DYNAMIC ROUTING WITH EMPHASIS ON THROUGHPUT

Andrei Iorgulescu & Ethan Vito

University of Nevada, Reno
CPE 400

Dynamic Routing with Emphasis on Throughput

General Terms

Node (router), Bandwidth (packet size), Throughput (bits/second).

1. Introduction

Using C++, we created a simulation of a dynamic routing algorithm that selects the most optimal path from one node (a router) to another. The algorithm uses depth first search (DFS) to traverse through the network from a source node to a destination node. The algorithm decides on a path based on the edge weight from one node to another. Edge weights represent the cost of sending a packet form one router to another. This cost is determined by three different packet delay sources: transmission delay, nodal processing delay, and queueing delay. We did not include propagation delay since our design is modeled after a small local network and the effect on throughput would be negligible given transfer rates close to the speed of light.

2. Protocol Design

2.1 *Assumptions*

Our proposed solution assumes that the routers in the network are static in nature, meaning that once the values read from the input file, they cannot be changed in the program. Additionally, our solution assumes a decentralized information scheme, where routers will know information related to their physically connected neighbors, such as costs associated with delay and bandwidth.

2.2 Proposed Solution

Network information, such as router links, bandwidth, queue delay, and nodal processing delay will be stored in an input file and will be read by the program. The program will parse this information and create a corresponding adjacency matrix describing the network. Throughput will then be calculated for each router link as a preprocessing step to determining the optimal path to maximize throughput between two routers. Following the throughput calculation, the program will calculate all the possible paths between the source router and destination router. The program will then compare all the possible paths to each other and return the path that has the highest throughput.

2.3 Error Handling

The only situation which there is specific error handling for is dealing with the file. This is the only point of failure in the algorithm, so we throw errors based on issues with the file loading (incorrect structure/failure to open).

3. Solution

3.1 Reading in the Network

The Input file will be structured as follows:

Source Router	Destination Router	Bandwidth	Queue Delay	Nodal Processing Delay

The first two columns specify a link between two routers. The subsequent columns specify the link details between the routers, including the bandwidth of the link and any queuing and nodal processing delays that will affect the throughput. Table 1 below shows the contents of the input file used by the program.

Table 1: "Network.txt" Input File

Source Router	Destination Router	Bandwidth	Queue Delay	Nodal Processing Delay
0	1	40	2	3
0	2	70	5	8
1	0	40	2	3
1	3	70	1	0
2	0	70	5	8
2	3	100	0	9
3	1	70	1	0
3	2	100	0	9

Figure 1 below shows the corresponding network described by the input file.

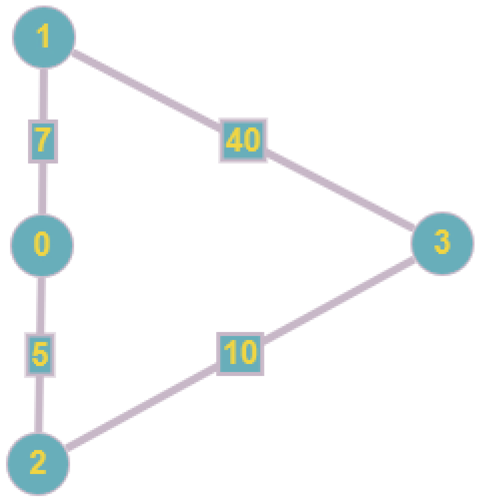


Figure 1: Graph of network with edge weights

After all the data is read in, the program will first calculate the throughput between each router and store the data in an adjacency matrix.

3.2 Calculating Throughput

Each line in the input file will be stored as an entry in a *paths* list. After every line is read, the program will begin calculating the throughput for each link in the network. Each path's throughput from source router and destination router will be calculated as follows:

Total nodal processing delay and total queuing delay have their own calculations. The total nodal processing delay is calculated by multiplying the nodal processing delay by the total number of packets that will be sent over the link. The total queuing delay is calculated by multiplying the queuing delay by one less than the total number of packets that will be sent over the link. The total number of packets is calculated by dividing the total number of bits that will be sent by the bandwidth of the link and rounding up to the nearest integer. We also considered the number of packets to be the transmission delay for the node as well. One packet is equivalent to a one second transmission delay.

