Dijkstra's Algorithm for a Dynamic Topology

Harshit R. Paleja

***Abstract*— The project proposes a Dijkstra's algorithm method to construct a table of shortest paths between each and every node in a network. Later on, we utilize the table in order to save time whenever changes are being made to the network. The time saved can be crucial in many real-time systems. A backup of the tables is stored in a MySQL database just in case the Java program is corrupted.**

***Index Terms*— Dijkstra's, dynamic, network, protocol**

I. INTRODUCTION

ROUTING is a technique by which the network comes to know about the paths to be followed by its packets, depending on the topology of the network. One of the key issues in network analysis is finding an easy and computationally simple way to compute the shortest distance between two nodes. Once this distance has been found, another problem is to make changes in the path values once the network has started functioning. We have to find a way such that the entire algorithm does not need to be run in case there are any updates in the network. That can get a bit cumbersome in a network with a large number of nodes. Hence MySQL tables are called used the next time any changes are to be made. The tables allow any user in any part of the world to access the data by a simple connection to the MySQL server. This allows for an increased scope for the project.

A good routing algorithm has the following characteristics :

1. Speedy delivery of packets : The algorithm should ensure there is minimum delay in calculating the routing table, and the errors are kept to a minimum.
2. Flexibility : As the network keeps changing continuously due to link failures, addition and removal of nodes, the algorithm should be flexible enough to adapt to these changes.
3. Link-cost change adaptability : The link costs might also change over time, due to varying traffic loads. The algorithm should take this into account
4. Avoiding loops : Once the routing has started, care should be taken that the packets are not delivered back to the sender due to bugs in the code, or the packet should not be stuck in a loop.

II. ROUTING TABLES AND ALGORITHMS

Routing tables are used to store the data. Figure 1a and 1b show the sample tables. The 2 tables have been linked using Foreign Keys in MySQL. The second table contains only text data showing the path to be followed.

Node	A	B	C	D	E
A	-	20	30	40	50
B	20	-	60	40	30
C	30	60	-	20	25
D	40	40	20	-	15
E	50	30	25	15	-

Fig 1a : Sample table for nodes and their distances from the other node.

Node	A	B	C	D	E
A	-	A-B	A-B-C	A-B-D	A-C-E
B	B-A	-	B-C	B-D	B-E
C	C-B-A	C-B	-	C-D	C-D-E
D	D-B-A	D-B	D-C	-	D-E
E	E-C-A	E-B	E-D-C	E-D	-

Fig 1 b Sample table for path from each node to all nodes, next node first, stored as a string in MySQL

Manuscript received March 21, 2010. This work was a part of coursework in EEL 5718 – Computer Communication Networks under Dr. Haniph Latchman.

H. R. Paleja is a student at the University of Florida, Gainesville, FL 32601 Phone : 353-665-6066; E-mail : pale177@ufl.edu

The routing protocols are classified into Global Routing Algorithm and Decentralized Routing Algorithm.

A Global Routing Algorithm has the inputs as all the node-link values and the connections. The calculations are either done at one site using a centralized algorithm, or replicated at multiple sites. They are also termed as Link-State Algorithms.

In Decentralized Routing Algorithms, none of the nodes stores information of the costs of the links that it is not connected to i.e. the nodes are concerned only with their neighbors, and hence memory is saved. They are also called Distance Vector Algorithms. These algorithms suffer a problem of getting into loops.

The techniques used for routing can also be classified based on their responsiveness as :

1. Static : Here the paths taken to deliver a packet are pre-computed and fixed. The paths are calculated and passed on to each and every node. In case of a link failure, transmission stops until the link is repaired. Also, when the link costs change, all the pre-computed paths will not be optimal anymore, leading to a reduced efficiency in routing time.

2. Dynamic : Here, each node knows what's going on in its neighborhood, and hence is informed about the entire network via its neighbors. These are useful when the network keeps changing continuously. The node itself can calculate the distances to other nodes.

Shortest distance routing protocols are of the following types

1. Bellman Ford's algorithm : This algorithm is preferred when the link values are negative. The algorithm is based on one simple rule. If we find that C is on the shorted path from A to B, then the same path from the node A to node C and the path from node B to node C is also the shortest path. The disadvantage of this algorithm is that it reacts slowly to link failure. For networks with non-negative edges, the faster Dijkstra's algorithm is used.

2.

III. DIJKSTRA'S ALGORITHM

One of the algorithms that I thought of was calculating the cost along each and every path from node A to node B, and then selecting the shortest path amongst them. That would be the easiest of algorithms to code, but very time consuming as it involves unnecessary calculations and discarding them later. A simple Dijkstra's algorithm allows us to eliminate the need for these calculations, by eliminating the nodes before they are involved in calculations.

