DV Routing Project Walkthrough

CS 6250

Distance Vector Routing

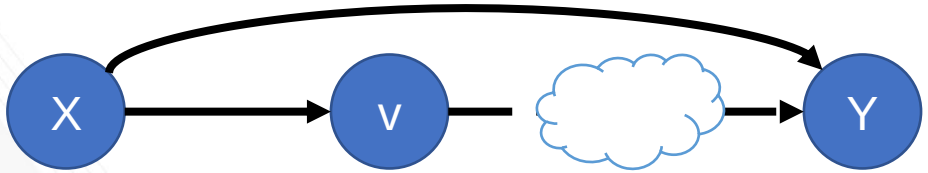
- The lectures discussed **Distance Vector (DV) routing protocols**, one of the two classes of routing protocols
- DV protocols, such as RIP, use a fully distributed algorithm that finds shortest paths by solving the Bellman-Ford equation *at each node*
- In this project, you will implement a distributed Bellman-Ford algorithm and use it to calculate routing paths in a network
  - This is similar to the STP project, except we are solving a **routing** problem, not a **switching** problem
- Kurose and Ross: "Where LS algorithms use global information, the DV algorithm is ***iterative, asynchronous,*** and ***distributed***."
  - Nodes will NOT have a complete map of the network

Distance Vector Routing

- “Pure” DV routing protocols use hop count (aka number of links to be traversed) to determine distance between nodes.
- Some DV routing protocols at higher levels (such as BGP for interdomain routing) must make routing decisions based on business relationships as well as hop count.
  - These protocols are sometimes called **Path Vector** protocols.
- This project will explore these routing decisions and challenges by using weighted links (including negatively weighted links) in our network topologies.
  - Each node represents an individual Autonomous System (AS), and the weight of the link represents business relationships between ASes.
  - Links are **directed** – originating at one Node and terminating at the other

Bellman-Ford Algorithm

- Each Node X updates its own distance vector using the Bellman Ford "equation": $D_x(y) = min\{c(x,v) + D_v(y), D_x(y)\}$
- Example:



- Breaking it down:
  - $D_x(y)$ is what Node X thinks the distance is from Node X to Node Y
    - This is the value that's getting updated. Read this like " x = x+1" – it's an assignment, not an equation.
  - $c(x,v)$ is the cost of the link between Node X and Node v.
    - Node v *must* be one of Node X's downstream neighbors.
  - $D_v(y)$ is what Node v thinks the distance is from Node v to Node Y.
    - This is what Node v advertised to Node X. Node X has no idea how Node v gets to Node Y – hence the cloud on the diagram.
  - *min* is the "minimum" function – take the smaller value.

Resources

- Some helpful resources to learn more about the Bellman-Ford algorithm:
  - Wikipedia
  - Kurose and Ross textbook

- And of course, we expect that you watched/read the course lectures relating to the topic. ☺

Project 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 scrip.

Project Files (continued)

- **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 .

TODOs in DistanceVector.py

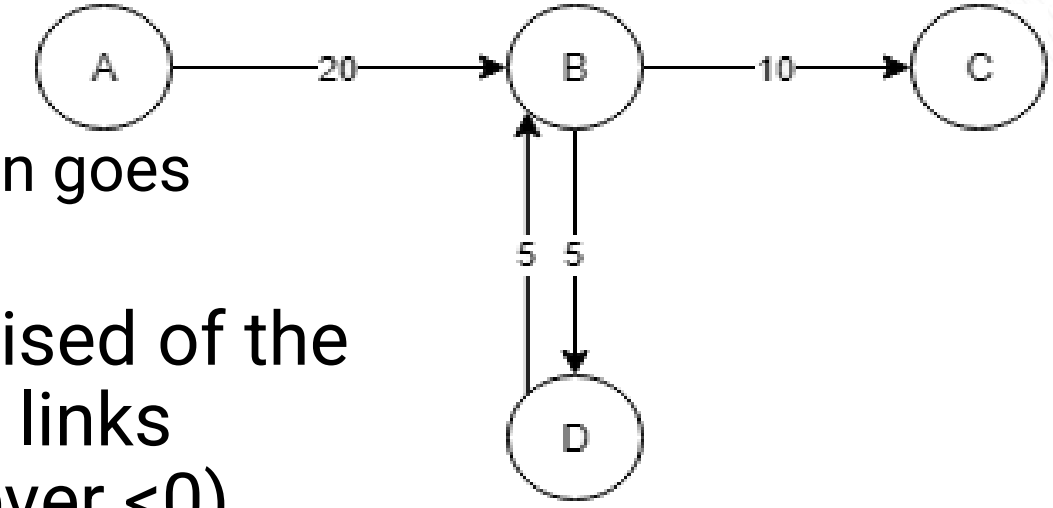
- Design and implement a **data structure** for each node to use to keep track of its distance vector (path weights).
  - This should be local to the node, defined in the `init` function, accessible via the self object (ex: “self.mylist”)
  - This does not need to be complicated – simple is probably best.
  - You’ll also have to design and implement what will constitute a *message* for your implementation
- Implement the Bellman-Ford algorithm
  - Each node will need to a) send out initial messages to its neighbors, b) process messages received from other nodes, and c) send updates as needed.
  - Initially, a node only knows of a) itself and that it is reachable at cost 0, and b) its neighbors and the weights on its links to its neighbors
    - Take a look at Node.py for more info on how each Node will know this information

