A Project on
Distance Vector Routing Protocol

Team Members:
MOHAMMAD SAIF
SAI HARIKA PALURI
HERLEEN KAUR SANHOTRA

Professor:
Dr. Dewan T. Ahmed
TA:
Sahithi Priya Gutta

ABSTRACT

Distance Vector Routing or DVR for short is a routing algorithm used to find the shortest/best route for data packets to traverse based on distance. There are a lot of factors involved in DVR such as cost, latency, and availability of routers but in this project, we will be focusing on cost and availability.
DVR protocol makes use of the Bellman-Ford algorithm to count the shortest path, it was originally used in ARPANET(the foundation of the internet) and was adapted to use in Local Area Networks later on.

TABLES OF CONTENT

1. Introduction

DVR works on the Bellman Ford algorithm developed in the 50s as an alternative to Dijkstra's algorithm (used in Link State routing) the algorithm is shown below:

```
function BellmanFord(list vertices, list edges, vertex source)
  ::distance[],predecessor[]

  // This implementation takes in a graph, represented as
  // lists of vertices and edges and fills two arrays
  // (distance and predecessor) about the shortest path
  // from the source to each vertex

  // Step 1: initialize graph
  for each vertex v in vertices:
      distance[v] := inf             // Initialize the distance to all vertices to infinity
      predecessor[v] := null         // And having a null predecessor

  distance[source] := 0              // The distance from the source to itself is, of course, zero

  // Step 2: relax edges repeatedly
  for i from 1 to size(vertices)-1:
      for each edge (u, v) with weight w in edges:
          if distance[u] + w < distance[v]:
              distance[v] := distance[u] + w
              predecessor[v] := u

  // Step 3: check for negative-weight cycles
  for each edge (u, v) with weight w in edges:
      if distance[u] + w < distance[v]:
          error "Graph contains a negative-weight cycle"

  return distance[], predecessor[]
```

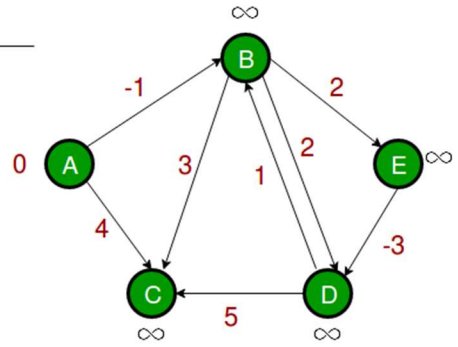
The algorithm runs in O(|V|.|E|) time, where |V| and |E| are the vertices and edges of the graph respectively.

2. Example

Let us understand the algorithm with following example graph.

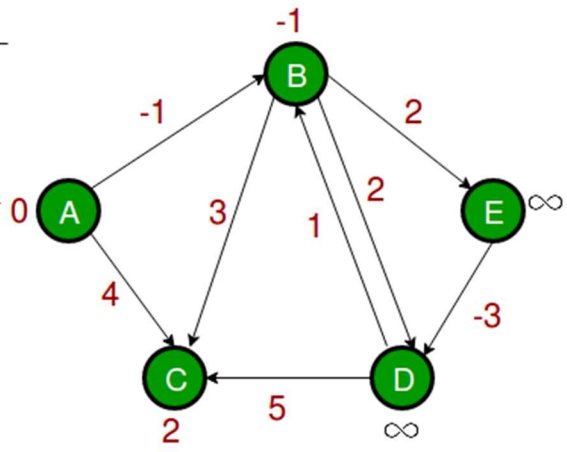
Let the given source vertex be 0. Initialize all distances as infinite, except the distance to source itself. Total number of vertices in the graph is 5, so *all edges must be processed 4 times.*

