ITCS 6166/8166 Computer Communication Networks

Project – 3 Report

Distance Vector Routing Algorithm

Team Members:

1. Arunkumar Bagavathi (800888454)
2. Chetan Borse (800936059)

Programming Language: Python Version 3.5

(Note:** The code was tested only on Python 3.5**)

Objective:

The objective of this project is to implement the distance vector routing protocol. Our goal in project is to implement an application that can be run either at several different machines or in a single machine. Implementation also handles link cost changes.
Instead of implementing the exact distance vector routing protocol described in the textbook, we have developed a variation of the protocol. In this protocol, each host sends out the routing information to its neighbours at a certain frequency (once every 15 seconds), regardless whether the information has changed since the last announcement. This strategy improves the robustness of the protocol. For instance, a lost message will be automatically recovered by later messages. In this strategy, typically a host re-computes its distance vector and routing table right before sending out the routing information to its neighbours.

Distance Vector Routing Algorithm:

Distance-vector routing protocols use the Bellman–Ford algorithm, Ford–Fulkerson algorithm, or DUAL FSM(in the case of Cisco Systems' protocols) to calculate paths.
A distance-vector routing protocol requires that a router inform its neighbours of topology changes periodically. Compared to link-state protocols, which require a router to inform all the nodes in a network of topology changes, distance-vector routing protocols have less computational complexity and message overhead.

Routers using distance-vector protocol do not have knowledge of the entire path to a destination. Instead they use two methods:

1. Direction in which router or exit interface a packet should be forwarded.
2. Distance from its destination

Distance-vector protocols are based on calculating the direction and distance to any link in a network. "Direction" usually means the next hop address and the exit interface. "Distance" is a measure of the cost to reach a certain node. The least cost route between any two nodes is the route with minimum distance. Each node maintains a vector (table) of minimum distance to every node. The cost of reaching a destination is calculated using various route metrics.
Updates are performed periodically in a distance-vector protocol where all or part of a router's routing table is sent to all its neighbors that are configured to use the same distance-vector routing protocol. Once a router has this information it can amend its own routing table to reflect the changes and then inform its neighbors of the changes. This process has been described as 'routing by rumor' because routers are relying on the information they receive from other routers and cannot determine if the information is valid and true.

Poisson Reverse

Count-to-infinity Problem:

The Bellman–Ford algorithm does not prevent routing loops from happening and suffers from the **count-to-infinity problem**. The core of the count-to-infinity problem is that if A tells B that it has a path somewhere, there is no way for B to know if the path has B as a part of it. To see the problem clearly, imagine a subnet connected like A–B–C–D–E–F, and let the metric between the routers be "number of jumps". Now suppose that A is taken offline. In the vector-update-process B notices that the route to A, which was distance 1, is down – B does not receive the vector update from A. The problem is, B also gets an update from C, and C is still not aware of the fact that A is down – so it tells B that A is only two jumps from C (C to B to A), which is false. Since B doesn't know that the path from C to A is through itself (B), it updates its table with the new value "B to A = 2 + 1". Later, B forwards the update to C and since A is reachable through B (From C's point of view), C decides to update its table to "C to A = 3 + 1". This slowly propagates through the network until it reaches infinity (in which case the algorithm corrects itself, due to the relaxation property of Bellman–Ford).

Solution:

Algorithm uses Poison Reverse as a solution to above problem. If Z routes through Y to get to X, then Z tells Y its (Z's) distance to X is infinite (so Y won't route to X via Z).

Project Implementation:

As mentioned in the project requirements, following challenges are handled in our project

- **Link Cost change between the routers**
- **Poisson Reverse method to solve looping problem**

Our project has 2 implementations:

1. Normal execution without Poisson Reverse implementation (**router.py**)
2. Execution with Poisson Reverse (**poisson_router.py**)

We tried to learn how both executions behave among the given routers.

Our implementation takes following order of execution in each router:

1. Open given distances file and read corresponding distance to the neighbors
2. Calculate the minimum distance to every other routers from the currently available router table (initial router table consists of only exponentially large value in each cell)
3. Print the distance and next hop as mentioned in project instructions
4. Sends the distance vector to neighbors if not the vector is converged

In Poisson Reverse implementation, value in the distance vector, to be sent to the neighbor, is assigned exponentially large number if router has to go through the neighbor to reach some other destination.

The above steps are followed every 15 seconds. Router socket keep on listening to the connection all the time except while the router sends some packets to its neighbors

For more details of implementation, please refer source code written in '**router.py**' and '**poisson_router.py**'. It has comments throughout the source code.