Listed below are the specific calculations:

Number of Packets

$$\mathrm{round}\left(\frac{\mathrm{double(totalBits)}}{\mathrm{double(bandwidth)}}\right)$$

Total Queueing Delay

$$queueDelay * (numPackets - 1)$$

Total Nodal Processing Delay

$$nodeProcDelay * numPackets$$

Total Time to Send Bits (TTSB)

$$Num\ Packets + Node\ Processing\ Delay + Queueing\ Delay$$

Throughput

$$\frac{Total\ bits\ to\ send}{TTSB}$$

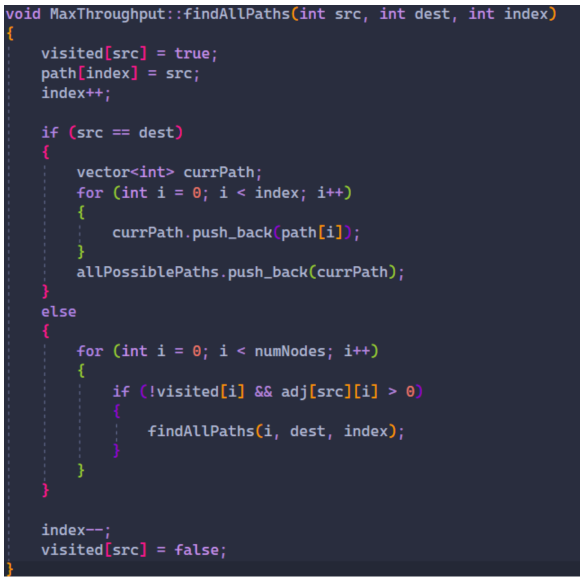
After the throughput is calculated for the path, the value is stored in an adjacency matrix. Using data from our “networks.txt” input file, the values can be seen in Table 2.

Table 2: Adjacency Matrix with Throughput Values

Nodes	0	1	2	3
0	0	7	5	0
1	7	0	0	40
2	5	0	0	10
3	0	40	10	0

3.3 Finding all Possible Paths between Source and Destination Routers

After populating the adjacency matrix with throughput values, the program will calculate all possible paths between the source and destination routers specified. This is done by using a Depth-First Search algorithm. Once a path is detected, the current path is appended to an *allPossiblePaths* list. The algorithm is defined in Figure 2.



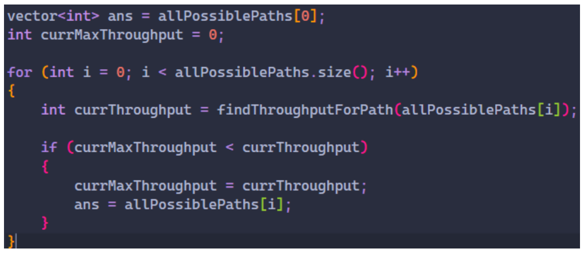
```
void MaxThroughput::findAllPaths(int src, int dest, int index)
{
    visited[src] = true;
    path[index] = src;
    index++;

    if (src == dest)
    {
        vector<int> currPath;
        for (int i = 0; i < index; i++)
        {
            currPath.push_back(path[i]);
        }
        allPossiblePaths.push_back(currPath);
    }
    else
    {
        for (int i = 0; i < numNodes; i++)
        {
            if (!visited[i] && adj[src][i] > 0)
            {
                findAllPaths(i, dest, index);
            }
        }
    }

    index--;
    visited[src] = false;
}
```

Figure 2: Function to find all paths using DFS

This algorithm follows closely with the classic Depth First Search Algorithm. Slight modifications have been made to prevent the algorithm from exiting until all paths have been discovered. Once the algorithm reaches the destination router, the current path is appended to the *allPossiblePaths* list. The source (which at this point is equal to destination) is then unmarked as being visited to allow the algorithm to call back up the recursive stack. Intermediary routers will then continue to be marked as unvisited until a new valid path can be created. After all the paths are found, the algorithm returns all the way back up the call stack and exits.

3.4 Finding Path with Maximum Throughput

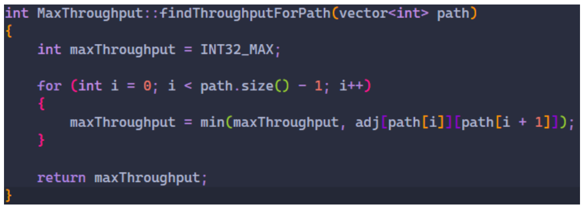
Now that the program has all the possible paths and throughput between routers stored, it becomes simple to find the path with the highest throughput. The maximum throughput of any path can be found by taking the minimum throughput between each link until the destination is reached. For example, given path 0 -> 1 -> 2, with throughput between 0 and 1 being 50 b/s and throughput between 1 and 2 being 30 b/s, the maximum throughput that can be achieved in this path is 30 b/s, since this the bottleneck. The optimal solution can be using the algorithm show in Figure 3 below.



```
vector<int> ans = allPossiblePaths[0];
int currMaxThroughput = 0;

for (int i = 0; i < allPossiblePaths.size(); i++)
{
    int currThroughput = findThroughputForPath(allPossiblePaths[i]);

    if (currMaxThroughput < currThroughput)
    {
        currMaxThroughput = currThroughput;
        ans = allPossiblePaths[i];
    }
}
```

