Nick Jordy

CPE400 Coding Project

Project Topic 1:

Create and simulate a new routing strategy that maximizes the overall throughput of a mesh network. Throughput is affected by many factors that should be considered, such as nodal processing delay, overloaded buffers, loss, etc. The more realistic assumptions you can make for your network, better it is.

Due: 12/5/2018

Table of Contents

Problem Statement

Throughput is a measure of how much data can move through a channel in a network, measured in Bits Per Second (bps). The higher the throughput, the faster data is transferred between two routers. There are many factors that can affect throughput, but the main one this project tackles is bandwidth. Bandwidth is similar to throughput, as it is the maximum amount of data that can be sent through a channel in a network. The solution to maximize the throughput, is to route to the path with the highest bandwidth, allowing the most data to be sent.

Proposed Solution

Taking the path with the highest bandwidth will allow the most amount of data to be sent through. Networks can easily be described as a Graph data structure in any programming language. This project uses an adjacency matrix to represent such a graph structure. The pseudo-code for the algorithm for finding the path with the maximum bandwidth is described in Figure 1.

1. Given any two nodes in a network A and B, use a modified Depth First Search to find all paths from A to B
2. Calculate the bandwidth of each path (pathBandwidth = min(all hop bandwidths))
3. The path taken is the one with the max(all path bandwidths)

Fig 1. The Pseudo-Code for the Algorithm

Solution breakdown

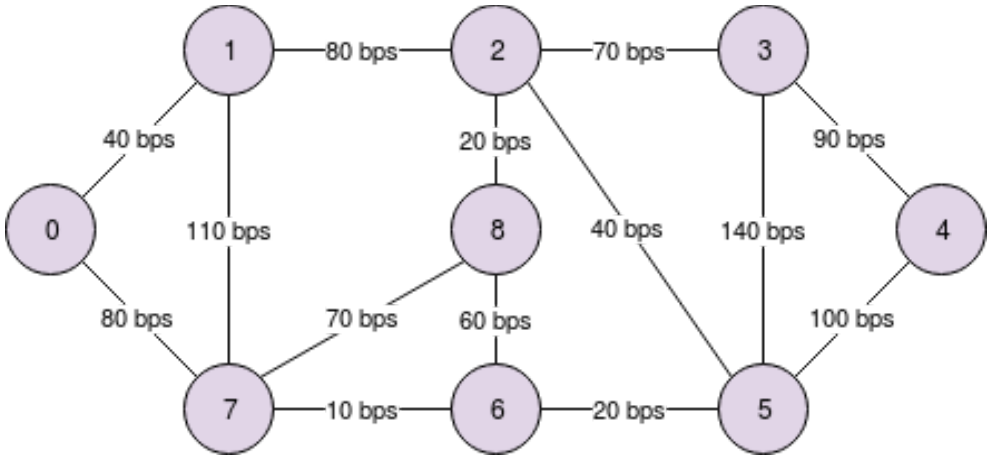

Fig 2. The visual representation of the example network used in the project

The network shown in Figure 2 is described as a set of values with the format:
<Id A><Id B><Bandwidth>
in a configuration file that is read by the main function called “graphData.txt.” Figure 3 shows the contents of the configuration file to assemble this network so that the algorithm can be ran on it.

```
FORMAT:<A><B><COST>
0 1 40
0 7 80

1 0 40
1 7 110
1 2 80

2 1 80
2 8 20
2 5 40
2 3 70

3 2 70
3 5 140
3 4 90

4 3 90
4 5 100

5 3 140
5 2 40
5 4 100
5 6 20

6 8 60
6 5 20
6 7 10

7 0 80
7 1 110
7 8 70
7 6 10

8 2 20
8 6 60
8 7 70
```

Fig 3. The contents of “graphData.txt”

The program reads in the configuration file and builds an adjacency matrix to represent the network in a more "code-friendly" format. Table 1 shows the adjacency matrix for Figure 2, and Figure 4 is a screenshot of the adjacency matrix printed by the program.