TODOs in DistanceVector.py

- Write a logging function that is specific to your particular data structures.
  - You can use the logging helper files to take care of the bulk of the logging.
  - Logging should happen at the Node level; **do NOT access the topology for logging.**

Clarifications and assumptions: Node behavior

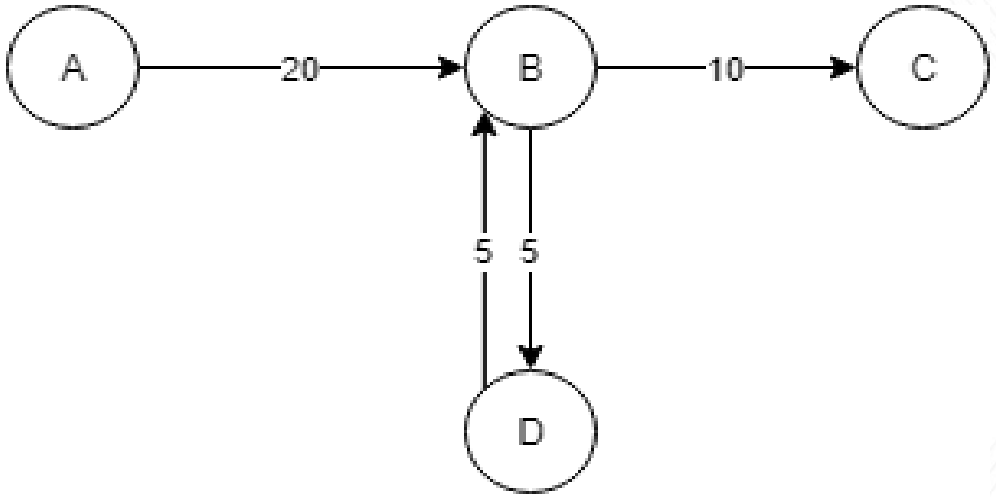
- Links are **unidirectional**; they show how traffic flows, but messages can be passed in either direction.
  - Node B can send messages to Node A, even though the link direction goes the other way.



- A Node's distance vector is comprised of the nodes it can reach via its outgoing links (**including** itself at distance = 0; never <0)
- Nodes do **not** implement poison reverse or split horizon.

Clarifications and assumptions: weights

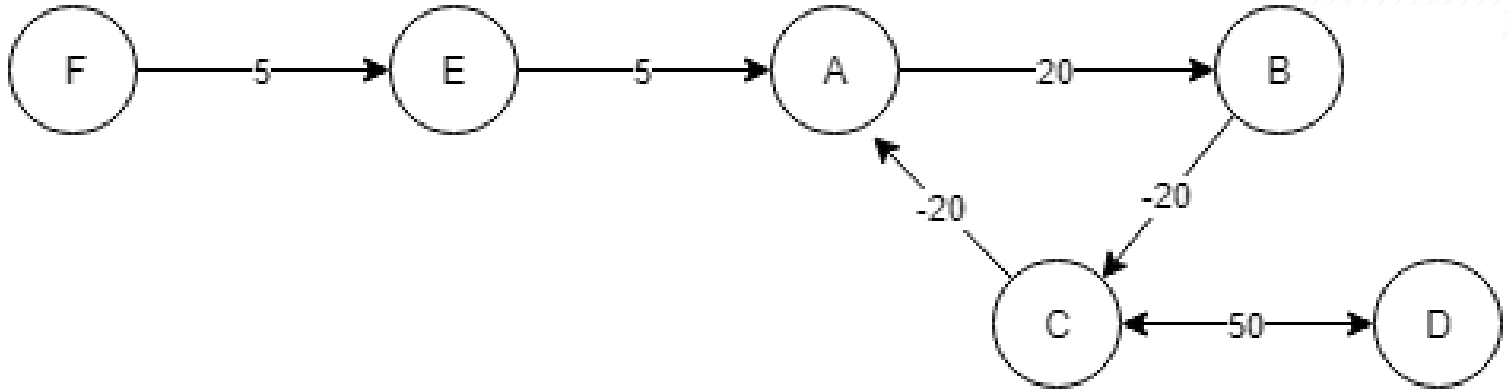
- Edge weight values may be between -50 and 50, inclusive.
- There is no upper limit for path weights.
  - “path” is a series of connected edges.
  - Example: A->B->D has path weight 25.



