COMPUTER COMMUNICATION AND NETWORKS
ITCS 6166/8166

IMPLEMENTING DISTANCE VECTOR ROUTING PROTOCOL

SUBMITTED BY:
Name: Aarti Nimhan
Student ID: 801098198

Implementation of Distance Vector Routing Protocol:-

I implemented the Distance Vector Routing Protocol in java. In distance vector each router is aware of only its neighbors and not the complete network topology. Each router sends its own routing information to each of its neighbors every 15 seconds. To implement this each router would require two threads one for sending packets and another thread in listening mode to receive the packets from its neighbors. For sending and receiving packets I have used UPD (java.net.DatagramPacket and java.net.DatagramSocket).

There are two main classes in this implementation :- *CreateRouter.java* and *SenderThread.java*

The *CreateRouter.java* is the main class which creates one router and assigns it a unique port number. The port numbers for all the routers are read from a file which is passed as a command line argument. This file contains the router id a space and the port number for that router and the next router on the next line. For example *"x 5555".*

The *SenderThread.java* is a thread which polls the routerFile, if changes are made in the file it will update the routing table, compute new vectors using the Bellman Ford equation and send a packet containing the updated information to its neighbors. This is done every 15 seconds and the outputs of the shortest paths discovered so far are displayed.

Flow of Implementation:

- Validates the command line arguments which include the path of the file containing the ports and the router file which contains information of each neighbor and cost to the neighbor.
- Validates the PortNumber File. If this file is valid it creates an *idPortNumbersMap* and maps each router id with its respective port number.
- It also creates a routing table for the current router and initializes it with the cost to self as 0.0 and to all other routers as maximum (since cost is not yet computed).
- It creates two maps namely - *costToEachRouterMap* and *hopRouteMap*. The *costToEachRouterMap* maintains the cost to each router with the current router as source and every other router as destination. Initially this cost is not known so it is initialized to max and cost to self is initialized to 0.0. The *hopRouteMap* maintains the neighbor through which the packet must be routed so as to get a cost mentioned in *costToEachRouterMap.*
- This also creates an instance of SenderThread.java class and starts the thread. When sending the latest computed costs to neighbors, poisoned reverse is implemented to avoid looping scenario.
- In poisoned reverse, if z routes through y to get to destination x, then z will advertise to y that its distance to x is infinity, that is, z will advertise to y that *costToEachRouterMap* for $x = \infty$.
- The main thread of *CreateRouter.java* will now continue forever in listening mode where it acts a receiver to all the packets sent from neighbors.
- Whenever the receiver thread receives a packet from a neighbor it updates the row in its routing table for that neighbor.

Steps for executing the implementation:

1. Open cmd for every router you wish to create.
2. Go to the source folder.

   For example:- cd C:\Workspaces\DistanceVector\DistanceVectorAlgorithm\src

3. Compile the source using javac command.

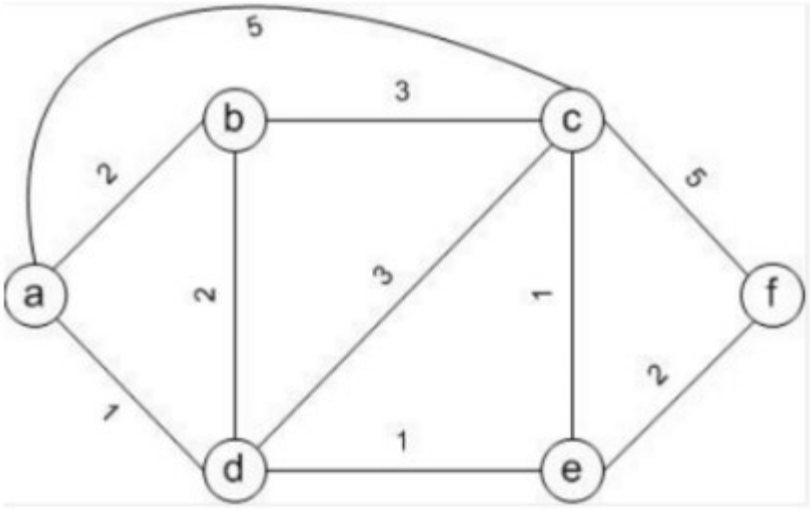
*For example:- javac *.java*

4. Run the *CreateRouter.java* file with the required paths of portFile and router file passed as command line arguments. Both the paths should be separated by a space.