Figure 3: Algorithm that discovers the optimal solution

We will initialize our answer to be the first path stored in *allPossiblePaths*, and initialize the maxThroughput to be 0. We will then loop through every path in *allPossiblePaths*, and calculate the throughput of the current path. If our current max throughput is smaller than the current throughput, then that means that we have found a better path to maximize throughput. The current max throughput is updated, and our answer is changed to the current path. Figure 4 below shows the algorithm to find the current throughput.



```
int MaxThroughput::findThroughputForPath(vector<int> path)
{
    int maxThroughput = INT32_MAX;

    for (int i = 0; i < path.size() - 1; i++)
    {
        maxThroughput = min(maxThroughput, adj[path[i]][path[i + 1]]);
    }

    return maxThroughput;
}
```

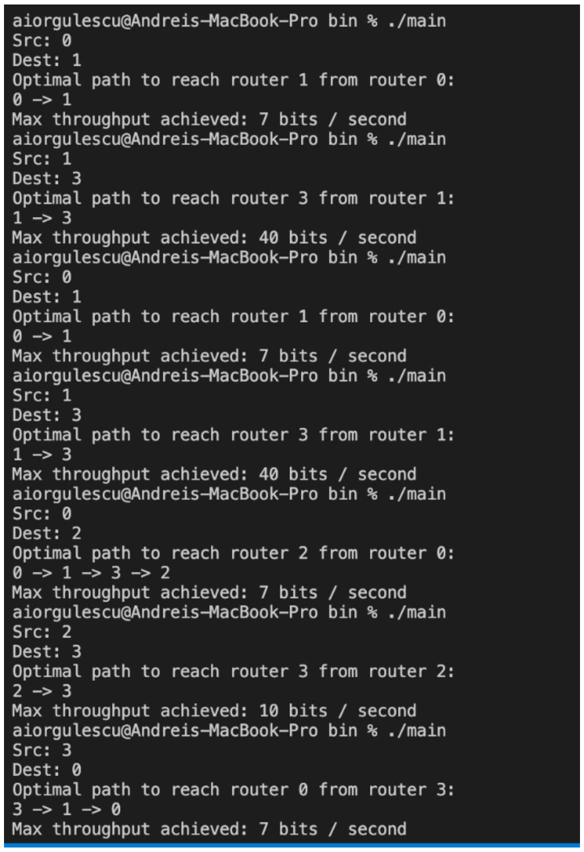
Figure 4: Algorithm that calculates throughput of a given path

To find the throughput for a given path, we first initialize the max throughput maxThroughput to be an arbitrarily large value. We then iterate through the path and take the minimum between the maxThroughput and the current throughput between the two routers. The max throughput is then returned.

4. Novelty of Design

This algorithm stands out for its capacity to be both dynamic and reactive. It is capable of being utilized from any source to any target. To work, a revised Depth First Search is used to detect any new routers and routes during the process. If a router is added or taken away from the network, the algorithm can be reset to discover new routes from the beginning to the desired destination. Additionally, we have tried to add as many real-life factors, such as bandwidth, queuing delay, and nodal processing delay, to create a routing mechanism that closely resembles real world routing mechanisms.

Shown in Figure 5, on the next page, is an example of the algorithm's dynamic ability. Given different source and destination routers, it can find the optimal path.



```
aiorgulescu@Andreis-MacBook-Pro bin % ./main
Src: 0
Dest: 1
Optimal path to reach router 1 from router 0:
0 -> 1
Max throughput achieved: 7 bits / second
aiorgulescu@Andreis-MacBook-Pro bin % ./main
Src: 1
Dest: 3
Optimal path to reach router 3 from router 1:
1 -> 3
Max throughput achieved: 40 bits / second
aiorgulescu@Andreis-MacBook-Pro bin % ./main
Src: 0
Dest: 1
Optimal path to reach router 1 from router 0:
0 -> 1
Max throughput achieved: 7 bits / second
aiorgulescu@Andreis-MacBook-Pro bin % ./main
Src: 1
Dest: 3
Optimal path to reach router 3 from router 1:
1 -> 3
Max throughput achieved: 40 bits / second
aiorgulescu@Andreis-MacBook-Pro bin % ./main
Src: 0
Dest: 2
Optimal path to reach router 2 from router 0:
0 -> 1 -> 3 -> 2
Max throughput achieved: 7 bits / second
aiorgulescu@Andreis-MacBook-Pro bin % ./main
Src: 2
Dest: 3
Optimal path to reach router 3 from router 2:
2 -> 3
Max throughput achieved: 10 bits / second
aiorgulescu@Andreis-MacBook-Pro bin % ./main
Src: 3
Dest: 0
Optimal path to reach router 0 from router 3:
3 -> 1 -> 0
Max throughput achieved: 7 bits / second
```