A	B	C	D	E
0	∞	∞	∞	∞



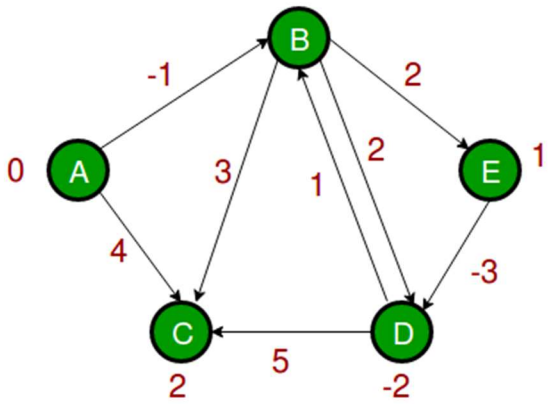
Let all edges are processed in the following order: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D). We get following distances when all edges are processed first time. The first row in shows initial distances. The second row shows distances when edges (B, E), (D, B), (B, D) and (A, B) are processed. The third row shows distances when (A, C) is processed. The fourth row shows when (D, C), (B, C) and (E, D) are processed.

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞



The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get following distances when all edges are processed second time (The last row shows final values).

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1



The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so the third and fourth iterations don't update the distances.

3. Modules

This program was done in Java 1.8

There are four classes in the program, namely - RTblEntry, RTable, RtStart and RtSend.

RtSend
Running this sends out data every 15 seconds, it will check for any cost changes and send RTable object of the node to the multicast group and prints what it sends.

RtStart
This file contains the main method for the program, it reads the binary data files and constructs a routing table. Then it calls the constructor of RtSend to start the transmission process. A Multicast Socket is initialised which is then added to a multicast group with IP address "x.x.x.x".
Therefore, any UDP datagrams this address receives are also received by the MulticastSocket, it's bound with the port number passed at the command line.
Whenever a node is run, it initialises a new socket and they all have the same port number and are added to the same multicast group, therefore all nodes receive the packets sent out by all the nodes.

Upon the arrival of a new datagram, the method checks whether the router that the object came from is an immediate neighbor, if so it updates the routing table, otherwise it doesn't.

RTable

RTable is a serializable class, each and every single object is one router.
The methods-

- getRouterName and setRouterName for getting the router variables.
- firstRouteTable uses the input from the binary data file passed and fills up the variable table with immediate neighbor entries, variable's immediate neighbors and originals are also updated.
- updRtrTbl accepts two inputs of RTable objects mytable and rcvdtable which are the routers object, it first checks for any new neighboring entries in received and adds them to the routers own object, the cost of said neighbors is set to infinite and the nextHop is set to '-'. Both tables and immNodes are updated.
  After this, it copies the RTable object into a new object, the distance between the current router and the router from which the object was received from is added to the cost values in the new table. The nextHop entries for all the rows are changed to the name of the object from which it was copied from.
  The 2-update problem is rectified by changing the cost of entries which had nextHop value the same as the name of the receiving router to a big finite number.
  The modified table and routers own table are compared and updated according to the algorithm.
- displayTable takes the input of ArrayList var and prints it out.
- chckLinkcst parses the file repeatedly to check if there are any link cost changes, and if so then it makes the necessary changes. It reads the data files, creates a new RTable object and adds the values from it, it then compares it with the original cost values to check if there are any cost changes, which if there are there then new costs are updated in the original table and the nextHop table variable is also changed accordingly.

RTblEntry

There are only getter and setter's for the variables and a constructor to initialize every single row in the routing table.