Table 1. Adjacency Matrix for the graph

Nodes	0	1	2	3	4	5	6	7	8
0	0	40	0	0	0	0	0	80	0
1	40	0	80	0	0	0	0	110	0
2	0	80	0	70	0	40	0	0	20
3	0	0	70	0	90	140	0	0	0
4	0	0	0	90	0	100	0	0	0
5	0	0	40	140	100	0	20	0	0
6	0	0	0	0	0	20	0	10	60
7	80	110	0	0	0	0	10	0	70
8	0	0	20	0	0	0	60	70	0

```
orion996@orion996-Satellite-C55-A:~/Documents/CS/cpe400/CPE400CodingProject$ ./Program
Enter the Source Router as an Integer Id (eg. '3'): 0
Enter the Destnation Router as an Integer Id (eg. '3'): 3
Graph Adj Table:
0 | 40 | 0 | 0 | 0 | 0 | 0 | 80 | 0 |
40 | 0 | 80 | 0 | 0 | 0 | 0 | 110 | 0 |
0 | 80 | 0 | 70 | 0 | 40 | 0 | 0 | 20 |
0 | 0 | 70 | 0 | 90 | 140 | 0 | 0 | 0 |
0 | 0 | 0 | 90 | 0 | 100 | 0 | 0 | 0 |
0 | 0 | 40 | 140 | 100 | 0 | 20 | 0 | 0 |
0 | 0 | 0 | 0 | 0 | 20 | 0 | 10 | 60 |
80 | 110 | 0 | 0 | 0 | 0 | 10 | 0 | 70 |
0 | 0 | 20 | 0 | 0 | 0 | 60 | 70 | 0 |
```

Fig 4. The Adjacency Matrix printed by the program

Given any two routers with id's A and B, and a Adjacency Matrix graph called G, G[A][B] will contain the bandwidth of the link between routers A and B. Subsequently, this matrix is symmetrical, so G[B][A] will also contain the bandwidth between routers A and B. After the matrix is generated, the algorithm is ran.

The first step of the algorithm is to find all paths from a source router (src) and a destination router (dest). This is done using a modified Depth First Search Algorithm which is shown in Figure 5.

```
void Graph::findAllPaths(int src, int dest, int pathIndex, ofstream& fout)
{
   visited[src] = true;
   path[pathIndex] = src;
   pathIndex++;

   if(src == dest)
   {
      for(int i=0; i<pathIndex ; i++)
      {
         fout << path[i] << " ";
      }
      fout << endl;
   }
   else
   {
      for(int v=0 ; v<numVert ; v++)
      {
         if(!visited[v] && graph[src][v] > 0)
         {
            findAllPaths(v, dest, pathIndex, fout);
         }
      }
   }

   pathIndex--;
   visited[src] = false;

}
```

Fig 5. The Modified Depth First Search

The Depth First Search (DFS) algorithm is used to discover all nodes in a graph given a single source node. In a regular DFS, the algorithm exits its recursive state when all nodes have been discovered. The modified DFS named "findAllPaths()" has a new base case if the id of src is equal to the id of dest. When the condition is met, all nodes stored in the array "path[]" are printed. Each time the function goes back on a recursive step, dest is removed from the list of visited nodes, so that the algorithm will not exit. Instead, it starts the algorithm again using the next undiscovered node.

When dest is found again it prints the array again, which still has the rest of the path saved within it. If no routes can be found by just removing dest from the list of visited nodes, it keeps going

back and removing nodes until a new path can be created. Each time the array is printed, it is saved to a data file so that it can be read and/or viewed by the program later. Figure 6 is the print out of the paths with src=0 and dest=3, generated by the program.

