Multipath Routing Algorithm in OSPF Protocol

Shicong Wang

Biheng Yang

Xing Wang

Table of Contents **Page#**

Abstract

The original Open Shortest Path First (OSPF) is a routing protocol for Internet Protocol (IP) networks. From its name, we can tell that it is a single path algorithm. However, due to the complexity of the network traffic, multipath algorithm is needed in order to solve the problems we are facing today such as traffic congestion. We intention in this research paper is to find a multipath algorithm that not only can split the traffic to multipath but more importantly there is no congestion in any of the link in our chosen paths.

1. Introduction

1.1 Objective

This project attempts to build an efficient multipath routing algorithm for Open Shortest Path First (OSPF) networks, simulate the network in the algorithm by specific software and compare the results with the other algorithm in other research paper.

1.2 What is the problem?

The original Open Shortest Path First (OSPF) is a routing protocol for Internet Protocol (IP) networks. It uses a link state routing algorithm and falls into the group of interior routing protocols, operating within a single autonomous system. The OSPF routing policies for constructing a route table are governed by link cost factors (external metrics) associated with each routing interface. Cost factors may be the distance of a router (round-trip time), data throughput of a link, or link availability and reliability, expressed as simple unit less numbers. This provides a dynamic process of traffic load balancing between routes of equal cost.

In order to transfer the data from the source to the destination in the lowest cost as possible, the network needs to forward data through a series of intermediate nodes. As a result, this problem becomes complex in finding these series of intermedia nodes. So we need to find a method to determine these nodes to meet our requirement in OSPF protocol.

1.3 Why this is a project related to this class?

In the OSI model of computer networks, network layer is between the data link layer and transport layer. It transfers packets of data from the source to their destination through a number of intermediate routers.

The routing process consists of forwarding packets between a source and a destination through network of nodes linked to each other. Routing makes sure that all the packets reach their correct

destination with minimum delay. Stability, robustness, simplicity and fairness are desired qualities of any routing protocol. This project attempts to study and improve one multipath routing algorithm for the OSPF network.

1.4 Why other approach is no good

In “Optimal OSPF Traffic Engineering using Legacy Equal Cost Multipath Load Balancing”, the author splits the network flows evenly which will cause some links become overload whereas some other links become low-efficiently used.

In “A Multipath Routing Algorithm Based on OSPF Routing Protocol”, the author provided the DSMC multipath algorithm. However, in the pseudo codes provided in this paper, it didn’t consider the cost of the network. It chooses the single path one by one randomly to generate the multipath, and it may cause us to choose a big cost to transfer the data from source to destination.

1.5 Why you think your approach is better

In our project, our algorithm will choose the reasonable multipath link to transfer the data and consider the cost at the same time. We will improve the DSMC algorithm in the cost part, and the cost will be generated according to the formula in particular situation.

1.6 Statement of the problem

To account for high efficiency of cost in OSPF, an improved multipath routing algorithm is required. The algorithm needs to build the formula of the cost firstly according to the network environment. Then we can build the multipath routing algorithm which consider the actual network delay and other cause.

1.7 Area or scope of investigation

In this project, we focus our attention on following aspects:

1. Multipath routing algorithm

2. The specialty of OSPF protocol

2. Theoretical Bases and Literature Review

2.1 Definition of the problem and theoretical background

Most routing is a single optimal path and it may not be the most optimal path because the overused single path would be congested. In order to fully utilize the network and reduce congestion, a multipath algorithm is introduced. Related researches have been done on how to optimize pats in OSPF and use multipath in OSPF.

2.2 Related research and disadvantage of this research

In [3], a method to optimize paths in OSPF has introduced. The authors improved previous research by not using the congested links in an OSPF problem. However, the only condition that the network is multipath is that there are several equally shortest paths present. If there are conditions that there is only one optimal path and multipath solution is available, their research cannot find the multipath solution.

2.3 Our solution to solve this problem

Our solution to the problem includes three algorithms:

Algorithm 1: Path constraint algorithm.

The first algorithm will generate the proper number of paths we need to find based on current network capacity and the traffic flow we need to send.

Algorithm 2: Multipath finding algorithm.

The second algorithm is designed to find sufficient shortest path in peer to peer sub-network based on the cost of each link.

Algorithm 3: Traffic splitting algorithm.

The third algorithm will calculate the amount of traffic flow on each path.

2.4 Where our solution different from others

Our PMT algorithm is different from DSMC [1], as DSMC searches for good paths without taking the network capacity and traffic flow amount into consideration. OSPF-TE-Virt [2] is based on ideal situation which only takes effects on two peers, in contrast, our PMT is aimed on the whole network with a traffic splitting algorithm to provide the final flow on each link.