4. How to run the program

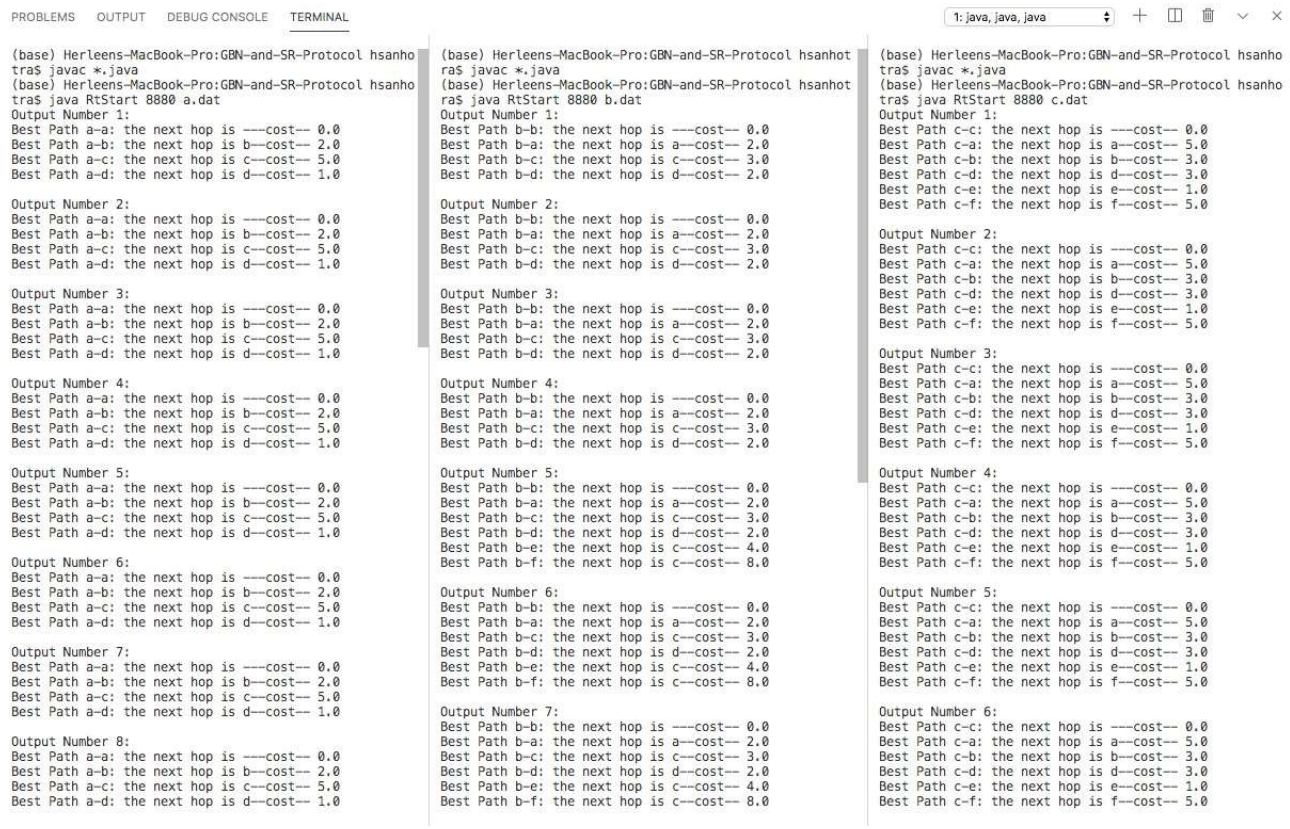
- Open command prompt and make sure the java version is 1.8 or higher.
- Compile all the programs by running javac *.java command.
- Go one level above the current directory and run

java RtStart <port-number> <file-path>

- Repeat the same for all nodes, and use the same port numbers

5. Screenshots

Screenshot for one case where we run a,b,c,d,e and f nodes.



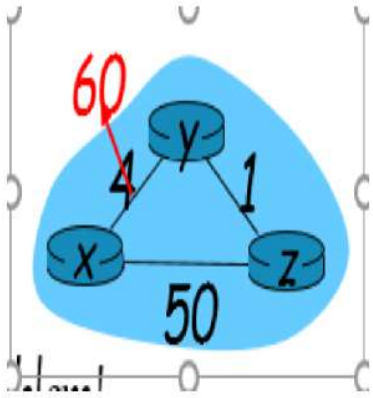
PUT DEBUG CONSOLE TERMINAL 1: java, java, jav