The algorithm consists of the following steps:
1. The distances in the table are initially set to infinity; in this project 1000 is considered as the initial distance.
2. One of the nodes is selected as the initial node and its neighbors are brought into the candidate set.
3. The next node will be the one at shortest distance from the first node, and again its neighbors are moved to the candidate set. The distances are noted down.
4. Depending on the distance from the initial node, the next node will be the one at the second largest distance from the initial node will be the next "initial node".
5. The nodes that have been already visited will be marked as visited, and the result array will be populated with their distances from the initial node.
6. The above steps are repeated until all the nodes have been visited, which gives us the minimum distance from the initial node to all other nodes.

When applied to all the nodes in a complete network, we can easily compute the routing table, which is stored in a Java array.

IV. CLASSES AND FUNCTIONS USED

The Java code has been split up into a number of functions and classes. They are described as follows:

1. Main.java

This function is used to run the program. It includes calls to many of the functions described below, and is basically a start function wherein initially, the nodes are assigned values and neighbors and then the control is passed to the other functions. The return type is void, and it takes in an input parameter via the command line to start the algorithm, called the starting node. It is required in any Java program, and the input arguments always point to this function.

2. Fillvalue.java

This function is used to assign neighbors and their corresponding distances. As of now, the input values are specified in the main function itself, and they are passed as parameters to this function. It returns a value of the type nodes. The function makes it easier to introduce new nodes by just entering the values, rather than typing each field every time a new node variable is declared.

3. Neigh_length.java

This function is used to calculate the number of neighbors. This was required as the user does not need to count the number of neighbors; he just needs to specify the links and the link values to the fillvalue function. The return type is an

integer. The maximum value of this integer will be 4, as I have limited the number of neighbors for this simulation.

4. Nodes.java

This class, as the name suggests, defines the objects called nodes. These represent the actual nodes in a network, and more parameters can be added to this class, whenever additional functions are to be added. Each variable of the class nodes has 4 elements – name (integer), neighbors (integer array), distance from neighbors (integer array) and flag (integer). The flag field has to be set whenever we want to mark the node as visited, so that we do not enter a loop while scanning the nodes.

5. Route.java

This class defines is used to find the next neighbor based on the shortest distance from the initial node. The input parameters are all the nodes, along with a node specified as the initial node by the user. It calls the function move_further.java. The initial node is passed as the second parameter always and the other nodes are presented in the order of their names. The first node is final array of results, which is a 2-D array that contains the node numbers as row and column headings, and the distance from other nodes as the table values. Finally, the function move_further () returns a table of values, which is again updated and displayed by this function.

6. Move_further.java

As the name suggests, this function is used to move beyond the first node, and is recursively called until all the nodes have been visited. The input parameters for this class include the initial node, and the next node. For the second iteration, the initial node's flag value is set to 1, marking it as visited. The next_node becomes the new initial node and the node with the second largest distance from the first node is marked as the new next_node.

7. Connect.java

This class is used to make a connection to a local MySQL instance. The parameters needed are the hostname (localhost), username (root) and the password for the MySQL server. The connection objects will be required in all the classes that write/read from the database.

V. The Network Used

The following network in Figure 2 was used during the project. As of now, the node limit has been kept to 8 nodes. Expansion can be done using HashMaps. Proper care was taken in designing this network, such as the indirect path 3 – 2 – 4 being of a lesser value than the direct path from 3 – 4. The algorithm has to be modified in such cases, and the proposed algorithm takes care of it.

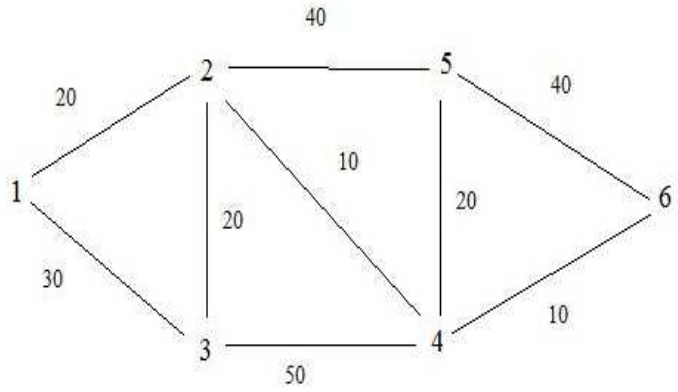


Fig. 2. The network used for the project. 6 nodes and their distances from neighbors. Can support up to 8 nodes.