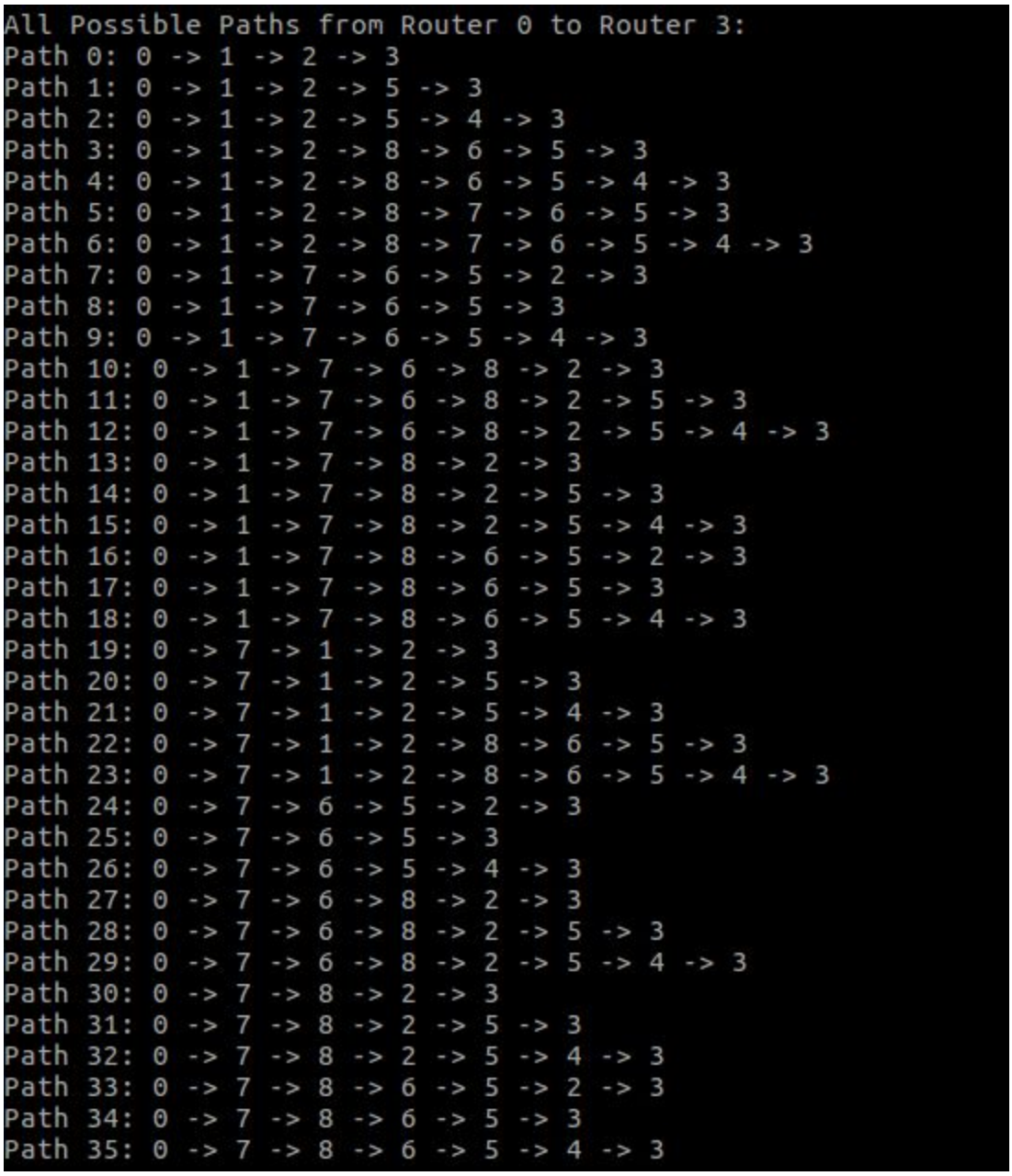

Fig 6. The printout of paths from Router 0 to Router 3

After all the paths between src and dest are found, the bandwidth of each path needs to be found. This is done by the function described in Figure 7.

```
int Graph::findPathBandwidth(int path[], int routersInPath)
{
   int bandwidth = __INT_MAX__;

   for(int i=0 ; i<routersInPath-1 ; i++)
   {
      bandwidth = min(bandwidth, graph[path[i]][path[i+1]]);
   }

   return bandwidth;
}
```

Fig 7. The function to find the bandwidth of a path

To calculate the bandwidth of a path, the minimum of all the bandwidths of that path. For example, take Path 19 shown in Figure 6:

0 - 7 - 1 - 2 - 3

We can use the Adjacency Matrix in table 1 to find the bandwidth of each hop in the path. If the matrix is called G, the bandwidths of path 19 will be stored at G[0][7], G[7][1], G[1][2], and G[2][3]. Referencing table 1, these values are:
G[0][7] = 80 bps
G[7][1] = 110 bps
G[1][2] = 80 bps
G[2][3] = 70 bps
The bandwidth of path 19, B19, is equal to the minimum of these numbers. Thus, we get:
B19 = min(80, 110, 80, 70) bps = 70 bps.