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

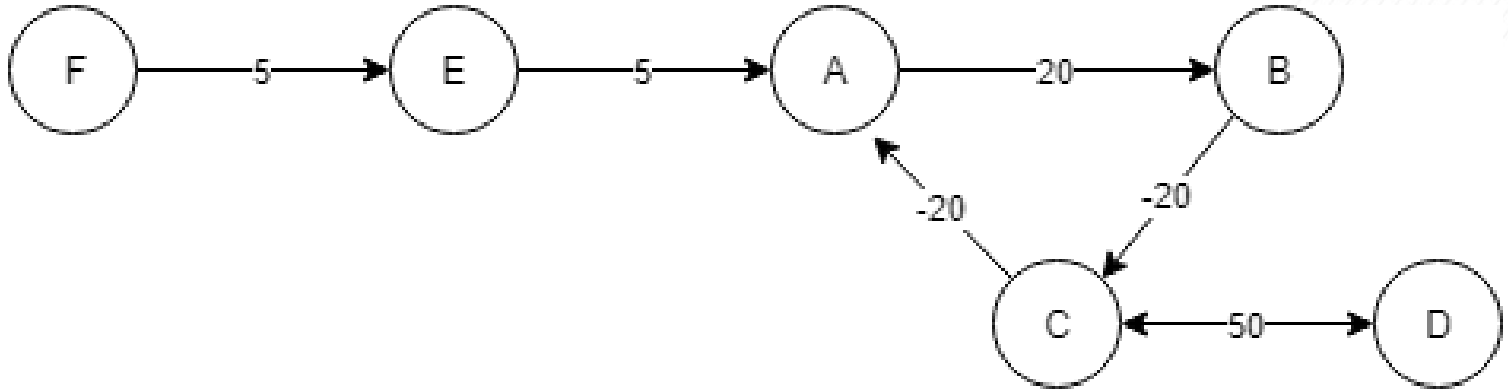
Negative Cycles

- Negative weights can interfere with shortest path algorithms when there is a *negative cycle*
- Example:
  - What's the shortest path from E to D?



Negative Cycles

- Negative weights can interfere with shortest path algorithms when there is a *negative cycle*
- Example:
  - What's the shortest path from E to D?
    - E->A->B->C->D is 5+20+(-20)+50 = **55**
    - E->A->B->C->A->B->C->D = 5+20+(-20)+(-20)+20+(-20)+50 = **35**
    - **We have a negative count-to-infinity problem**

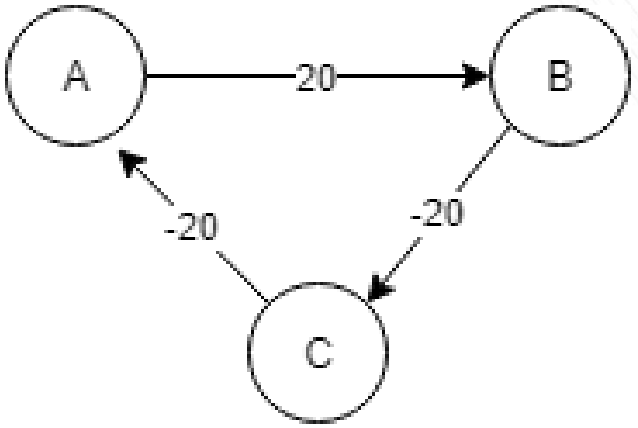


Negative Cycles

- “-99” represents negative infinity for our project.
  - Any node that can reach a destination node and infinitely traverse a negative cycles en route will set the distance to that node to -99.
- Your implementation only needs to detect and record these traversals; it does not need to mitigate them.
  - Prof Vigoda talks about this, specifically with the Bellman-Ford algorithm, in one of the “Intro to Graduate Algorithms” videos on Udacity (linked in project description)
- A Node can advertise a negative distance for other nodes (but not itself.)
  - A Node will **not** forward traffic destined to itself

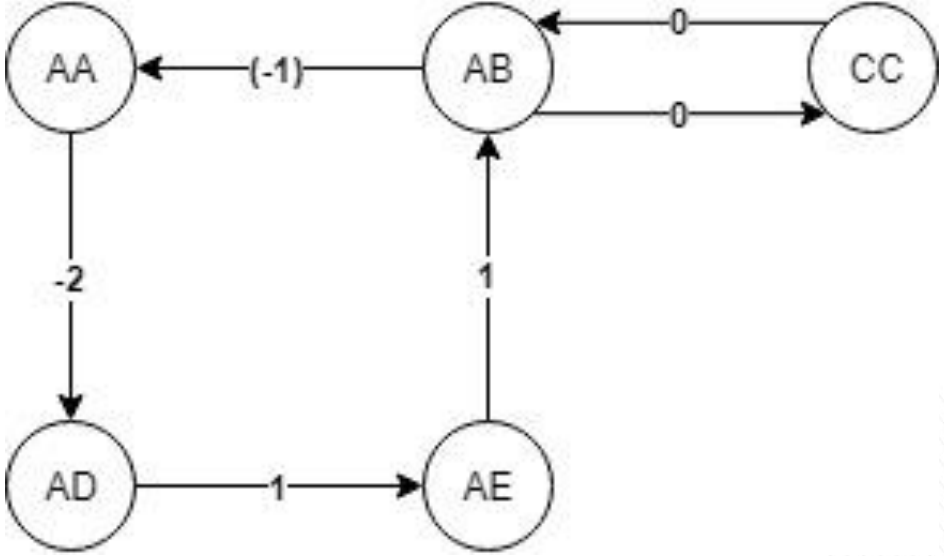
Negative Cycles

- Example:
  - There is no “count-to-infinity” problem here.
    - A to C will follow A->B->C = 0, because C will not forward it back to A (C is the destination)
    - C to A will follow C->A = -20, because A will not forward traffic meant for A (A is the destination)
  - Note that negative paths do exist.
    - B should advertise to A that B has a path of -20 to C.
  - Notice that this example is a subgraph of the earlier Negative Cycle example.