VI. Issues Faced

The following issues were faced during the initial stages of the project.

1. According to the network design, there is a direct path from node 1 to node 3(cost - 30), and also there is an indirect path from 1 – 3 via node 2 (cost - 40). According to the algorithm, the next_node after node 1 was node 2, and node 2 has 3 as its neighbor. Since node 3 was not marked as visited, the value 30 was bring replaced by 40, which is obviously a bug. So a condition was introduced, wherein the stored value will be replaced only if the new value is lesser than the previous value.

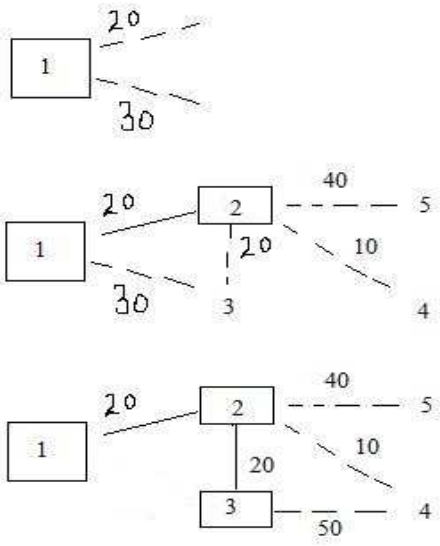


Fig. 3. The flow when node 1 is the initial node, and distance from node 1 to node 3 is wrongly replaced in step 2. Dotted lines indicate candidate paths, straight lines indicate completed paths, boxes indicate the node has been visited.

Conversely, the least distance from 3 – 4 in Fig.3 is observed to be 30 via node 2, but the direct distance from node 3 to node 4 was 40. That has to be taken into consideration. So the mentioned condition takes care of that too.

2. According to Dijkstra's algorithm, the next_node was at the least distance from the initial node. The control was shifted to that node, but the second node also had the first node as its neighbor. Hence, what I got was a distance from the initial node to the initial node itself, which was twice the distance between the first node and the initial node. What happened subsequently was the program flow entered an infinite loop between the initial node and the next_node. That's when I introduced a flag bit for marking the visited nodes, so that they are not visited again. That explains the need for a flag bit. Figure 3 shows the output without a flag bit.

3. The connection to MySQL required a JDBCdriver, which had to be downloaded from the MySQL website. Neither Netbeans nor Eclipse has an inbuilt JDBC driver. That has to be downloaded and inserted into the proper MySQL directory. It can be downloaded from the following link http://dev.mysql.com/downloads/connector/j/3.1.html The current-most version is 3.2.0. It's only 8.4MB compared to the 3.1.14 used in the project, which is 27.6MB.

4. All the nodes were being passed as parameters from the functions, which can get a bit tedious when larger networks are involved. So Hashmaps were used to reduce the complexity of the program.

5. Initially, the tables in MySQL had to be designed from the command line. MySQLAdmin was downloaded as a faster and easier option to build tables in MySQL. It is shown in Figure 4

6. MySQL tables had to be divided into 2, one for holding link values, and one for holding the link paths. This can be avoided by the use of multi dimensional datasets.

VII. Results

Assuming initial node as node 2, the sequence of events that were simulated are shown in figure 4. The figure 6 shows screenshots of the output step by step