The bandwidths printed for each path by the program from src = 0 and dest = 0 are shown in Figure 8.

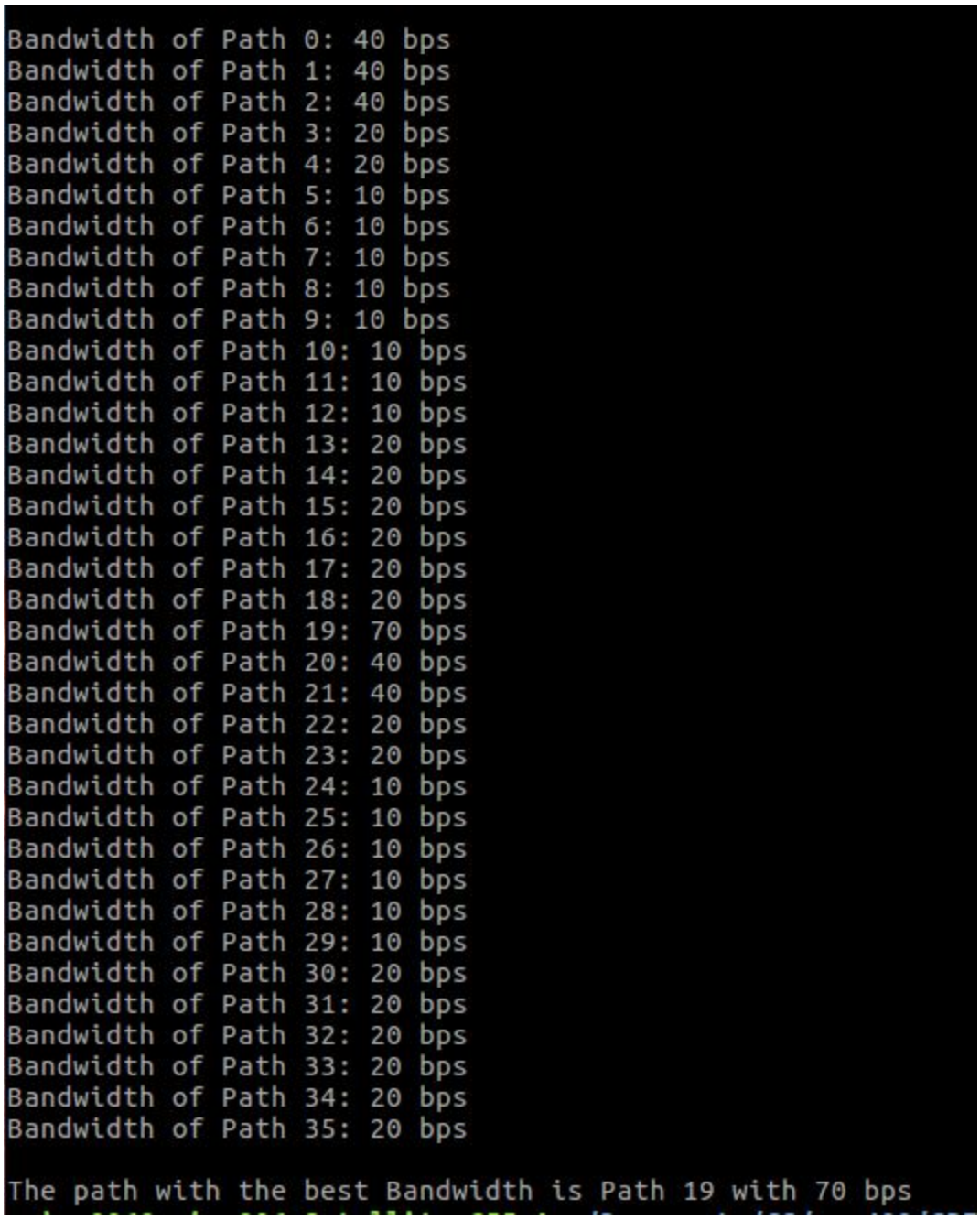

Fig 8. The bandwidths from Router 0 to Router 3 as printed by the program.

After all bandwidths B1 through Bn are found, the path taken is the one with the maximum bandwidth. Denoting this path as “mp,” we get Bmp = max(B1, B2, …, Bn), which from Figure 8 is 70 bps through path 19.

