# Distance Vector

## Table of Contents

## PROJECT GOAL

In the lectures, we learned about Distance Vector (DV) routing **protocols,** one of the two classes of routing protocols. DV protocols, such as RIP, use a fully distributed algorithm to find shortest paths by solving the Bellman-Ford equation at each node. In this project, we will develop a distributed Bellman-Ford algorithm and use it to calculate routing paths in a network. This project is similar to the Spanning Tree project, 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. Some distance vector routing protocols, that operate at higher levels (like BGP), must make routing decisions based on business valuations. 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.

## Part 0: Getting Started

We 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")

## Part 1: Files Layout

The `DistanceVector` directory contains the following files:

- `DistanceVector.py` - This is the only file we 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 we will implement.
- `Node.py` - Represents a network node, i.e., a router.
- `Topology.py` - Represents a network topology. It is 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.
- `*Topo.txt` - These are valid topology files that we will pass as input to the run.sh script (see below). Topologies should end with ".txt".
- `BadTopo.txt` - This is an invalid topology file, provided as an example of what not to do, and so we can see what the program says if we 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` – A helper script that runs some basic system checks, the topology, and the validator, a wrapper for `run_topo.py` and `output_validator.py`.

## Part 2: TODOs

There are a few TODOs in DistanceVector.py:

A. ***Review the methods already implemented in Node.py***.
   a. Because DistanceVector is a subclass of Node, consider how we might use the existing methods to complete 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. 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 the distance vector structure.

   a. We should use the *self.add_entry* function to help with logging.
   b. We should assume that the logging function only knows itself.
      i. **Do NOT access the topology for logging**; logging should happen at the Node level.

## Part 3: Testing and Debugging

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

```
./run.sh *Topo
```

Substitute the correct, desired filename for `*Topo`. Don't use the .txt suffix on the command line. This will execute the implementation of the algorithm in `DistanceVector.py` on the topology defined in `*Topo.txt` and log the results (per the logging function) to `*Topo.log`.

**NOTE**: We should *not* include the full filename of the topology when executing the run.sh script. For example, to run the algorithm on `topo1.txt` we should only specify topo1 as the argument to `run.sh`.

For this project, we may create as many topologies as we wish and share them on Ed Discussion. We encourage sharing new topologies with log outputs. Topologies with format errors will get an error back when we try to run them.

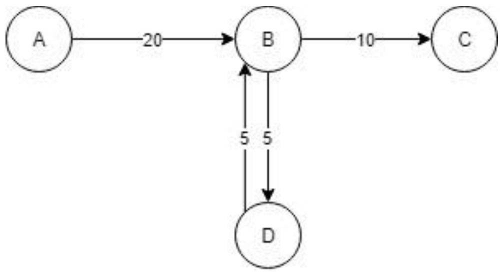
We've included four good topologies for we to use in testing and one bad topology to demonstrate invalid topology. **The provided topologies do not cover all the edge cases; the code will be graded against more complex topologies**.

## Part 4: 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. The edge weight value type is an **integer**.

c. There is no upper limit for path weights.

d. 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, the implementation must be able to detect negative cycles to **terminate on its own**.

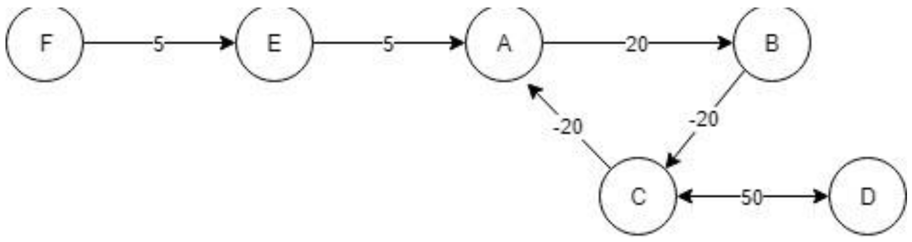
ii. Any node that can reach a destination node and infinitely traverse a negative cycle enroute will set the distance to that node to -99.

1. The implementation only needs to detect and record these traversals appropriately; it does not need to mitigate them.

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

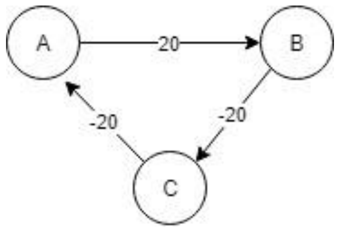
iv. A Node that receives an advertisement with a distance of -99 from a downstream neighbor should also assume that it can reach the same destination at infinitely low cost (-99).