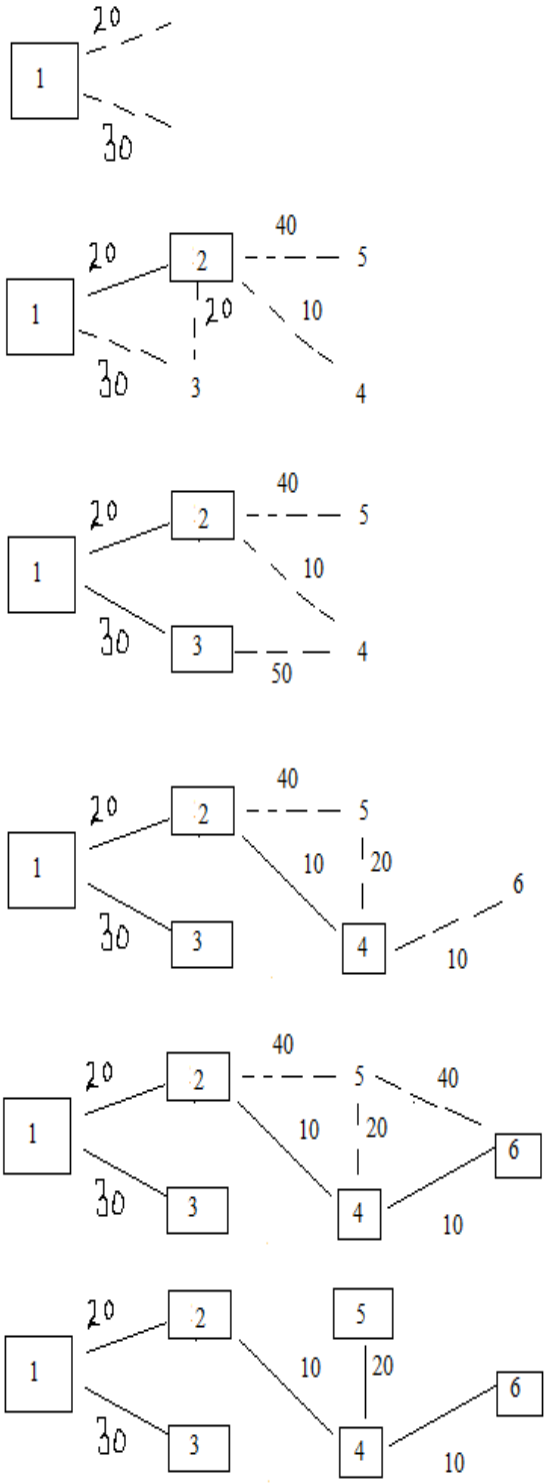


Fig. 4. The flow when node 1 is the initial node. Dotted lines indicate the candidate paths, straight lines indicate shortest path from node 1 to that node. The nodes numbers inside a box indicate that the node has been visited.