2.5 Why our solution is better

Comparing with DSMC [1], our PMT has path constraint and reduced complexity, we don't need to find all the available paths, but find the sufficient paths which can carry out our traffic flow with acceptable interference to the current network. As mentioned above, OSPF-TE-Virt [2] only takes one pair of the nodes, and figures out the proper virtual links on each link position, it is only a model. However, our PMT takes one more step close to the real world, which will definitely have a better result.

3. Hypothesis

The related researches on this area only focus on the optimal paths and multipath that split evenly on each link. We believed that these are not the most efficient way to fully utilize the network. Our hypothesis is that we can further improve the congestion issue by setting a path constraint to eliminate the already congested links, implementing our algorithm to not evenly split the traffic but to fully utilize each link's bandwidth, and most importantly using multipath that can improve the performance of the network traffic dramatically.

4. Methodology

4.1 How to generate/collect input data

We implement a small network generator using Java.

Firstly, it will generate the nodes in the network and the corresponding connection link among them.

Secondly, it will assign the propagation delay and bandwidth based on a random seed and some constraints made by us.

Furthermore, we have traffic flow generator, which will randomly choose two nodes inside the network and provide the flow amount between them.

Thus, we have the method to generate all the inputs we need.

4.2 How to solve the problem

As mentioned on Section 2.3, our PMT algorithm contains three steps.

The first step is to evaluate the routing table to get a network capacity overview, and try to figure out how many paths we need to transmit the current amount of traffic flow.

Then, PMT will call the multipath finding algorithm which is similar to DSMC [1], but use the path constraint as one of the input. The multipath algorithm will find the most available paths from source to destination according to the path constraint. For example, if the path constraint indicates PMT to find three available paths, PMT will stop the loop after it finds that number of paths, if it cannot find that many, it will try it best to find as more paths as possible (maybe 1 or 2). Finally, a traffic flow splitting algorithm will be used to determine the corresponding flow amount on each path.

4.2.1 Algorithm design

PMT Algorithm

1. Path Constraint Algorithm
2. Multipath Finding Algorithm
3. Traffic Split Algorithm

4.2.2 Language used

We will use Java as our main implementation language.

4.2.3 Tools used

JDK, Eclipse IDE.

4.3 How to generate output

At first, results will show us how network flows go through our link including which node the flow will pass and the sequence of these nodes. Then we will make our results friendlier to be accepted which include graphs comparing utilization of network for varying randomized input parameters and the cost of network in different randomized input.

4.4 How to test against hypothesis

While current hypothesis is limited, we can verify if the final design proposed by our team improves on overall cost by running this OSPF protocol through the same simulation and input parameters compared to the other algorithm. To validate the hypothesis, our algorithm need to achieve lower cost than others simulated.

5. Implementation

5.1 Code

Our simulator is a Java application with the following main Classes. For full code, please refer to Appendix section in the end.

1. GenerateNetwork class: Generate a simulation network with source, destination, and the bandwidth between each node.

2. GenerateOutput class: Randomly generate a starting (source) and an ending (destination) node. We are simulating our algorithm over these two nodes.