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.



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 the 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 the submission against the following topologies (which means the 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) o topologies that do not require intermediate steps
   (such as a topology with a single node)
   - 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)

## Part 5: Correct Logs for Provided Topologies

Below are the correct final logs for the provided topologies. We are providing them to help we 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:(A,0) (B,1) (C,3) (D,3) B:(B,0) (A,1) (C,2) (D,2) C:(C,0) (B,2) (A,3)
(D,0) D:(D,0) (C,0) (B,2) (A,3) E:(E,0) (D,-1) (C,-1) (B,1) (A,2)
```

SingleLoopTopo:

```
A:(A,0) (D,5) (E,6) (B,6) (C,16) B:(B,0) (A,2) (D,7) (C,10) (E,0)
C:(C,0) D:(D,0) (E,1) (B,1) (A,3) (C,11) E:(E,0) (B,0) (A,2) (D,7)
(C,10)
```

SimpleNegativeCycle:

```
AA:(AA,0) (AD,-2) (AE,-1) (AB,0) (CC,-99) AB:(AB,0) (AA,-1) (AD,-3)
(CC,-99) (AE,-2) AD:(AD,0) (AE,1) (AB,2) (AA,1) (CC,-99) AE:(AE,0)
(AB,1) (AA,0) (AD,-2) (CC,-99) CC:(CC,0) (AB,0) (AA,-1) (AD,-3) (AE,-
2)
```

ComplexTopo:

```
ATT:(ATT,0) (CMCT,-99) (TWC,-99) (GSAT,-8) (UGA,-99) (VONA,-11) (VZ,-3)
CMCT:(CMCT,0) (TWC,-99) (ATT,1) (VONA,-10) (GSAT,-7) (UGA,-99) (VZ,-2)
DRPA:(DRPA,0) (EGLN,1) (GT,-1) (UC,-1) (CMCT,-99) (TWC,-99) (ATT,13) (OSU,-1)
(VONA,2) (GSAT,5) (UGA,-99) (PTGN,1) (VZ,10) EGLN:(EGLN,0) (GT,-2) (UC,-2)
(DRPA,1) (CMCT,-99) (OSU,-2) (TWC,-99) (ATT,13) (PTGN,0) (VONA,3) (GSAT,5)
(UGA,-99) (VZ,11) GSAT:(GSAT,0) (VONA,-3) (VZ,5) (UGA,-99) (ATT,7) (CMCT,-99)
(TWC,-99) GT:(GT,0) (UC,0) (EGLN,2) (OSU,0) (DRPA,3) (PTGN,2) (CMCT,-99)
(VONA,5) (TWC,-99) (ATT,15) (VZ,13) (GSAT,7) (UGA,-99) OSU:(OSU,0) (UC,0)
(GT,0) (EGLN,2) (PTGN,2) (VONA,5) (DRPA,3) (VZ,13) (GSAT,7) (CMCT,-99)
(ATT,15) (UGA,-99) (TWC,-99) PTGN:(PTGN,0) (OSU,-1) (UC,-1) (GT,-1) (EGLN,1)
(VONA,3) (VZ,11) (GSAT,5) (DRPA,2) (ATT,13) (UGA,-99) (CMCT,-99) (TWC,-99)
TWC:(TWC,0) (CMCT,-99) (ATT,1) (VONA,-10) (VZ,-2) (GSAT,-7) (UGA,-99)
UC:(UC,0) (GT,0) (EGLN,2) (OSU,0) (PTGN,2) (DRPA,3) (VONA,5) (CMCT,-99)
```

```
(VZ,13) (GSAT,7) (TWC,-99) (ATT,15) (UGA,-99) UGA:(UGA,0) (ATT,50) (CMCT,-99)
(TWC,-99) (GSAT,42) (VONA,39) (VZ,47) VONA:(VONA,0) (VZ,8) (GSAT,2) (ATT,10)
(UGA,-99) (CMCT,-99) (TWC,-99) VZ:(VZ,0) (ATT,2) (CMCT,-99) (TWC,-99) (GSAT,-
6) (UGA,-99) (VONA,-9)
```

## Part 6: Spirit of the Project

The goal of this project is to implement a simplified version of a network protocol using a distributed algorithm. This means that the 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.