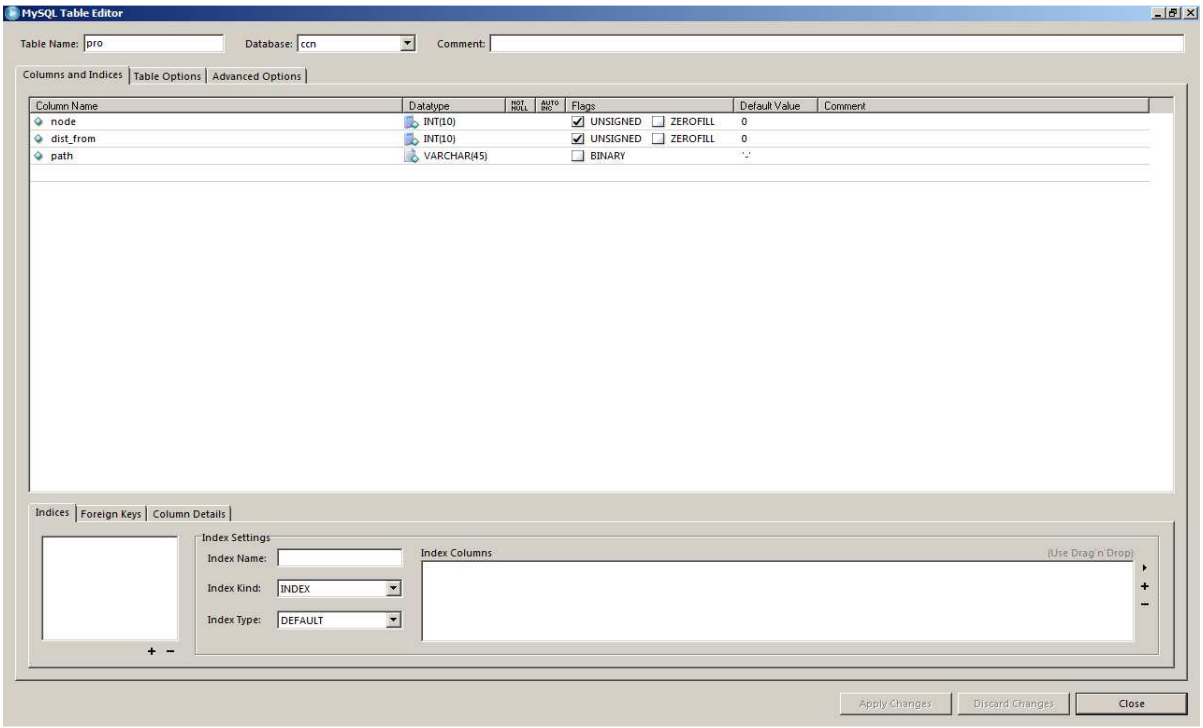


Fig. 5. MySQLAdmin – The GUI for MySQL

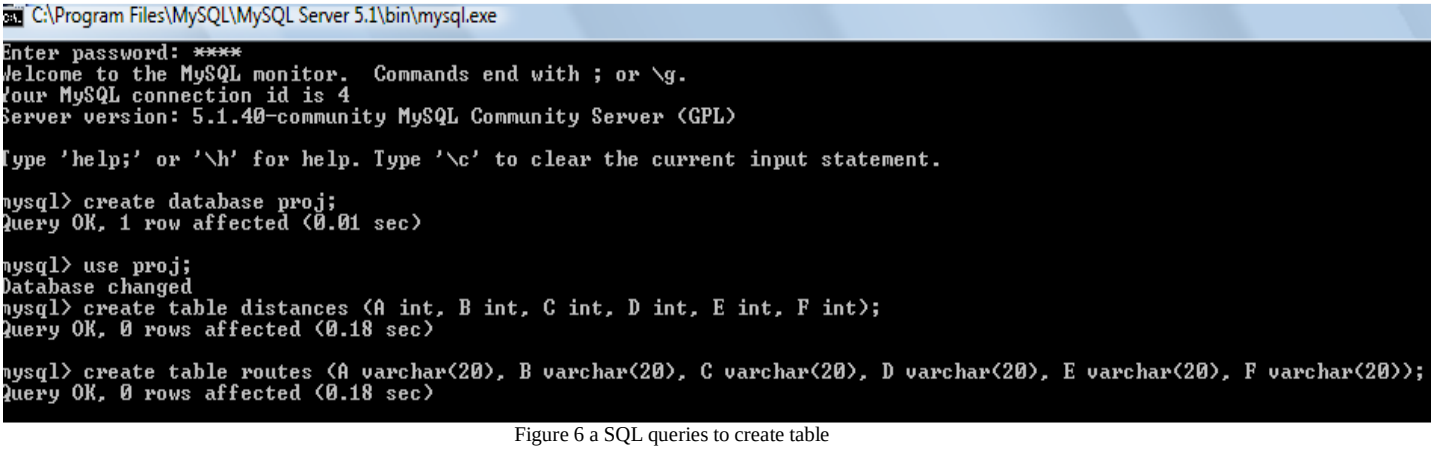


Figure 6 a SQL queries to create table

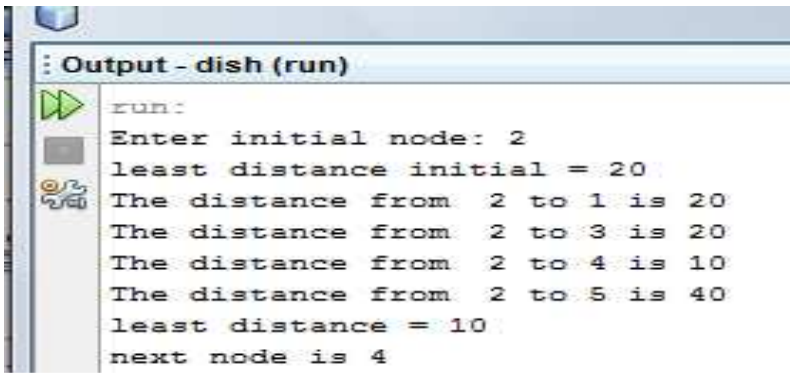


Fig. 6b. Result : Step 1

```
1000 1000 1000 1000 1000 1000 1000 1000
1000 1000 1000 1000 1000 1000 1000 1000
1000 1000 1000 1000 1000 1000 1000 1000
1000 1000 1000 1000 1000 1000 1000 1000
1000 1000 1000 1000 1000 1000 1000 1000
1000 1000 1000 1000 1000 1000 1000 1000
1000 1000 1000 1000 1000 1000 1000 1000
1000 1000 1000 1000 1000 1000 1000 1000
```

Figure 6c. Result : The initial routing table array from Java

```
ysql> select * from distance;
+------+------+------+------+------+------+------+
| node | A    | B    | C    | D    | E    | F    |
+------+------+------+------+------+------+------+
| A    | 1000 | 1000 | 1000 | 1000 | 1000 | 1000 |
| B    | 1000 | 1000 | 1000 | 1000 | 1000 | 1000 |
| C    | 1000 | 1000 | 1000 | 1000 | 1000 | 1000 |
| D    | 1000 | 1000 | 1000 | 1000 | 1000 | 1000 |
| E    | 1000 | 1000 | 1000 | 1000 | 1000 | 1000 |
| F    | 1000 | 1000 | 1000 | 1000 | 1000 | 1000 |
+------+------+------+------+------+------+------+
 rows in set (0.00 sec)
```