Novelty of the Design

The novelty of this algorithm is that it is both dynamic and reactive. It is dynamic because it can be ran from any source to any destination, without the rest of the network being know to the user. The algorithm uses a modified Depth First Search, so it discovers new routers and paths as it is ran. It is reactive because, if a new router is added to the network, or a router is taken off of the network, the algorithm can be ran again to find all the new paths from the source to the destination. Similarly, if the bandwidths change, the algorithm can be ran again. It also stores the path information, so that the Depth First Search does not need to be ran again. It simply needs to recalculate the bandwidths again with the list of known paths.

As mentioned prior the algorithm can be ran from any source to any destination. Figure 9. Shows The process with the source being router 5 and the destination being router 7.

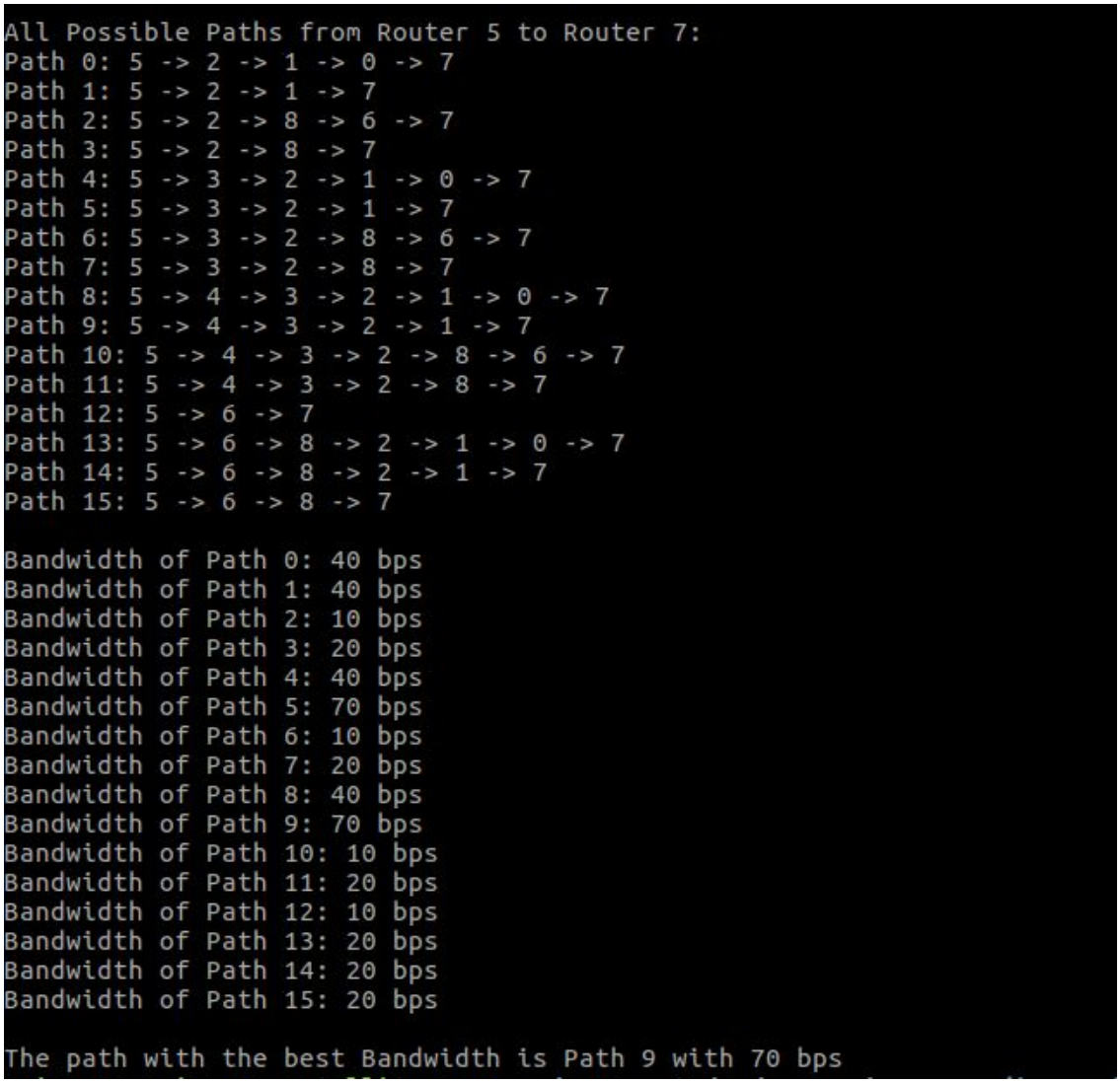

Fig 9. The program printout for the “Max Bandwidth Algorithm” from Router 5 to Router 7.