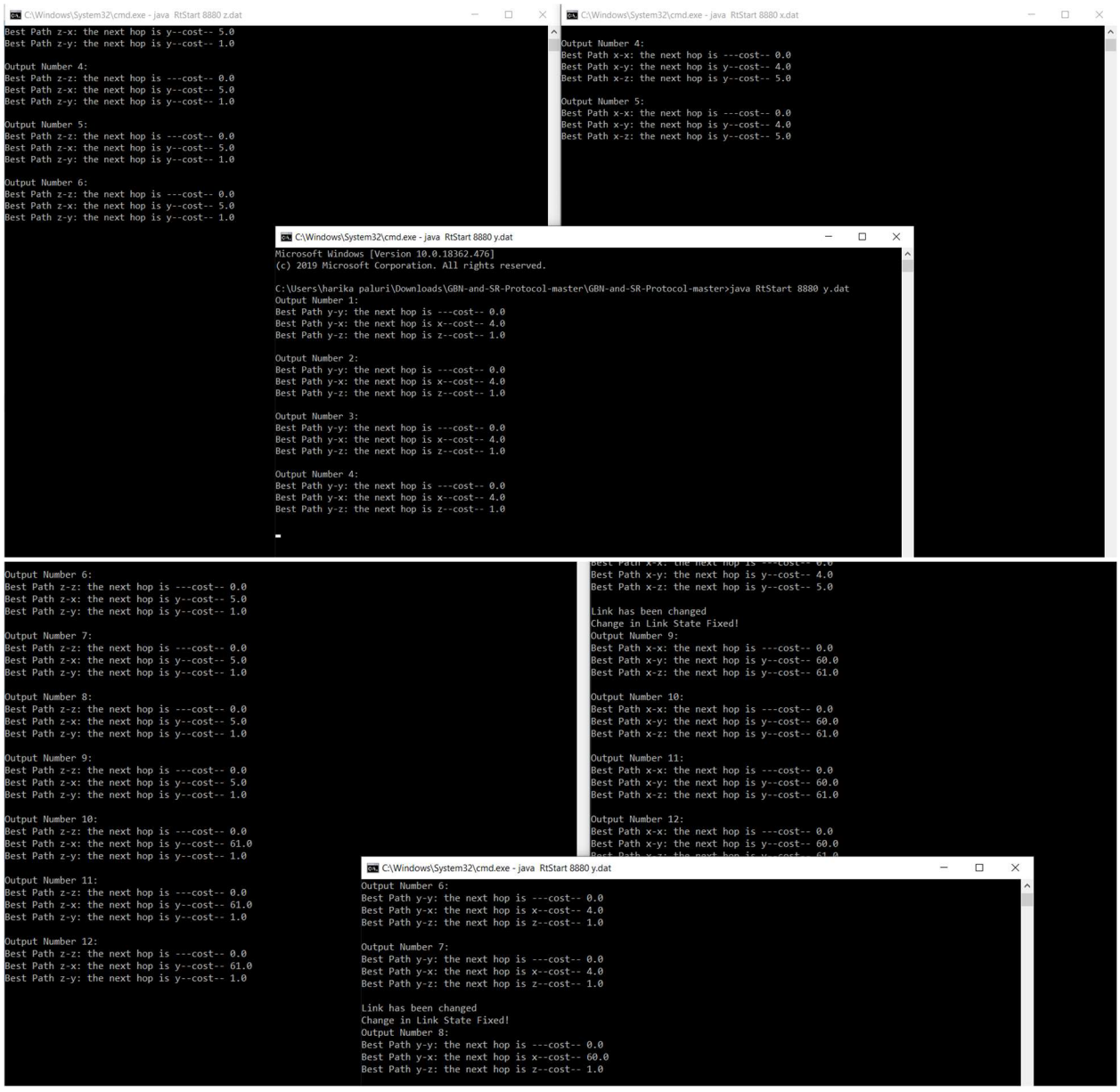
```
(base) Herleens-MacBook-Pro:GBN-and-SR
-Protocol hsanhotra$ javac *.java

       (base) Herleens-MacBook-Pro:GBN
-and-SR-Protocol hsanhotra$ java RtSta
rt 8880 d.dat
Output Number 1:
Best Path d-d: the next hop is ---cost
-- 0.0
Best Path d-a: the next hop is a--cost
-- 1.0
Best Path d-b: the next hop is b--cost
-- 2.0
Best Path d-c: the next hop is c--cost
-- 3.0
Best Path d-e: the next hop is e--cost
-- 1.0
Best Path d-f: the next hop is c--cost
-- 8.0

Output Number 2:
Best Path d-d: the next hop is ---cost
-- 0.0
Best Path d-a: the next hop is a--cost
-- 1.0
Best Path d-b: the next hop is b--cost
-- 2.0
Best Path d-c: the next hop is c--cost
-- 3.0
Best Path d-e: the next hop is e--cost
-- 1.0
Best Path d-f: the next hop is c--cost
-- 8.0


Output Number 3:
Best Path d-d: the next hop is ---cost
-- 0.0
Best Path d-a: the next hop is a--cost
-- 1.0
Best Path d-b: the next hop is b--cost
-- 2.0
Best Path d-c: the next hop is c--cost
-- 3.0
Best Path d-e: the next hop is e--cost
-- 1.0
Best Path d-f: the next hop is c--cost
-- 8.0

Output Number 4:
Best Path d-d: the next hop is ---cost
-- 0.0
```

```
(base) Herleens-MacBook-Pro:GBN-an
d-SR-Protocol hsanhotra$ javac *.j
ava
(base) Herleens-MacBook-Pro:GBN-an
d-SR-Protocol hsanhotra$ java RtSt
art 8880 e.dat
Output Number 1:
Best Path e-e: the next hop is ---
cost-- 0.0
Best Path e-c: the next hop is c--
cost-- 1.0
Best Path e-d: the next hop is d--
cost-- 1.0
Best Path e-f: the next hop is f--
cost-- 2.0
Best Path e-a: the next hop is d--
cost-- 2.0
Best Path e-b: the next hop is d--
cost-- 3.0

Output Number 2:
Best Path e-e: the next hop is ---
cost-- 0.0
Best Path e-c: the next hop is c--
cost-- 1.0
Best Path e-d: the next hop is d--
cost-- 1.0
Best Path e-f: the next hop is f--
cost-- 2.0
Best Path e-a: the next hop is d--
cost-- 2.0