   For example:- java CreateRouter
   C:\Users\DELL\Documents\Coursework\CCN\DistanceVector\portRouter.dat
   C:\Users\DELL\Documents\Coursework\CCN\DistanceVector\6NodeGraph\a.dat

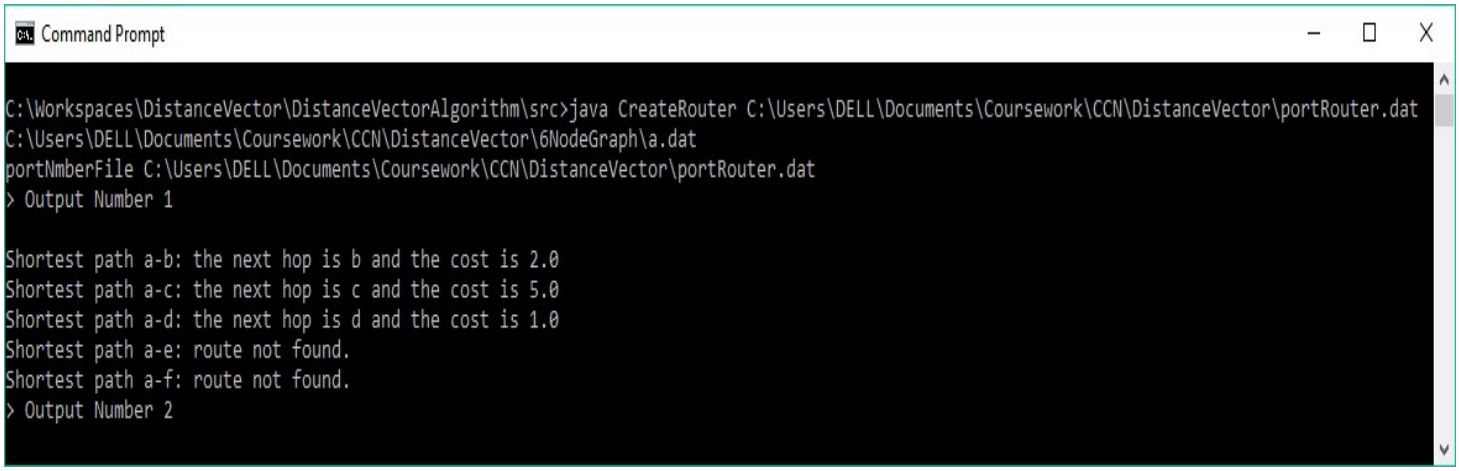
   This will start the router with id a.
5. Repeat the steps 1, 2 and 4 for each router in the network topology.

   Below are screenshots of all routers when executing Distance Vector routing for the following network topology:-