Figure 10 shows the program running with the central Router, Router 8, being removed from the network. Src = 0 and dest = 3.

```
orion996@orion996-Satellite-C55-A:~/Documents/CS/cpe400/CPE400CodingProject$ ./Program
Enter the Source Router as an Integer Id (eg. '3'): 0
Enter the Destnation Router as an Integer Id (eg. '3'): 3
Graph Adj Table:
0 | 40 | 0 | 0 | 0 | 0 | 0 | 80 |
40 | 0 | 80 | 0 | 0 | 0 | 0 | 110 |
0 | 80 | 0 | 70 | 0 | 40 | 0 | 0 |
0 | 0 | 70 | 0 | 90 | 140 | 0 | 0 |
0 | 0 | 0 | 90 | 0 | 100 | 0 | 0 |
0 | 0 | 40 | 140 | 100 | 0 | 20 | 0 |
0 | 0 | 0 | 0 | 0 | 20 | 0 | 10 |
80 | 110 | 0 | 0 | 0 | 0 | 10 | 0 |

All Possible Paths from Router 0 to Router 3:
Path 0: 0 -> 1 -> 2 -> 3
Path 1: 0 -> 1 -> 2 -> 5 -> 3
Path 2: 0 -> 1 -> 2 -> 5 -> 4 -> 3
Path 3: 0 -> 1 -> 7 -> 6 -> 5 -> 2 -> 3
Path 4: 0 -> 1 -> 7 -> 6 -> 5 -> 3
Path 5: 0 -> 1 -> 7 -> 6 -> 5 -> 4 -> 3
Path 6: 0 -> 7 -> 1 -> 2 -> 3
Path 7: 0 -> 7 -> 1 -> 2 -> 5 -> 3
Path 8: 0 -> 7 -> 1 -> 2 -> 5 -> 4 -> 3
Path 9: 0 -> 7 -> 6 -> 5 -> 2 -> 3
Path 10: 0 -> 7 -> 6 -> 5 -> 3
Path 11: 0 -> 7 -> 6 -> 5 -> 4 -> 3

Bandwidth of Path 0: 40 bps
Bandwidth of Path 1: 40 bps
Bandwidth of Path 2: 40 bps
Bandwidth of Path 3: 10 bps
Bandwidth of Path 4: 10 bps
Bandwidth of Path 5: 10 bps
Bandwidth of Path 6: 70 bps
Bandwidth of Path 7: 40 bps
Bandwidth of Path 8: 40 bps
Bandwidth of Path 9: 10 bps
Bandwidth of Path 10: 10 bps
Bandwidth of Path 11: 10 bps

The path with the best Bandwidth is Path 6 with 70 bps
orion996@orion996-Satellite-C55-A:~/Documents/CS/cpe400/CPE400CodingProject$
```

Fig 10. The program running with Router 8 removed.