Figure 6d Initial routing table in SQL

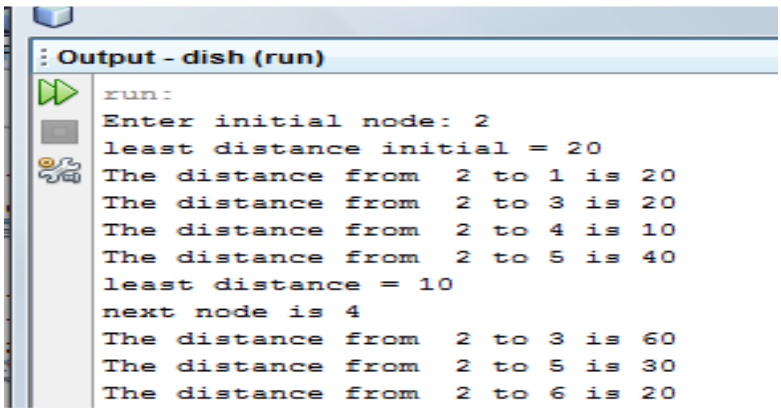


Figure 6e : Result : Step 2 (Program goes to move_further.java)

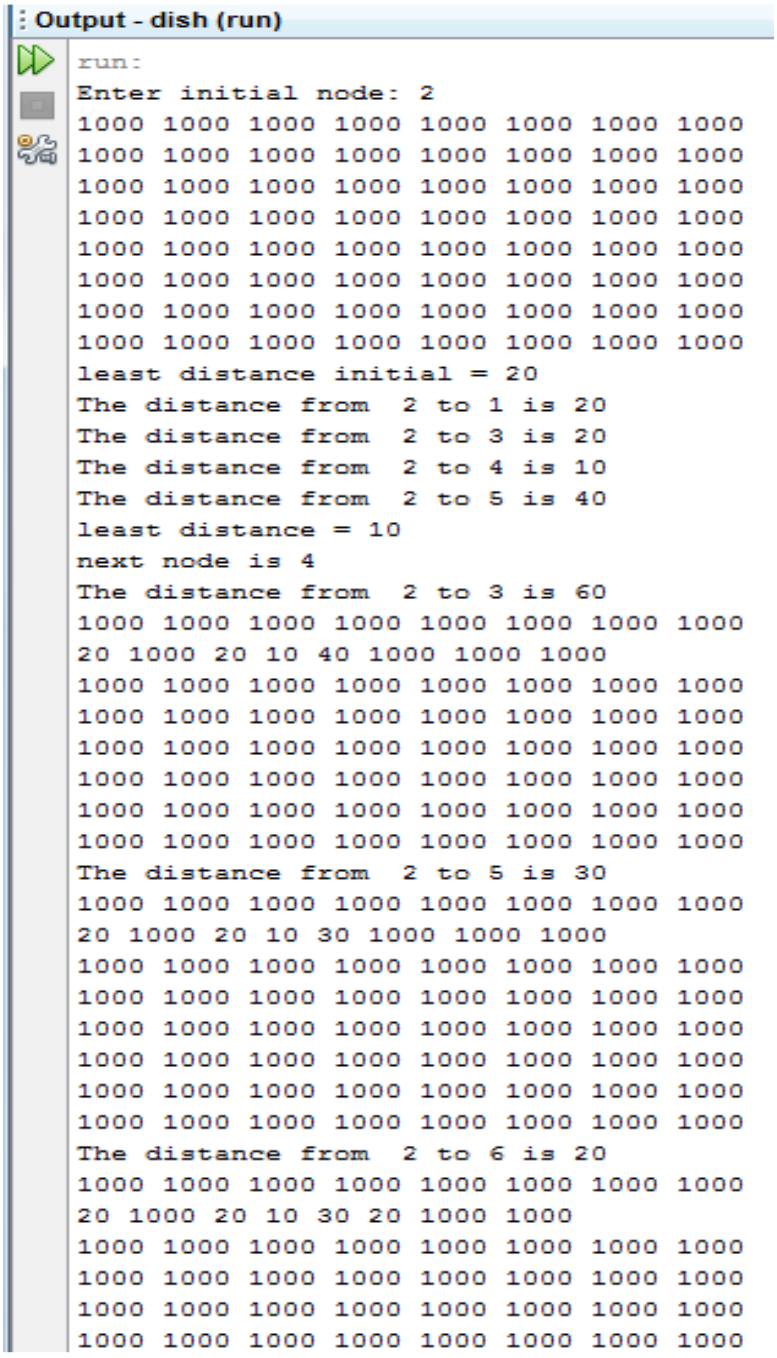
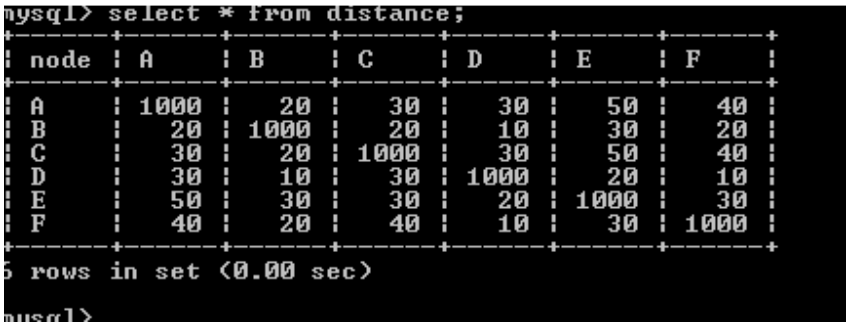