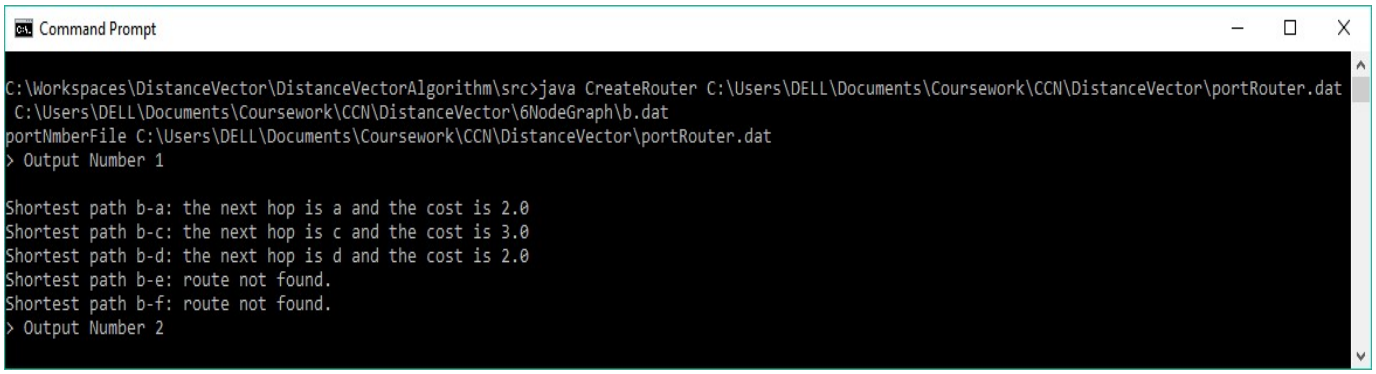


Output 1 at each Router

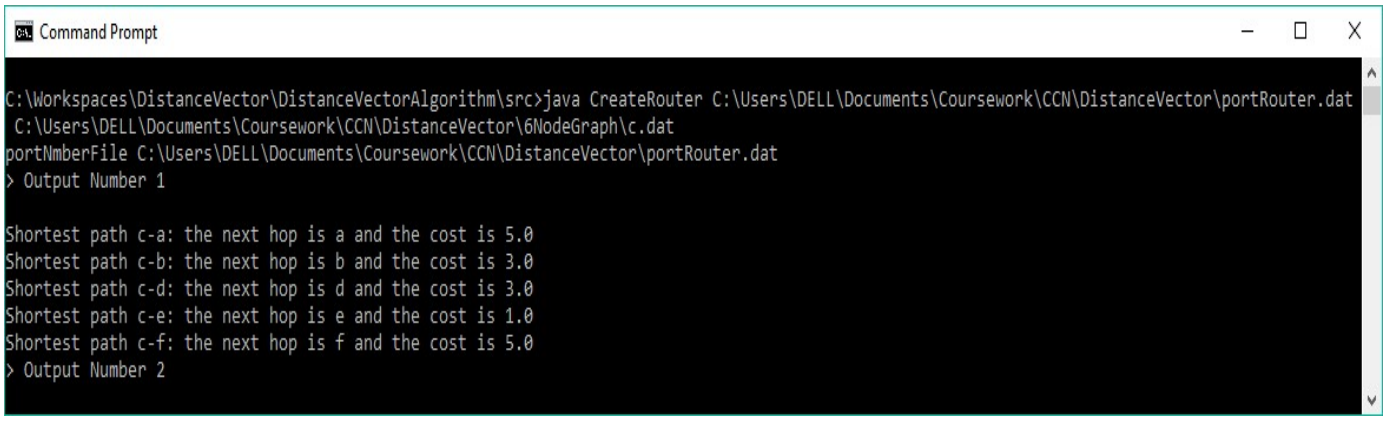
Router a:-



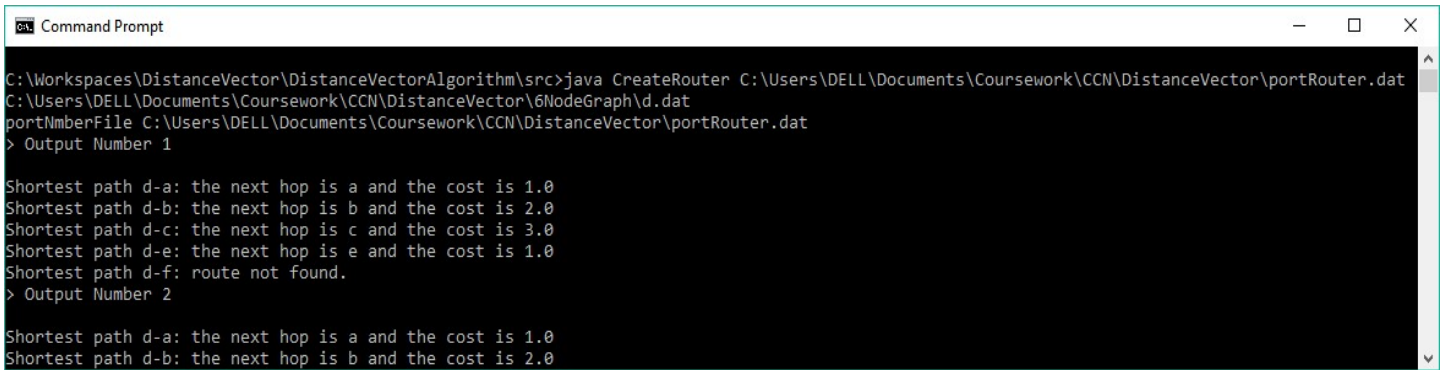
Router b:-



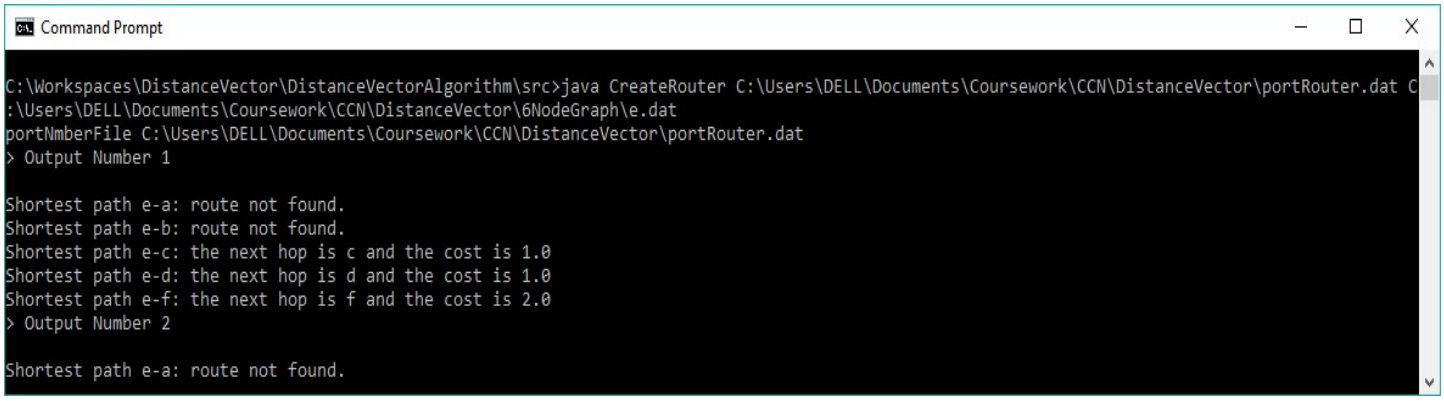
Router c:-