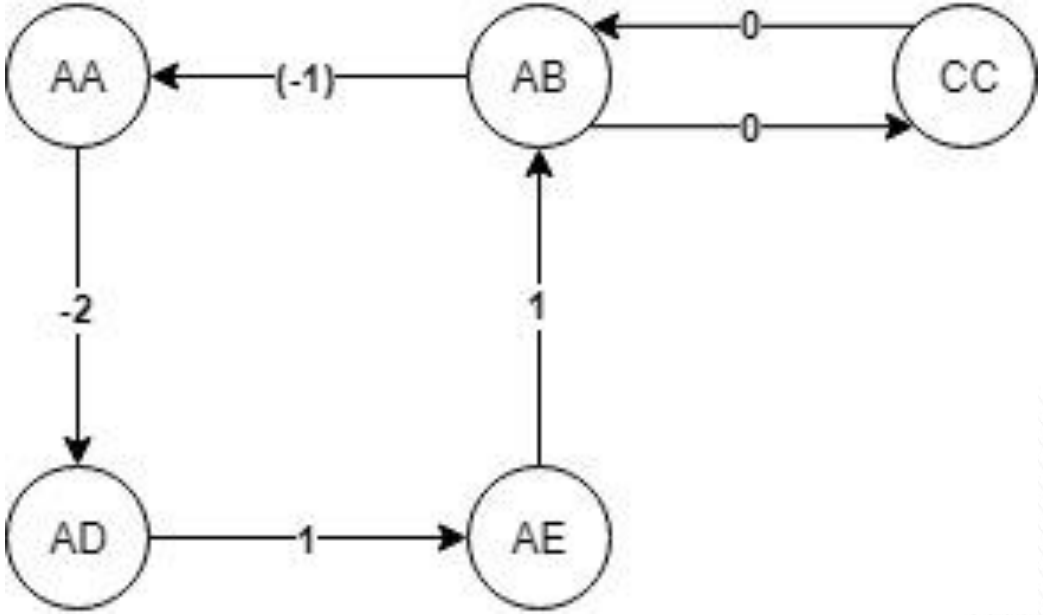


Example: Initialization

- See P4_Example.xlsx
  for the following example.
  - Note that there are different iterations on different tabs



Example: Initialization



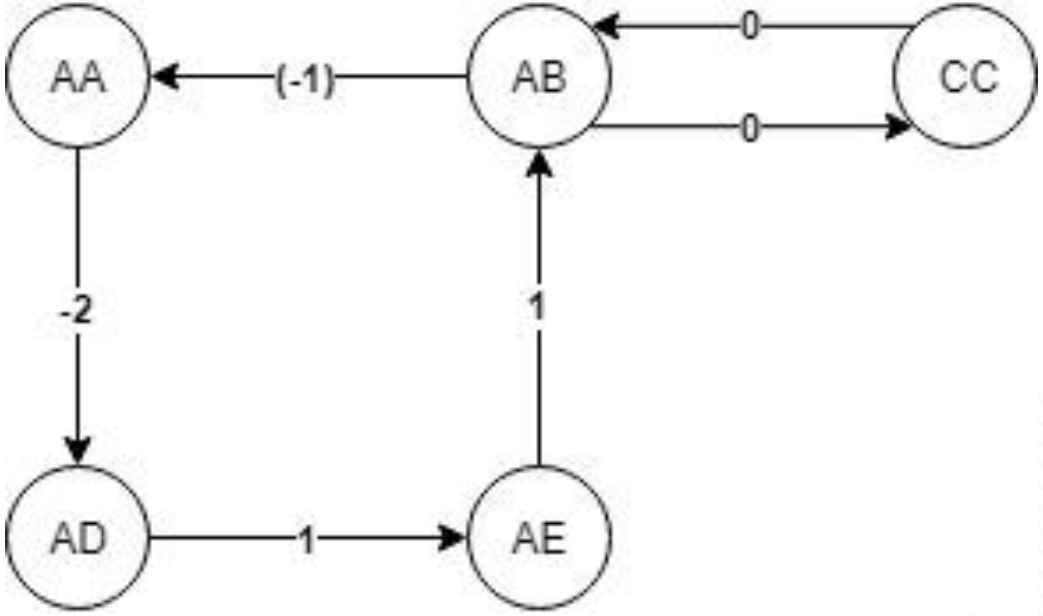
Msg format: (origin node, origin node distance vector)

Messages are sent to incoming links/upstream neighbors only.

This is because a node only cares about other nodes it can reach so it only wants messages from its outgoing nodes.