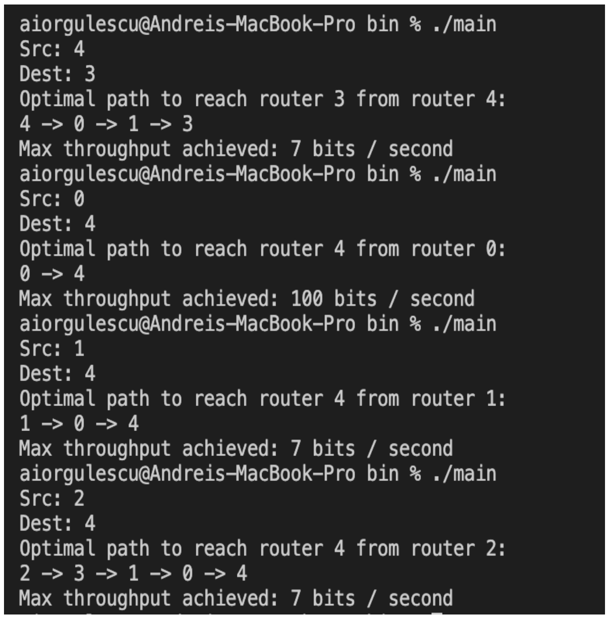
Figure 5: Maximum throughput path for multiple source and destination routers

We can demonstrate the algorithm's reactive ability by adding an additional router, router 4, to Table 1. Table 3 shows the added router information.

Table 3: Modified "Network.txt" input file

Source Router	Destination Router	Bandwidth	Queue Delay	Nodal Processing Delay
0	1	40	2	3
0	2	70	5	8
1	0	40	2	3
1	3	70	1	0
2	0	70	5	8
2	3	100	0	9
3	1	70	1	0
3	2	100	0	9
0	4	100	0	0
4	0	100	0	0
4	2	30	2	2
2	4	30	2	2

After recompiling the program, we can now find optimal paths with router 4 being either the source or destination. A figure is shown below illustrating this.



```
aiorgulescu@Andreis-MacBook-Pro bin % ./main
Src: 4
Dest: 3
Optimal path to reach router 3 from router 4:
4 -> 0 -> 1 -> 3
Max throughput achieved: 7 bits / second
aiorgulescu@Andreis-MacBook-Pro bin % ./main
Src: 0
Dest: 4
Optimal path to reach router 4 from router 0:
0 -> 4
Max throughput achieved: 100 bits / second
aiorgulescu@Andreis-MacBook-Pro bin % ./main
Src: 1
Dest: 4
Optimal path to reach router 4 from router 1:
1 -> 0 -> 4
Max throughput achieved: 7 bits / second
aiorgulescu@Andreis-MacBook-Pro bin % ./main
Src: 2
Dest: 4
Optimal path to reach router 4 from router 2:
2 -> 3 -> 1 -> 0 -> 4
Max throughput achieved: 7 bits / second
```

Figure 6: Maximum throughput path with router 4

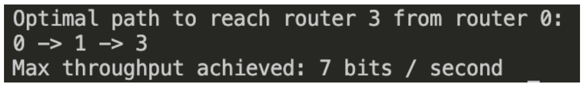
Please note that router "4" is not included in the project submission, as it is only an example to illustrate the algorithm's reactivity. The output that is shown in the results section is the output that should be mimicked if you run the main executable in the project submission.

5. Results

What we found is that the algorithm can determine the path that results in the most amount of throughput from a source node to a destination node. As discussed before, the graph of the network details information about the associated delays and bandwidth for nodes. The bandwidth determines the size of the packets that can be sent at a time and the delay information allows the throughput to be calculated for a given path.

Based on our calculations, and algorithm we can determine why this is. Paths with high bandwidth capabilities generally allow the most throughput if the associated delays for that path are not relatively high. A path with a smaller bandwidth but also fewer delay costs could be chosen over a path with higher bandwidth and larger delay costs. However, the effect of a node's bandwidth weighs more heavily on the throughput calculation. This is because a larger bandwidth value results in less packets needing to be sent and therefore fewer delay calculations.

5.1 Expected Program Output



```
Optimal path to reach router 3 from router 0:
0 -> 1 -> 3
Max throughput achieved: 7 bits / second  _
```

Figure 7: Expected program output

Figure 7 shows the expected program output. It does not include the previously added fourth router. The value we get here is 7 $\frac{bits}{second}$.

Although Table 2 and Figure 1 show throughput values of 40, those paths are limited by the weakest link along the path. Therefore, our max throughput is only 7 $\frac{bits}{second}$.