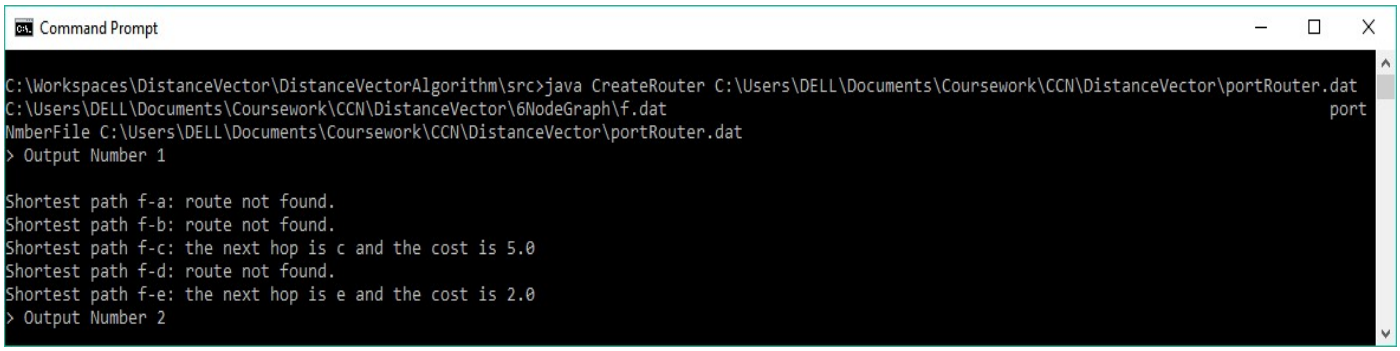
Router d:-



Router e:-

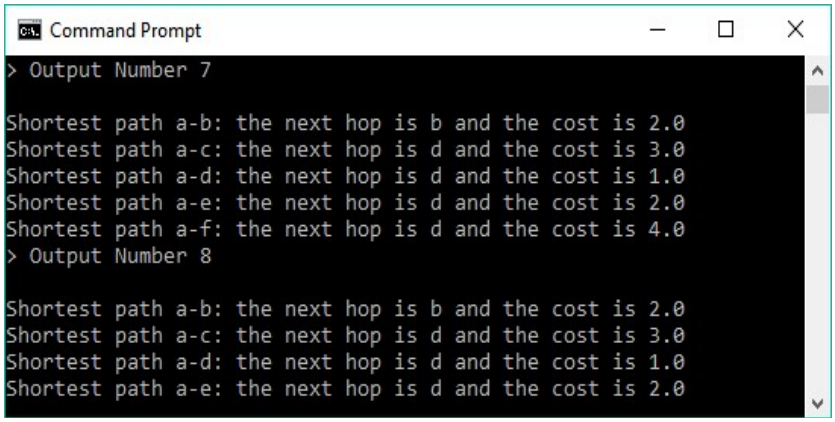


Router f:-

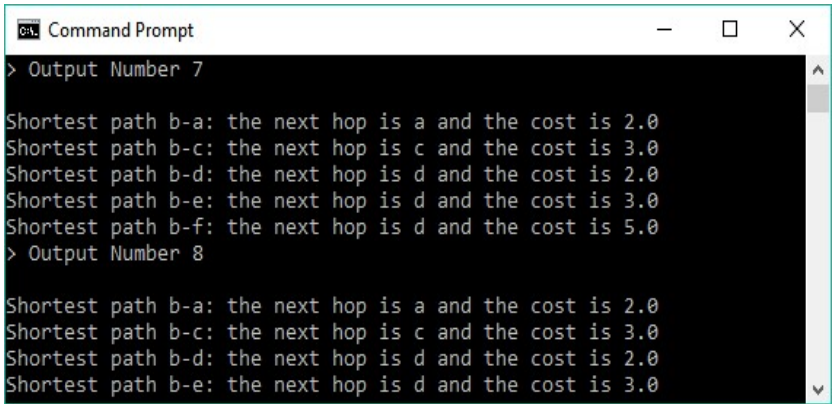


After a few iterations, the output stabilizes. Output for each router.