Figure 6f Entire algorithm in Java



```
mysql> select * from distance;
+------+------+------+------+------+------+------+
| node | A    | B    | C    | D    | E    | F    |
+------+------+------+------+------+------+------+
| A    | 1000 |   20 |   30 |   30 |   50 |   40 |
| B    |   20 | 1000 |   20 |   10 |   30 |   20 |
| C    |   30 |   20 | 1000 |   30 |   50 |   40 |
| D    |   30 |   10 |   30 | 1000 |   20 |   10 |
| E    |   50 |   30 |   30 |   20 | 1000 |   30 |
| F    |   40 |   20 |   40 |   10 |   30 | 1000 |
+------+------+------+------+------+------+------+
6 rows in set (0.00 sec)

mysql>
```

Figure 6g Final result of all nodes calculates and stored in MySQL

VIII. APPENDIX – SOURCE CODE

Code for the entire project is as follows

```
/*
 Main.Java
 */
package dish;
import java.awt.*;
import java.lang.*;
import java.util.*;
import java.math.*;
import java.io.*;
import java.util.Scanner;

/**
 *
 * @author User
 */
public class Main
{
   public static void main(String[] args)
   {
      // command line for length of
      nodes A = new nodes();
      nodes B = new nodes();
      nodes C = new nodes();
      nodes D = new nodes();
      nodes E = new nodes();
      nodes F = new nodes();
      nodes G = new nodes();
      nodes H = new nodes();
      nodes I = new nodes();
```

```
        nodes[] node_array = new nodes[8];

        node_array[0] = A;
        node_array[1] = B;
        node_array[2] = C;
        node_array[3] = D;
        node_array[4] = E;
        node_array[5] = F;
        node_array[6] = G;
        node_array[7] = H;
 //       node_array[0] = A;


        //node_array[] = [A;B;C;D;E;F;G;H];

int[] candidate = new int[5];

//       A.ne[0] = B;
//       A.ne[1] = C;

        A = fillvalue.fill(A, 1, 2, 20, 3, 30, 0, 0, 0, 0);
        B = fillvalue.fill(B, 2, 1, 20, 3, 20, 4, 10, 5, 40);
        C = fillvalue.fill(C, 3, 1, 30, 2, 20, 4, 50, 0, 0);
        D = fillvalue.fill(D, 4, 2, 10, 3, 50, 5, 20, 6, 10);
        E = fillvalue.fill(E, 5, 2, 40, 4, 20, 6, 40, 0, 0);
        F = fillvalue.fill(F, 6, 4, 10, 5, 40, 0, 0, 0, 0);
        G = fillvalue.fill(G, 7, 0, 0, 0, 0, 0, 0, 0, 0);
        H = fillvalue.fill(H, 8, 0, 0, 0, 0, 0, 0, 0, 0);



        Scanner user_input = new Scanner( System.in );
        int init_node;
        System.out.print("Enter initial node: ");
        init_node = Integer.parseInt(user_input.next());
//           System.out.println("Echo: " + init_node);


            switch(init_node){

                case 1 : { int[] crap = new int[4];
                crap = route.dist_between(A,A,B,C,D,E,F,G,H); break;
                }
                case 2 : { int[] crap = new int[4];
                crap = route.dist_between(B,A,B,C,D,E,F,G,H); break;
                }
                case 3 : { int[] crap = new int[4];
                crap = route.dist_between(C,A,B,C,D,E,F,G,H); break;
                }
```

```
            case 4 : { int[] crap = new int[4];
            crap = route.dist_between(D,A,B,C,D,E,F,G,H); break;
            }
            case 5 : { int[] crap = new int[4];
            crap = route.dist_between(E,A,B,C,D,E,F,G,H);break;
            }
            case 6 : { int[] crap = new int[4];
            crap = route.dist_between(F,A,B,C,D,E,F,G,H);break;
            }
            case 7 : { int[] crap = new int[4];
            crap = route.dist_between(G,A,B,C,D,E,F,G,H);break;
            }
            case 8 : { int[] crap = new int[4];
            crap = route.dist_between(H,A,B,C,D,E,F,G,H);break;