Project Execution:

1. To begin the execution:

   python DriverApp.py -f <Network_Config_File> -x <Driver_IP_Addr> -y <Driver_Port> -i <Source_Folder_of_Network_Config_File> -d <Source_Folder_of_Router_Distances_Files> -p <Poisson_Flag>

   where,
   Network_Config_File – Comma Separated file telling IP address and Port of each router
   Poisson_Flag – Boolean flag for Poisson Reverse execution
   Router_Distances_Files – Router files containing router's neighbors and their corresponding distance/cost

   All the above values are set to default values. Default values are:

   - Network_Config_File: RouterNetwork.txt
   - Driver_IP_Addr: 127.0.0.1

- Driver_Port: 8080
- Source_Folder_of_Network_Config_File: NetworkConf
- Source_Folder_of_Router_Distances_File: InputDistances
- Poisson_Flag: True

  All example files are attached in the project

2. Simple execution commands:

   python DriverApp.py -p False

3. Given the above command, the project open '*n*' terminals, where *n* is number of routers in the network. Each terminal is contributed for a single router running in a particular IP address and Port
4. Attached input files contain 4 router. Please refer to the attached distances files: **a.dat, b.dat, c.dat, d.dat**
5. Running the command given in 2 opens 4 terminals as given in Figure-1:

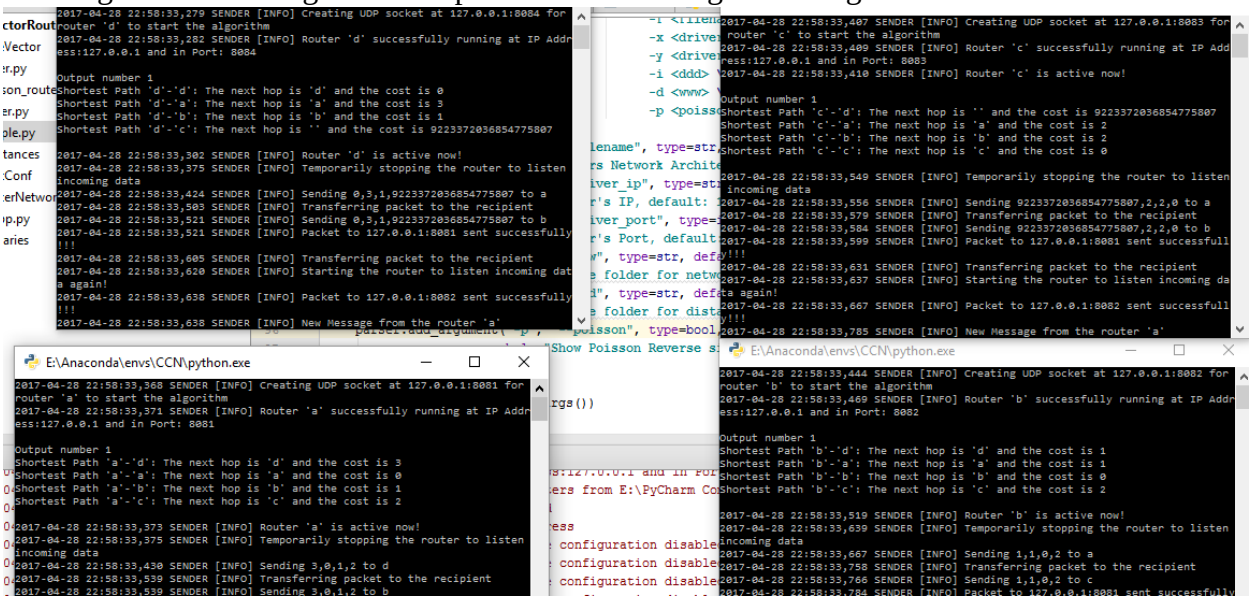

Figure 1: Starting the program opens up terminals

6. Figure-2 gives Output number 3 for execution without Poisson Reverse implementation

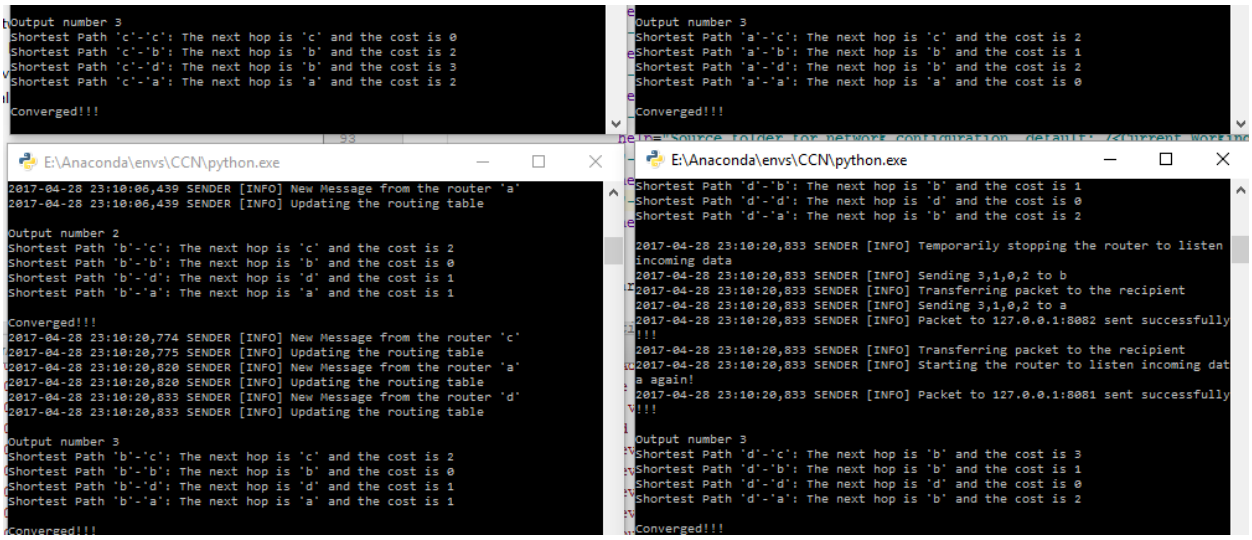

Figure 2: Output 3

7. Figure-3 shows Output-2 of Poisson Reverse Execution:

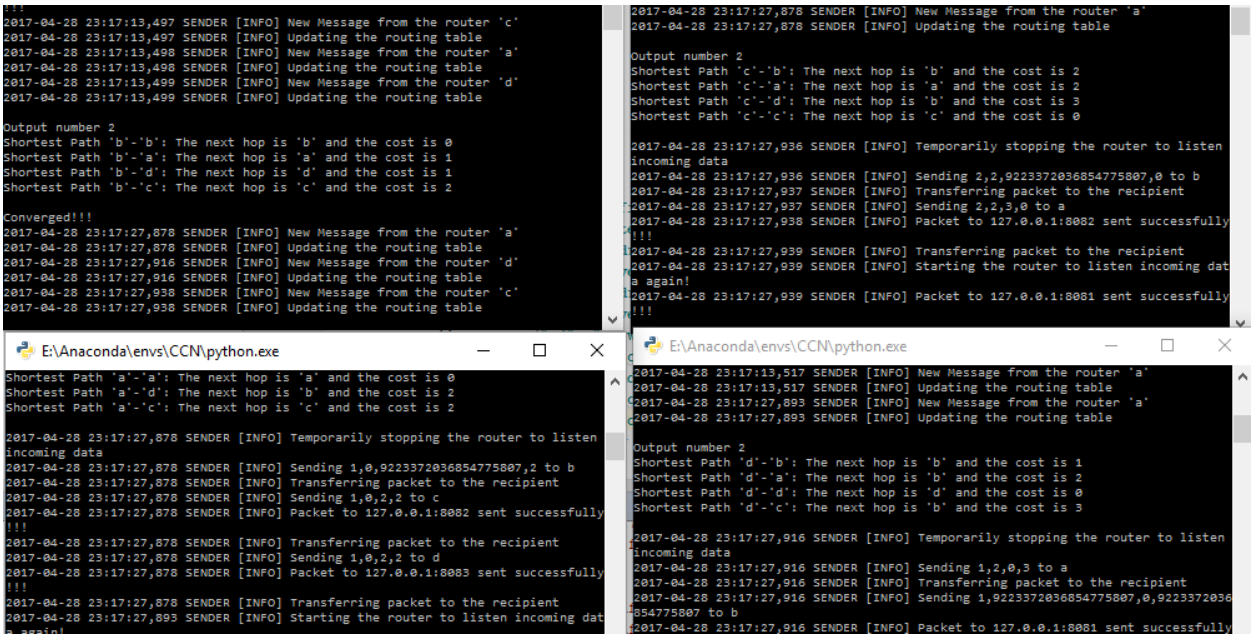

Figure 3: Output-2 of Poisson Reverse execution

Also, distances between routers can be changed at any time. Our program can handle those link cost changes in both Poisson and Poisson less executions

REFERENCE:

https://en.wikipedia.org/wiki/Distance-vector_routing_protocol