3. GenerateTable class: Generate a routing table for each node.

4. PathConstraint class: Give us an indication of number of paths we should use.

5. MultipathAlgorithm class: Find all the possible paths between source and destination.

6. SplitPath class: Split the flow into multipath from source to destination.

5.2 Design document and flowchart

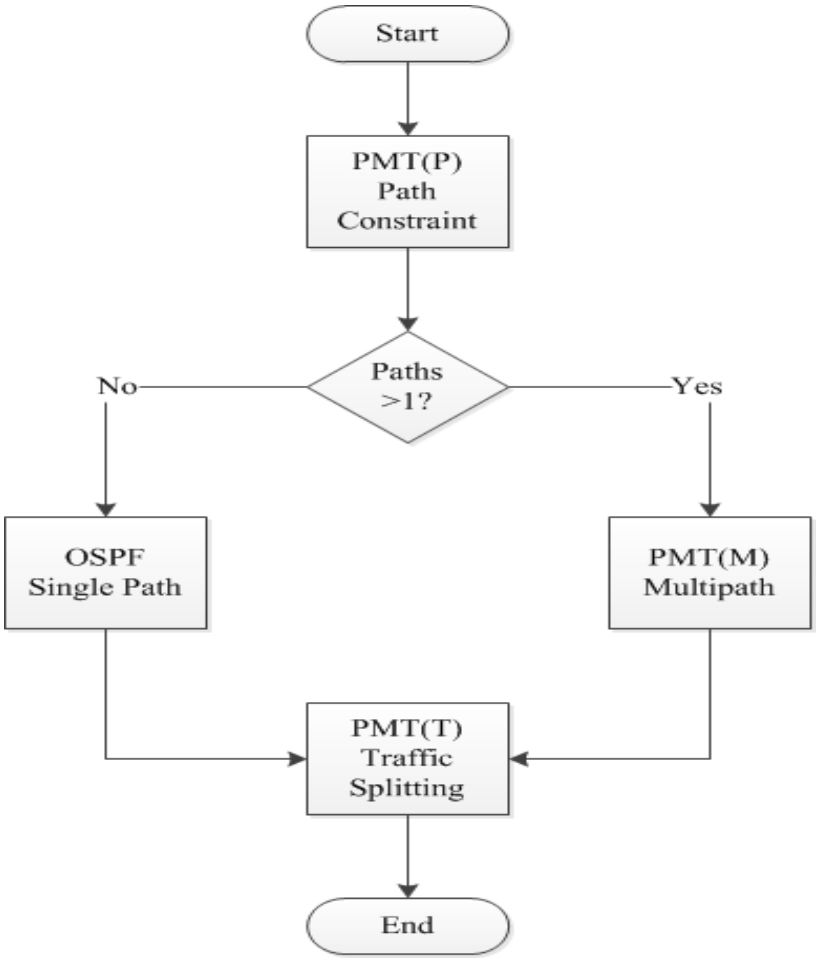

Figure 1

6. Data Analysis and Discussion

6.1 Output generation

1. We randomly generate a network with the number of nodes as parameter. This generated network has source, destination, and bandwidth between each node. As an example, we use 10 nodes to generate a network and the connectivity between each node is shown in table 1.

Source	Destination	Bandwidth (Mbps)
6	10	117
6	9	140
9	2	102
9	4	84
10	5	90
6	8	128
8	1	113
6	7	115
9	3	109
6	5	50
6	1	123
10	9	80
10	2	102
10	3	91
9	7	132
2	4	125
2	8	93
4	10	103
4	1	94
4	7	93
5	9	76
5	2	148
5	8	95
5	1	93
5	3	106
8	10	88
8	9	76
8	4	82
8	7	97
1	10	68

1	9	141
1	2	99
7	10	96
7	3	141
3	2	81
3	4	56

Table 1

We are also able to print the routing table and the connectivity of each node and its neighbors and we are going to use this routing table further for our multipath algorithm.

```
The Router 6's neighbors are: 10 9 8 7 5 1
The Router 9's neighbors are: 6 2 4 3 10 7 5 8 1
The Router 10's neighbors are: 6 5 9 2 3 4 8 1 7
The Router 8's neighbors are: 6 1 2 5 10 9 4 7
The Router 2's neighbors are: 9 10 4 8 5 1 3
The Router 4's neighbors are: 9 2 10 1 7 8 3
The Router 5's neighbors are: 10 6 9 2 8 1 3
The Router 1's neighbors are: 8 6 4 5 10 9 2
The Router 7's neighbors are: 6 9 4 8 10 3
The Router 3's neighbors are: 9 10 5 7 2 4
Router 6's bandwidthslist is:  117  140  128  115  50  123
Router 9's bandwidthslist is:  140  102  84  109  80  132  76  76  141
Router 10's bandwidthslist is:  117  90  80  102  91  103  88  68  96
Router 8's bandwidthslist is:  128  113  93  95  88  76  82  97
Router 2's bandwidthslist is:  102  102  125  93  148  99  81
Router 4's bandwidthslist is:  84  125  103  94  93  82  56
Router 5's bandwidthslist is:  90  50  76  148  95  93  106
Router 1's bandwidthslist is:  113  123  94  93  68  141  99
Router 7's bandwidthslist is:  115  132  93  97  96  141
Router 3's bandwidthslist is:  109  91  106  141  81  56
```

2. We randomly pick 1 source and 1 destination from the 10 nodes we generated. Then we generate a bandwidth that we want to send from source to bandwidth. This bandwidth is a random number but it is much larger than each link's bandwidth because we want to make sure multipath exists. Refer to sample output below.

Bandwidth is: 353Mbps
From 8 to 7:

3. We find possible paths between source and destination. We found 5 paths in our example.

Source	Destination	Path
8	7	8 - 7
8	7	8 - 6 - 7
8	7	8 - 1 - 6 - 10 - 7
8	7	8 - 2 - 9 - 7
8	7	8 - 5 - 10 - 9 - 4 - 7

Table 2

4. We split the bandwidth to each path. First we will set a limit for each path and the bandwidth that designate to each path cannot exceed its limit.

```
The 1th path limit is 97 Mbps
The 2th path limit is 115 Mbps
The 3th path limit is 96 Mbps
The 4th path limit is 93 Mbps
The 5th path limit is 80 Mbps
The 1th path gets 71.18711018711019 Mbps
The 2th path gets 84.3970893970894 Mbps
The 3th path gets 70.45322245322245 Mbps
The 4th path gets 68.25155925155926 Mbps
The 5th path gets 58.71101871101872 Mbps
```

6.2 Output analysis

First of all, our network and test data are generated randomly and it fits the nature of the randomness of the network traffic. Second, the bandwidth of our testing data is larger in order to make sure that there is multipath every time. Last, by using a constraint parameter path limit bandwidth, we are making sure that there is no congestion issue in every link which could possible cause the propagation issue and slow down the transmission speed. All of our allocated bandwidth is smaller than our limit bandwidth so it proves our intention which is that all the links are not fully utilized and we optimize each path's performance.