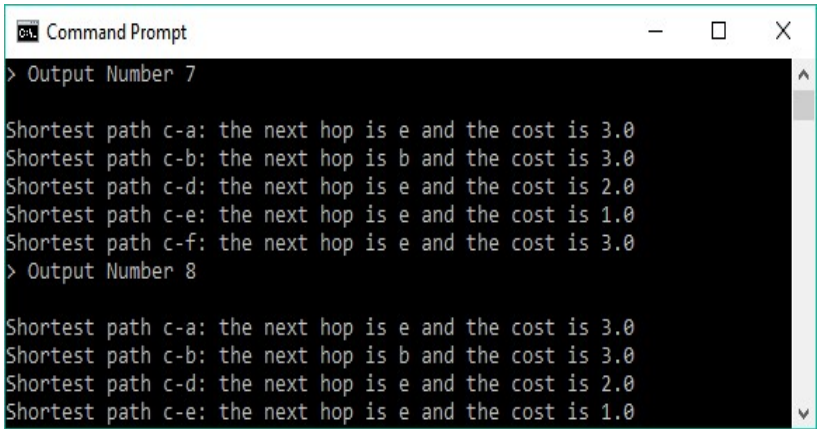
Router a:-



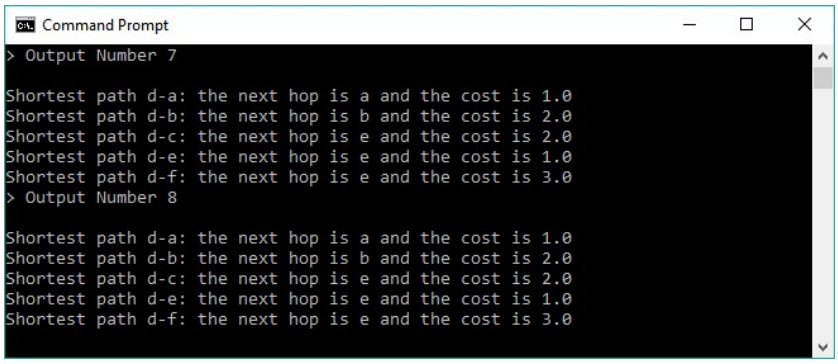
Router b:-



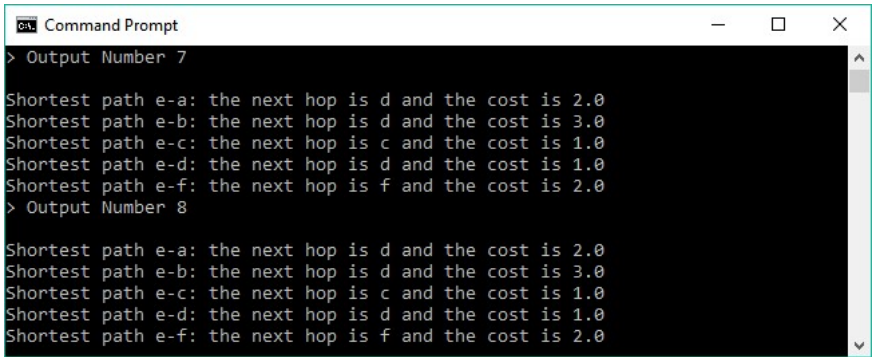
Router c:-



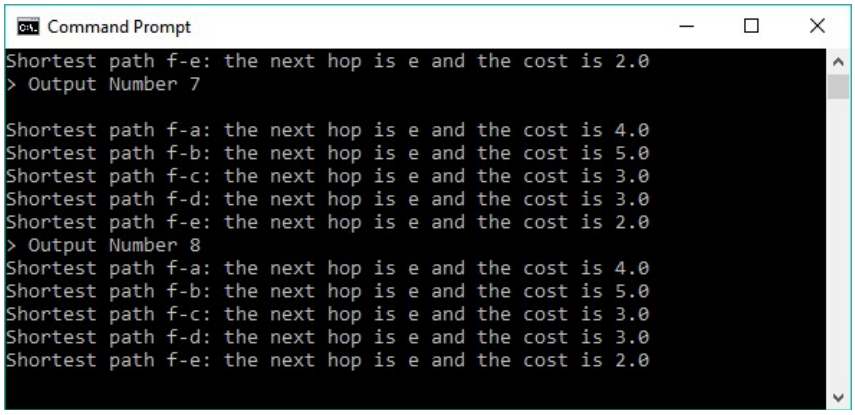
Router d:-



Router e:-

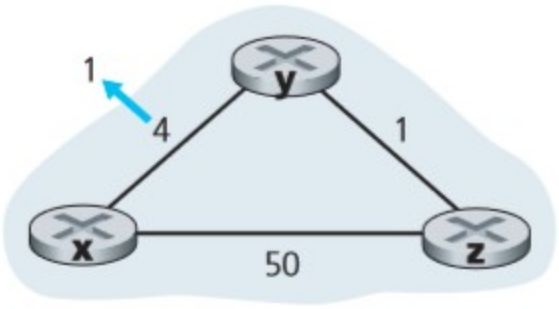


Router f:-

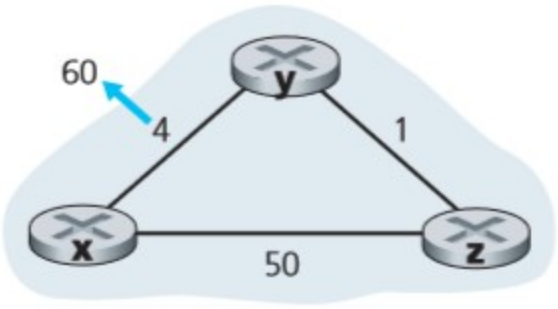


Link state change