Node	Vector	Messages	
AA	AA0	AD, AD0	
AB	AB0	AA, AA0	CC, CC0
AD	AD0	AE, AE0	
AE	AE0	AB, AB0	
CC	CC0	AB, AB0	

Example: 1st Iteration



Processing: Looping through the nodes in the distance vector
Add cost to the message origin to the weight for that node

Process all messages before going on to the next node (loop through the queue)
So AB will process the message from CC before moving onto AD

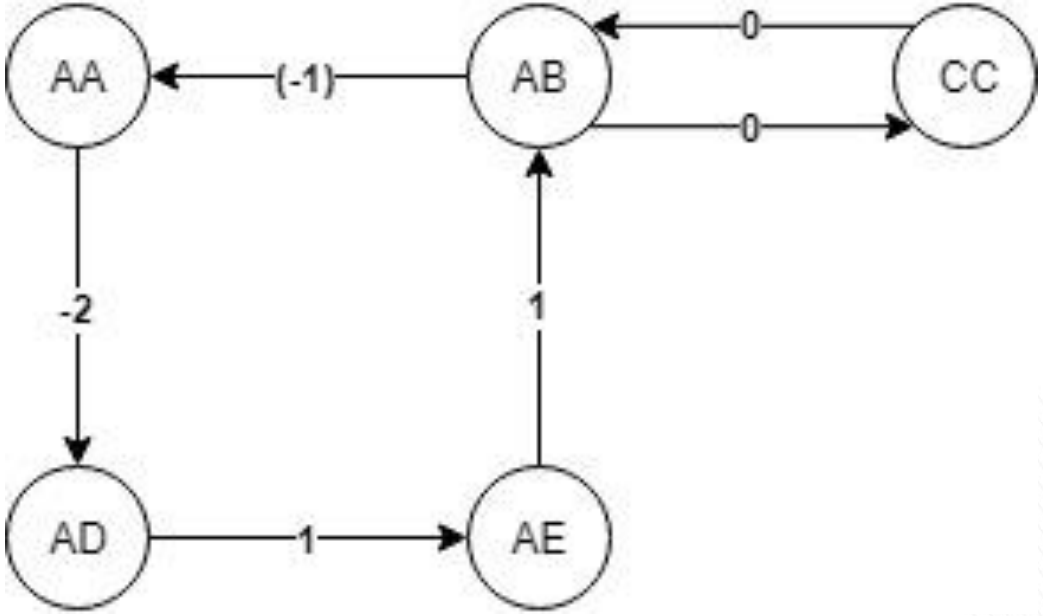
If messages cause a change, then the node will send messages to incoming links

Nodes will ignore distances to themselves
because they want traffic that arrives at themselves to stay there

	Before Processing			Processing: cost to origin + NodeWeight	After Processing		
Node	**Vector**	**Messages: Origin, NodeWeight**			**New Vector**	**Messages**	
AA	AA0	AD, AD0		AD:-2+0=-2, add to vector	AA0, AD-2	AD, (AD0, AE1)	
AB	AB0	AA, AA0	CC, CC0	AA:-1+0=-1,add;CC:0+0=0, add	AB0, AA-1, CC0	AA, (AA0, AD-2)	CC, (CC0, AB0)
AD	AD0	AE, AE0		AE:1+0=1,add	AD0, AE1	AE, (AE0, AB1)	
AE	AE0	AB, AB0		AB:1+0=1,add	AE0, AB1	AB, (AB0, AA-1, CC0)	
CC	CC0	AB, AB0		AB:0+0,add	CC0, AB0	AB, (AB0, AA-1, CC0)	

Example: 1.5st Iteration

Example: AA will add AE to its Vector since it learns about AE from AD.
Cost is the addition of cost from AA to AD + cost from AD to AE or (-2) + (1)