            }
            default : System.out.println("Enter another value "); break;
            }

  }
}



/*
 * Move_further.Java
 */

package dish;

/**
 *
 * @author User
 */
public class move_further {


 static int nextnode (int table[][],nodes n1,nodes k1,nodes k11,nodes k2, nodes k3,nodes k4, nodes k5, nodes k6, nodes k7,nodes
k8,int init_min)
  {
    System.out.println("next node is " + n1.name);


      int p1 = n1.name;
      int p = k1.name;
      int q = k2.name;
      int r = k3.name;
      int s = k4.name;
      int t = k5.name;
      int u = k6.name;
      int v = k7.name;
```

```
      int w = k8.name;

int neigh_2 = 0;
neigh_2 = neigh_length.neigh_length(n1);

int min = 0;

//if m1.neigh
int initial = n1.n_dist[0];
int position  = 0;

//  System.out.println("least distance initial = " + initial);
        if((n1.neighbour[0] == k1.neighbour[0]) || (n1.neighbour[0] == k1.neighbour[1]) || (n1.neighbour[0] == k1.neighbour[2]) ||
(n1.neighbour[0] == k1.neighbour[3]) || (n1.neighbour[0] != k1.name))
        {System.out.println("The initial distance from  " + k1.name + " to " +n1.neighbour[0] + " is " + (init_min+n1.n_dist[0]));
        if ((table[k1.name-1][n1.neighbour[0]-1] > init_min+n1.n_dist[0]))
         table[k1.name-1][n1.neighbour[0]-1] = init_min+n1.n_dist[0];

        }
// Loop to find least element begins here
       for (int i = 1; i <= neigh_2-1 ; i++)
       {
          if(n1.neighbour[i] == k1.name)
        {i++;}
          if (i > neigh_2-1){break;}
    //    if((n1.neighbour[i] == k1.neighbour[0]) || (n1.neighbour[i] == k1.neighbour[1]) || (n1.neighbour[i] == k1.neighbour[2]) ||
(n1.neighbour[i] == k1.neighbour[3]))
    //    {i++;}
          if (i > neigh_2-1){break;}
          if(n1.neighbour[i] == k1.name)
        {i++;}
          if (i > neigh_2-1){break;}

    //      if((n1.neighbour[i] == k1.neighbour[0]) || (n1.neighbour[i] == k1.neighbour[1]) || (n1.neighbour[i] == k1.neighbour[2])
|| (n1.neighbour[i] == k1.neighbour[3]))
    //    {i++;}
          if (i > neigh_2-1){break;}

          //           System.out.println(" neigh1 = " + n1.neighbour[i] + "    k1.name = "+ k1.name);

if (i <= neigh_2-1)
{ //    System.out.println("element = " + n1.n_dist[i]);
          int something = init_min+n1.n_dist[i];
          System.out.println("The distance from  " + k1.name + " to " +n1.neighbour[i] + " is " + something);
          if ((table[k1.name-1][n1.neighbour[i]-1] > something))
          table[k1.name-1][n1.neighbour[i]-1] = something;
          if (n1.n_dist[i] < initial)
          {
             min = n1.n_dist[i];
// System.out.println("min = " + min);
```

```
            initial = min;
//  System.out.println("least distance = " + min);
            position = i;

          }


 for (int i1 = 0; i1<=7;i1++)
{for (int j1 = 0; j1<=7; j1++)
{System.out.print(table[i1][j1] + " ");}
        System.out.println(" ");
}

}

else break;
       }// end for

 return 0;
  } //end func


}//end class
```

IX. REFERENCES

[1] F. Laurent, V. Aguilera, “An Atomic Dijkstra Algorithm for dynamic shortest paths in traffic assignment” Available at http://www.tzi.de/~edelkamp/publications/workshops/tt/33AK_Workshop.pdf
[2] A. Orda, R. Rom, “Shortest-path and minimum delay algorithms in networks with time-dependent edge length,” Journal of the ACM, 37(3):607-625, 1990.
[3] Leon-Garcia, Widjaja, ”Computer Communication Networks”, 2nd edition, McGraw-Hill Publications, 2003.
[4] http://en.wikipedia.org/wiki/Dijkstra's_algorithm
[5] www.java.sun.com