submission against the following topologies (which means your algorithm does not need to account for them):
        - topologies with more than one link from *the same origin to the same destination* (multi-graphs)
        - topologies with portions of the network disconnected from each other (partitioned networks)
        - topologies with a valid path between two indirectly linked nodes with no cycle with an actual total cost of ≤ -99 (topologies will respect that -99 is “negative infinity” for this project)

6. 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
```

7. 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 its 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!

8. 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, leave alphabetization on so that your classmates can use the `diff` tool to see if you are getting the same log outputs as they are.

9. What to turn in

Submit DistanceVector.py in a .zip with your GTID the project number. This will take the form GTLogin_p4.zip where GTLogin should be replaced with your ID you use to log into Canvas (e.g., smith7_p4.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_p4.zip file.

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

A. **Ensure that your submission self-terminates.** If your submission runs indefinitely (i.e. contains an infinite loop) or throws an error at runtime, it will not receive full credit. Manually killing your submission via console commands or interrupts is NOT an acceptable means of termination.
B. **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.)
C. **Ensure your logs are formatted properly**. Logging 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 manually to make sure it matches the requested format. (See the TODO comment for logging located in DistanceVector.py for format details.)
    a. Incorrectly formatted logs will fail the autograder and *will receive no credit*. We will not be manually inspecting incorrectly named/formatted/etc. logs due to the number of students in the class.
D. **Ensure your solution generates completely correct output.** Partial credit for individual topologies will not be awarded, even if the distance vector logs are "mostly correct."
E. **Check your submission after uploading.** As usual, we do not accept resubmissions past the stated deadlines.

10. Rubric

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 (4 total)	For correct Distance Vector results (log file) on the provided topologies.
90 pts	Unannounced Topologies (4 total)	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 some edge cases.

There is no partial credit for individual topologies; each topology is either "passed" or "failed".

As with previous projects in this course, due to the size of the class, we will not accept resubmissions, modifications to old submissions past the deadline, etc.

11. 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, from the "Assumptions and Clarifications", note: " 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)."