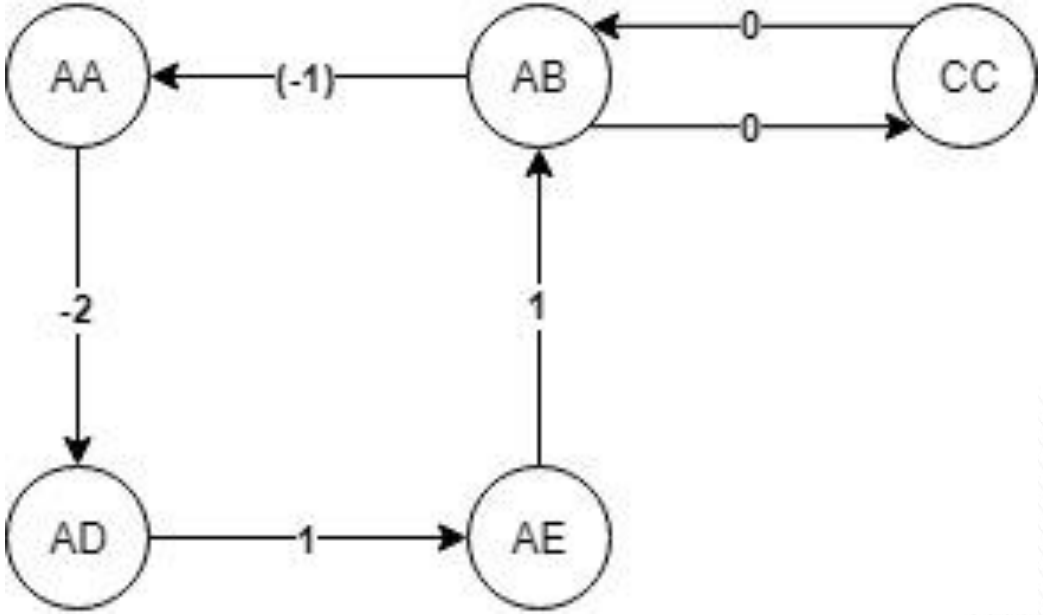
	Before Processing			Processing: cost to origin + NodeWeight	After Processing		
Node	**Vector**	**Messages: Origin, NodeWeight**			**New Vector**	**Messages**	
AA	AA0, AD-2	AD, (AD0, AE1)		AD:-2+0=-2, no change; AE:-2+1=-1,add	AA0, AD-2, AE-1	AD, (AD0, AE1, AB2)	
AB	AB0, AA-1, CC0	AA, (AA0, AD-2)	CC, (CC0, AB0)	AA:-1+0=-1,no change; AD:-1+-2=-3, add; CC:0+0=0,no change; AB – self, ignore	AB0, AA-1, CC0, AD-3	AA, (AA0, AD-2, AE-1)	CC, (CC0, AB0, AA-1)
AD	AD0, AE1	AE, (AE0, AB1)		AE:1+0=1,no change; AB:1+1=2,add	AD0, AE1, AB2	AE, (AE0, AB1, AA0, CC1)	
AE	AE0, AB1	AB, (AB0, AA-1, CC0)		AB:1+0=1,no change; AA:1+-1=0,add;CC:1+0=1, add	AE0, AB1, AA0, CC1	AB, (AB0, AA-1, CC0, AD-3)	
CC	CC0, AB0	AB, (AB0, AA-1, CC0)		AB:0+0=0,no change; AA:0+-1=-1, add; CC: self, ignore	CC0, AB0, AA-1	AB, (AB0, AA-1, CC0, AD-3)	

Example: 2nd, 3rd, 4th Iteration

Node	Vector	Messages: Origin, NodeWeight	
AA	AA0, AD-2, AE-1	AD, (AD0, AE1, AB2)	
AB	AB0, AA-1, CC0, AD-3	AA, (AA0, AD-2, AE-1)	CC, (CC0, AB0, AA-1)
AD	AD0, AE1, AB2	AE, (AE0, AB1, AA0, CC1)	
AE	AE0, AB1, AA0, CC1	AB, (AB0, AA-1, CC0, AD-3)	
CC	CC0, AB0, AA-1	AB, (AB0, AA-1, CC0, AD-3)	

Node	Vector	Messages: Origin, NodeWeight	
AA	AA0, AD-2, AE-1, AB0	AD, (AD0, AE1, AB2, AA1, CC2)	
AB	AB0, AA-1, CC0, AD-3, AE-2	AA, (AA0, AD-2, AE-1, AB0)	CC, (CC0, AB0, AA-1, AD-3)
AD	AD0, AE1, AB2, AA1, CC2	AE, (AE0, AB1, AA0, CC1, AD-2)	
AE	AE0, AB1, AA0, CC1, AD-2	AB, (AB0, AA-1, CC0, AD-3, AE-2)	
CC	CC0, AB0, AA-1, AD-3	AB, (AB0, AA-1, CC0, AD-3, AE-2)	

Node	Vector	Messages: Origin, NodeWeight	
AA	AA0, AD-2, AE-1, AB0, CC0		
AB	AB0, AA-1, CC0, AD-3, AE-2	AA, (AA0, AD-2, AE-1, AB0, CC0)	CC, (CC0, AB0, AA-1, AD-3, AE-2)
AD	AD0, AE1, AB2, AA1, CC2		
AE	AE0, AB1, AA0, CC1, AD-2		
CC	CC0, AB0, AA-1, AD-3, AE-2		



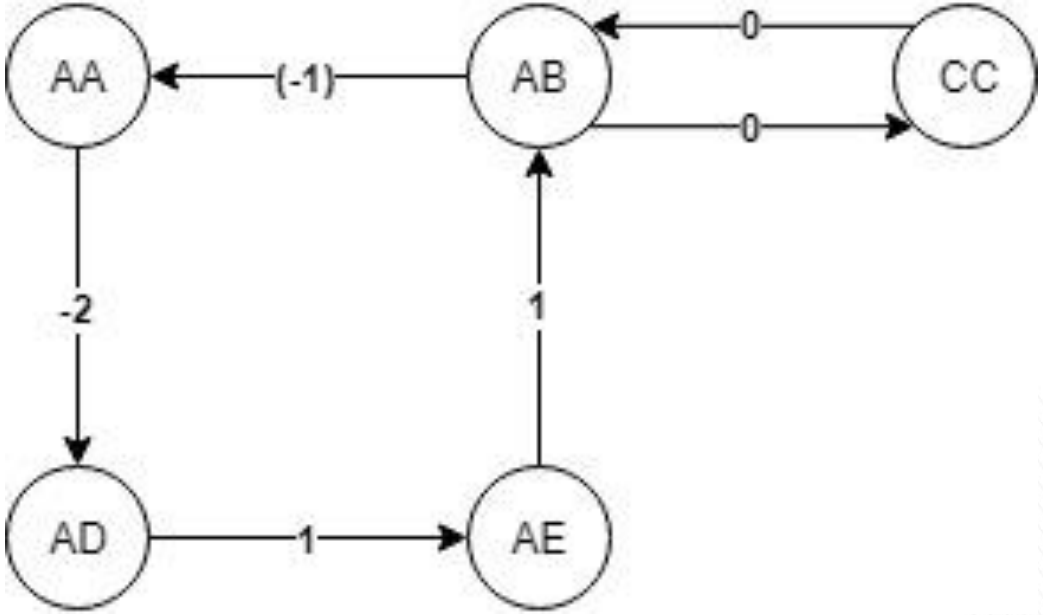
Remember, only send out messages if there is a change in the DV
Only AA and CC had changes, AA and CC only have AB as incoming link
So only AB receives messages
(Try doing the math for AB, AD, AE, CC to see no changes)