To test link state changes we run the Distance vector for fig a. To test Bad News, we then increase the cost of x->y to 60 as shown in fig b. To test Good News we decrease the cost of x->y to 4 again.

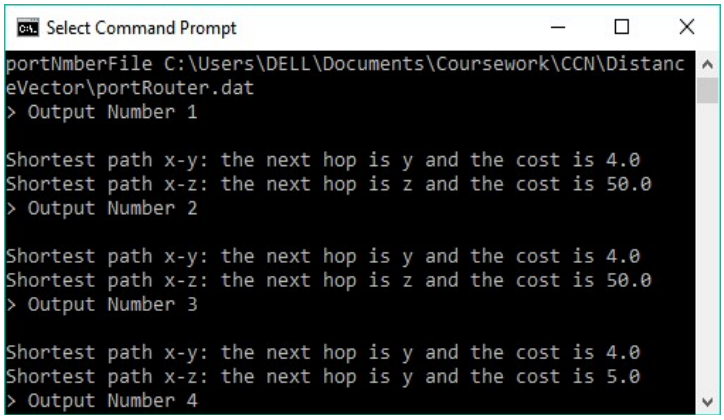


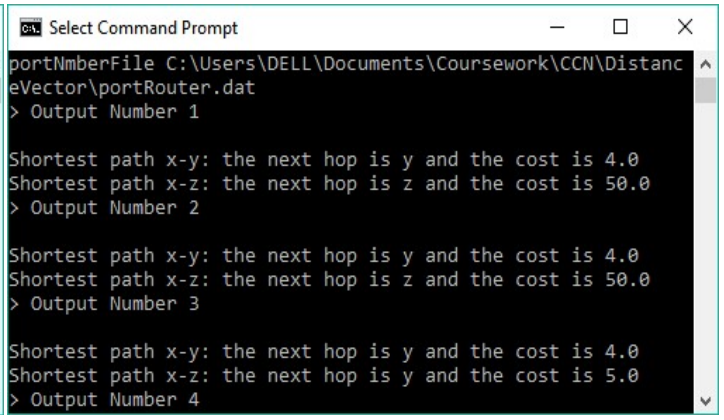
a.



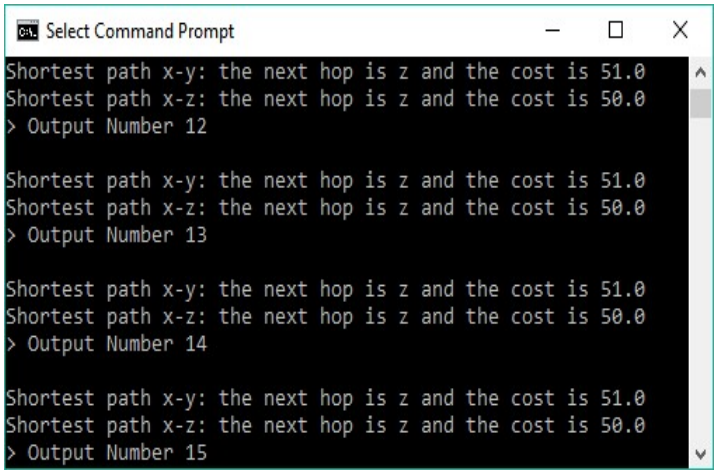
b.