Best Path e-b: the next hop is d--
cost-- 3.0

Output Number 3:
Best Path e-e: the next hop is ---
cost-- 0.0
Best Path e-c: the next hop is c--
cost-- 1.0
Best Path e-d: the next hop is d--
cost-- 1.0
Best Path e-f: the next hop is f--
cost-- 2.0
Best Path e-a: the next hop is d--
cost-- 2.0
Best Path e-b: the next hop is d--
cost-- 3.0

Output Number 4:
Best Path e-e: the next hop is ---
cost-- 0.0
Best Path e-c: the next hop is c--
```

```
(base) Herleens-MacBook-Pro:GBN-and-SR-Prot
ocol hsanhotra$ javac *.java
(base) Herleens-MacBook-Pro:GBN-and-SR-Prot
ocol hsanhotra$ java RtStart 8880 f.dat
Output Number 1:
Best Path f-f: the next hop is ---cost-- 0.
0
Best Path f-c: the next hop is e--cost-- 3.
0
Best Path f-e: the next hop is e--cost-- 2.
0
Best Path f-d: the next hop is e--cost-- 3.
0
Best Path f-a: the next hop is e--cost-- 4.
0
Best Path f-b: the next hop is e--cost-- 5.
0

Output Number 2:
Best Path f-f: the next hop is ---cost-- 0.
0
Best Path f-c: the next hop is e--cost-- 3.
0
Best Path f-e: the next hop is e--cost-- 2.
0
Best Path f-d: the next hop is e--cost-- 3.
0
Best Path f-a: the next hop is e--cost-- 4.
0
Best Path f-b: the next hop is e--cost-- 5.
0

Output Number 3:
Best Path f-f: the next hop is ---cost-- 0.
0
Best Path f-c: the next hop is e--cost-- 3.
0
Best Path f-e: the next hop is e--cost-- 2.
0
Best Path f-d: the next hop is e--cost-- 3.
0
Best Path f-a: the next hop is e--cost-- 4.
0
Best Path f-b: the next hop is e--cost-- 5.
0

Output Number 4:
Best Path f-f: the next hop is ---cost-- 0.
0
Best Path f-c: the next hop is e--cost-- 3.
0
Best Path f-e: the next hop is e--cost-- 2.
```

6. Screenshots for recursive update scenario





7. References

1. https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/
2. https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm
3. https://en.wikipedia.org/wiki/Distance-vector_routing_protocol