All nodes have learned of all reachable nodes (no more "add" showing up)*

*Remember, in some topologies, some nodes won't be reachable
Meaning not all nodes are in every other nodes' DV
Example: A<--B-->C: C won't be A's DV

Something weird is happening. The message from AA makes AB go around the loop to get to CC.
Keep going to find out why.

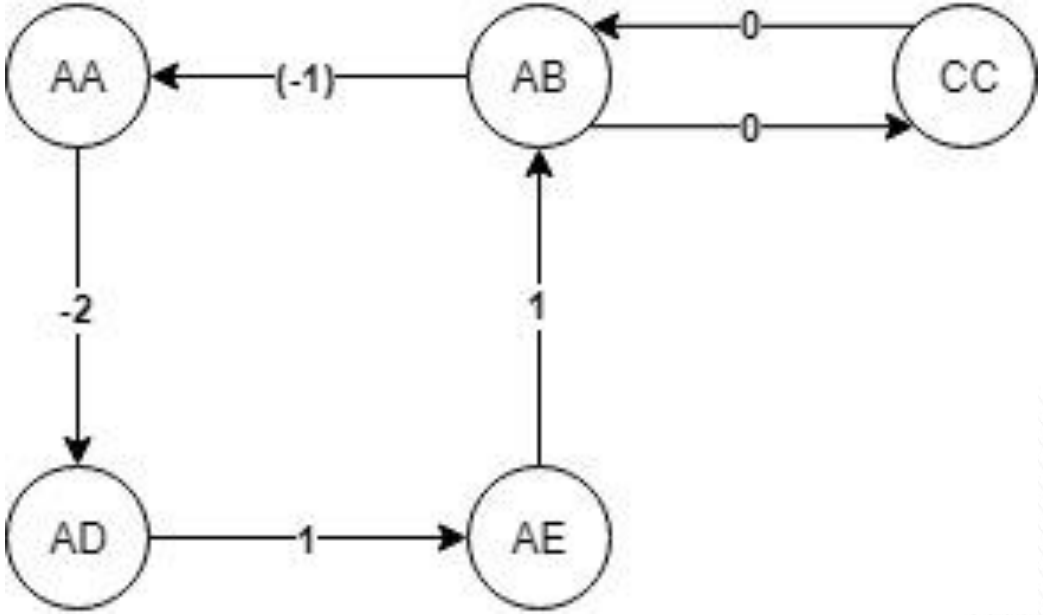
Example: 5th Iteration



Negative cycle started to develop when AB evaluated message from AA
AB has link to AA at -1 and sees it can reach CC at a cheaper cost by going through AA instead of directly to CC
AB has lost that it will be routing traffic back through AB to get to CC. No way to tell. Starting negative count to infinity
Node CC won't ever actually receive traffic because of the infinite loop
(AB will send to AA, go around the loop and again come to AB)
There is a negative cycle because AA->AD->AE->AB->AA is a cost of -1
and it can be traversed an infinite number of times, always decreasing by 1
In assumptions, you may stop at -99 and mark it as a negative cycle.
Keep doing the processing until you see that the cost to CC continually decreases for every node in this topology.
Note: in a larger topology, it won't be every node that needs to set the weight to CC to -99.
What if there were another node, DD, off to the right of CC? Say AB<--(0)-->CC<--(-4)--DD? Then DD wouldn't have a cost -99.