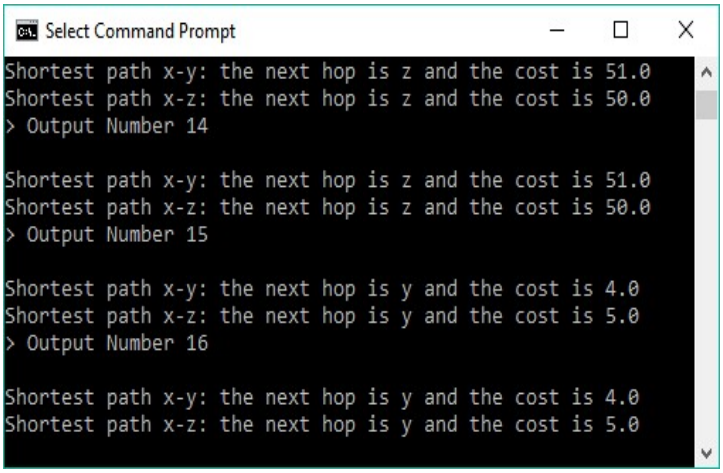
Router x: **Left side: Initial iteration for each router.** **Right side: the output after it stabilizes**



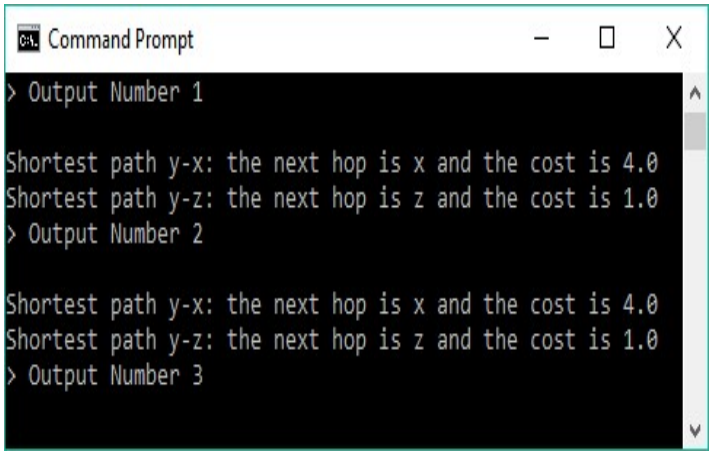


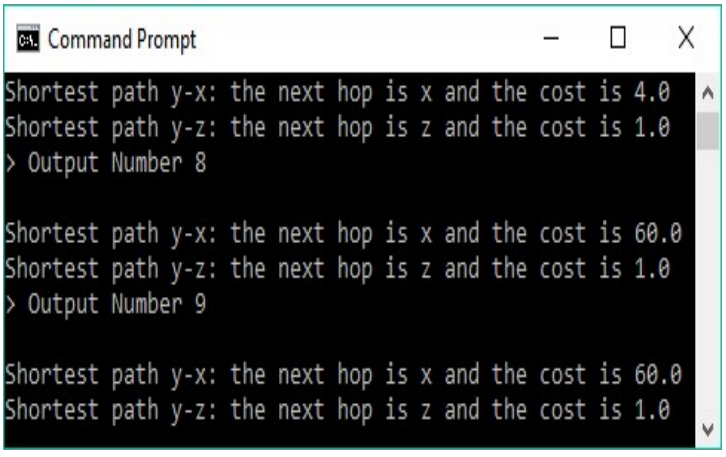
Left side shows Bad News: When x→ y is increased to 60 again. **Right side shows Good News**: When x→y is reduced to 4