Figure 11 and Figure 12 shows the program running with a new Router, Router 9, being attached to Router 2 with a bandwidth of 100 bps. Src = 0 and dest = 9.

```
orion996@orion996-Satellite-C55-A:~/Documents/CS/cpe400/CPE400CodingProject$ ./Program
Enter the Source Router as an Integer Id (eg. '3'): 0
Enter the Destnation Router as an Integer Id (eg. '3'): 9
Graph Adj Table:
0 | 40 | 0 | 0 | 0 | 0 | 0 | 80 | 0 | 0 |
40 | 0 | 80 | 0 | 0 | 0 | 0 | 110 | 0 | 0 |
0 | 80 | 0 | 70 | 0 | 40 | 0 | 0 | 20 | 100 |
0 | 0 | 70 | 0 | 90 | 140 | 0 | 0 | 0 | 0 |
0 | 0 | 0 | 90 | 0 | 100 | 0 | 0 | 0 | 0 |
0 | 0 | 40 | 140 | 100 | 0 | 20 | 0 | 0 | 0 |
0 | 0 | 0 | 0 | 0 | 20 | 0 | 10 | 60 | 0 |
80 | 110 | 0 | 0 | 0 | 0 | 10 | 0 | 70 | 0 |
0 | 0 | 20 | 0 | 0 | 0 | 60 | 70 | 0 | 0 |
0 | 0 | 100 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

All Possible Paths from Router 0 to Router 9:
Path 0: 0 -> 1 -> 2 -> 9
Path 1: 0 -> 1 -> 7 -> 6 -> 5 -> 2 -> 9
Path 2: 0 -> 1 -> 7 -> 6 -> 5 -> 3 -> 2 -> 9
Path 3: 0 -> 1 -> 7 -> 6 -> 5 -> 4 -> 3 -> 2 -> 9
Path 4: 0 -> 1 -> 7 -> 6 -> 8 -> 2 -> 9
Path 5: 0 -> 1 -> 7 -> 8 -> 2 -> 9
Path 6: 0 -> 1 -> 7 -> 8 -> 6 -> 5 -> 2 -> 9
Path 7: 0 -> 1 -> 7 -> 8 -> 6 -> 5 -> 3 -> 2 -> 9
Path 8: 0 -> 1 -> 7 -> 8 -> 6 -> 5 -> 4 -> 3 -> 2 -> 9
Path 9: 0 -> 7 -> 1 -> 2 -> 9
Path 10: 0 -> 7 -> 6 -> 5 -> 2 -> 9
Path 11: 0 -> 7 -> 6 -> 5 -> 3 -> 2 -> 9
Path 12: 0 -> 7 -> 6 -> 5 -> 4 -> 3 -> 2 -> 9
Path 13: 0 -> 7 -> 6 -> 8 -> 2 -> 9
Path 14: 0 -> 7 -> 8 -> 2 -> 9
Path 15: 0 -> 7 -> 8 -> 6 -> 5 -> 2 -> 9
Path 16: 0 -> 7 -> 8 -> 6 -> 5 -> 3 -> 2 -> 9
Path 17: 0 -> 7 -> 8 -> 6 -> 5 -> 4 -> 3 -> 2 -> 9
```

Fig 11. First half of the program running with a new Router 9

```
Bandwidth of Path 0: 40 bps
Bandwidth of Path 1: 10 bps
Bandwidth of Path 2: 10 bps
Bandwidth of Path 3: 10 bps
Bandwidth of Path 4: 10 bps
Bandwidth of Path 5: 20 bps
Bandwidth of Path 6: 20 bps
Bandwidth of Path 7: 20 bps
Bandwidth of Path 8: 20 bps
Bandwidth of Path 9: 80 bps
Bandwidth of Path 10: 10 bps
Bandwidth of Path 11: 10 bps
Bandwidth of Path 12: 10 bps
Bandwidth of Path 13: 10 bps
Bandwidth of Path 14: 20 bps
Bandwidth of Path 15: 20 bps
Bandwidth of Path 16: 20 bps
Bandwidth of Path 17: 20 bps

The path with the best Bandwidth is Path 9 with 80 bps
orion996@orion996-Satellite-C55-A:~/Documents/CS/cpe400/CPE400CodingProject$
```

Fig 12. Second half of the program running with a new Router 9

Results and Analysis

By testing each printed path bandwidth manually by using Figure 2, the information each time the program runs is correct. While the information is correct, and the algorithm is both dynamic and reactive, it does have two major issues:

1. It is inefficient with large networks
2. It is impractical to update the paths in real time.

Addressing the issue of efficiency, the modified DFS algorithm uses a nested for loop with a conditional recursive call inside the nested loop. This makes the time efficiency of the algorithm O(n^2) + O(Recursion). With a smaller network like Figure 2, the program runs very well, but it would stall if the network had 1000000 Routers. The way to fix this would be to find an algorithm for finding all paths between two routers that is less than the current running time.

This algorithm is also inefficient to be reactive in real time. Due to the running time of it, if it were to be run constantly, it would lag. The solution to fix this is to run it on a spread interval, to discover new paths. However, if it is ran spread out, new routers could be added or old routers could be deleted from the network without knowledge by the network. So, the algorithm should also be ran each time a router leaves or enters the network.