Node	Vector	Messages: Origin, NodeWeight	Processing: cost to origin + NodeWeight	New Vectors	New Messages
AA	AA0, AD-2, AE-1, AB0, CC0				
AB	AB0, AA-1, CC-1, AD-3, AE-2				
AD	AD0, AE1, AB2, AA1, CC2				AE, (AE0, AB1, AA0, CC0, AD-2)
AE	AE0, AB1, AA0, CC1, AD-2	AB, (AB0, AA-1, CC-1, AD-3, AE-2)	AB: 1+0=no change; AA: 1+-1=0, no change; CC:1+-1=0, update; AD: 1+-3=-2,no change; AE=self, ignore.	AE0, AB1, AA0, CC0, AD-2	
CC	CC0, AB0, AA-1, AD-3, AE-2	AB, (AB0, AA-1, CC-1, AD-3, AE-2)	AB: 0+0=0,no change; AA: 0+-1=-1, no change; CC: self, ignore; AD: 0+-3=-3, no change; AE:0+-2=-2, no change	CC0, AB0, AA-1, AD-3, AE-2	

Example: 6th Iteration

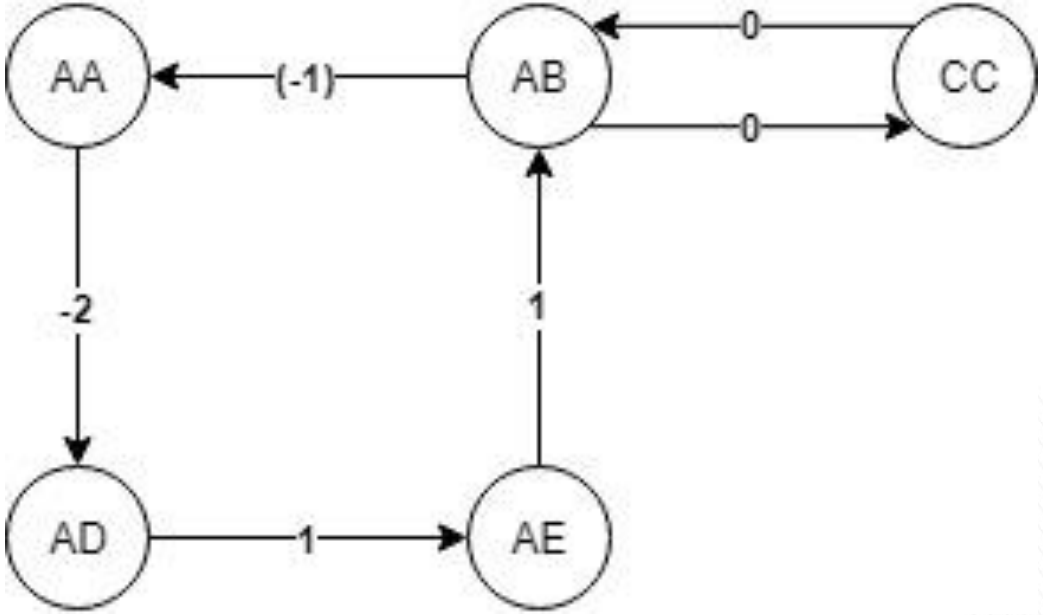


Look at how AE-CC changed. It'll keep going. We stop at -99.
AB should recognize that it keeps going down in cost
AB will eventually get to CC-99, stay there, don't send out new message.
Other nodes already reached -99 for CC, so just don't keep adding costs.
So every node will check if cost = -99 for CC, don't send out messages anymore or change

Node	Vector	Messages: Origin, NodeWeight	Processing: cost to origin + NodeWeight	New Vectors	New Messages
AA	AA0, AD-2, AE-1, AB0, CC0				AD, (AD0, AE1, AB2, AA1, CC1)
AB	AB0, AA-1, CC-1, AD-3, AE-2				
AD	AD0, AE1, AB2, AA1, CC2	AE, (AE0, AB1, AA0, CC0, AD-2)	AE:1+0=1, no change; AB: 1+1=2, no change; AA: 1+0=no change; CC: 1+0 =1, update; AD: self, ignore	AD0, AE1, AB2, AA1, CC1	
AE	AE0, AB1, AA0, **CC0**, AD-2				
CC	CC0, AB0, AA-1, AD-3, AE-2				

Example: Xth Iteration

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



Logging questions

- Grading is performed on the generated log file, not std out.
- Autograder checks for presence of previous iterations (no specific quantity), and the final iteration for correctness
- Each vector's nodes don't need to be in a particular order
  - Both of these would be acceptable
    - A:A0,C16,B6,**E6,D5**
    - A:A0,C16,B6,**D5,E6**
  - Helpers.py alphabetizes after every round.

Topologies

- What we may test your code against:
  - 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

- What we may test your code against:
  - 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

Topologies

- What we will NOT test your code against (so you don't need to account for these):
  - topologies with more than one link from *the same origin to the same destination* (multi-graphs)
    - Example: You won't see
      A, B, 1, B, 2
  - 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)
  - Improperly formatted topologies

Spirit of the Project

- The goal is a ***distributed algorithm*** that does NOT depend on global knowledge. This rules out:
  - Declaring global variables or using static variables
  - Directly accessing the topology object
  - Using the provided code to allow one Node to access another Node's internal state
- The autograder is set up to throw an error if a student's submission attempts to directly access the topology object. (If this error is thrown, the submission will receive no credit.)

Final comments

- Your submission must terminate on its own.
- Print statements can drastically slow down implementations, particularly inefficient ones. Submissions should take <10s per topology.
  - But we do encourage temporarily using print statements for debugging!
- Logging is very important – this is how we check if your implementation is correct.
  - Match the format in the TODO comment in DistanceVector.py
- There's a FAQ at the end of the project description
- Use the office hours to ask more questions.