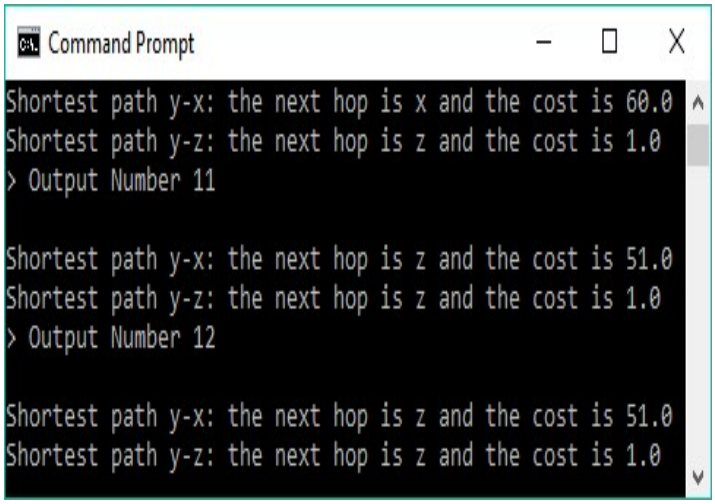


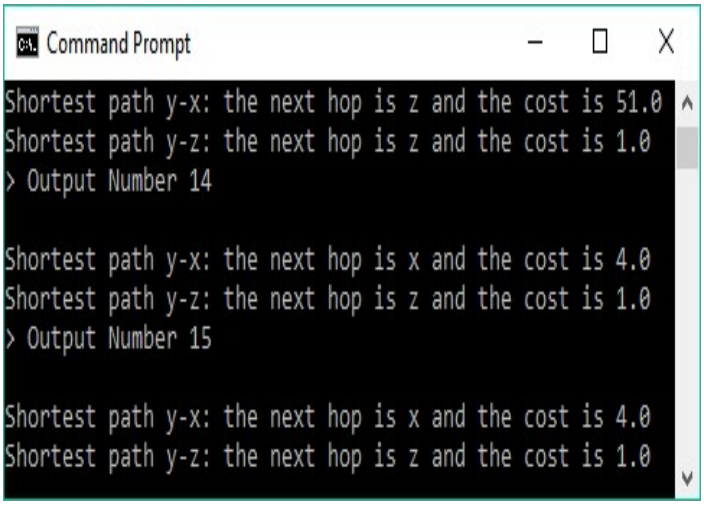
Router y: **Left side: Initial iteration for each router.** **Right side: the output after it stabilizes**



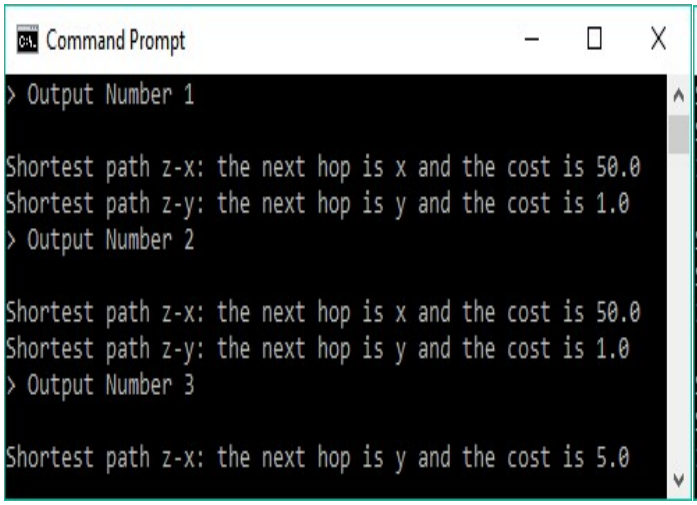


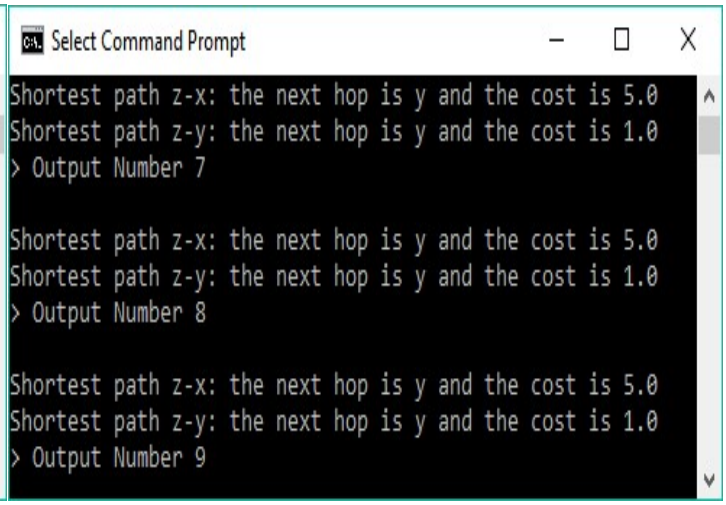
Left side shows Bad News: When x→ y is increased to 60 again. **Right side shows Good News**: When x→y is reduced to 4





Router z: **Left side: Initial iteration for each router.** **Right side: the output after it stabilizes**





Left side shows Bad News: When x→ y is increased to 60 again. **Right side shows Good News**: When x→y is reduced to 4