6.3 Compare against hypothesis

We indeed improve the performance of the network by using our PMT algorithm. We eliminate the already congested links and our split algorithm is not even split which is the algorithm used

in [1]. One problem of evenly splitting the bandwidth is that some links could be fully utilized. However, our algorithm eliminates this issue by using a constraint parameter.

6.4 Statistic regression

By using larger testing bandwidth, we have found more paths. Another parameter that would affect the number of paths is the total nodes in the network. More nodes can have more connectivity. As a result, the number of path is larger too. We test the above scenarios by increasing the testing bandwidth and total number of nodes and the number of paths is proportional to these two parameters.

7. Conclusions and recommendations

Although single path could be a solution to some of the networks, however single path definitely is not the most optimal because it may have the congestion issue. As a result, the introduction of multipath algorithm could solve this issue easily by splitting the bandwidth to different paths which could decrease the probability of the congestion of each link. However, without the proper definition, even multipath could also experience congestion in the case of the bandwidth of each link is too small and the transmission bandwidth is too large. Previous research had been done to split the bandwidth evenly as in research paper [1]. However, the above assumption could be the case in [1]. Our PMT algorithm further improves the splitting path algorithm. Our multipath does not split evenly, but instead it split to each path according to a path limit parameter. This parameter restricts the bandwidth being so large which could cause the congestion of the link. By applying this parameter, our PMT algorithm would make sure that every path will never reach its limit so there would be no congestion. Further research still needs to be done to optimize the calculation time and over performance.

8. Bibliography

[1] Li Lan, Li Li, Chen Jianya, “A Multipath Routing Algorithm Based on OSPF Routing Protocol”, Semantics, Knowledge and Grids (SKG), 2012 Eighth International Conference on, pp 269-272, Oct 2012.

[2] M. Dzida, M. Zagozdzon, M. Pioro, A. Tomaszewski, “Optimization of The Shortest-Path Routing with Equal-Cost”, Transparent Optical Networks, 2006 International Conference on, pp 9-12, June 2006.

[3] Makarem Bamatraf, Mohamed Othman, Rozita Johari, Shamala Subramaniam, “Optimizing Paths in OSPF Routing”, Industrial Informatics, IEEE International Conference on, pp 602-609, Nov 2

9. Appendices

Code

```
************************ GenerateNetwork Class **********************************
import java.util.ArrayList;
import java.util.Random;

public class GenerateNetwork {
	public ArrayList<Link> GenerateNetworkLink(int nodeNumber) {
		ArrayList<String> unvisited = new ArrayList<String>();
		ArrayList<String> visited = new ArrayList<String>();
		ArrayList<Link> generatedLink = new ArrayList<Link>();
		for (int i = 1; i <= nodeNumber; i++) {
			String tempString = String.valueOf(i);
			unvisited.add(tempString);
		}
		Random randomGenerator = new Random();
		int firstNode = randomGenerator.nextInt(unvisited.size());
		visited.add(unvisited.get(firstNode));
		unvisited.remove(firstNode);
		while (unvisited.size() != 0) {
			Link newlink = new Link();
			int visitedNode = randomGenerator.nextInt(visited.size());
			int unvisitedNode = randomGenerator.nextInt(unvisited.size());
			newlink.source = visited.get(visitedNode);
			newlink.destination = unvisited.get(unvisitedNode);
			newlink.bandwidth = 50 + randomGenerator.nextInt(100);
			newlink.cost = 100/(double)newlink.bandwidth;
			if(newlink.cost < 1) newlink.cost = 1;
			generatedLink.add(newlink);
			visited.add(unvisited.get(unvisitedNode));
			unvisited.remove(unvisitedNode);
		}
		// Tree built finished
		for (int i = 0; i < visited.size(); i++) {
			for (int j = 0; j < visited.size(); j++) {
				if (i != j) {
					boolean iflinkexists = false;
					int k = 0;
					while (k < generatedLink.size()) {
						if
(visited.get(i).equals(generatedLink.get(k).source) == true) {
							if (visited.get(j).equals(

	generatedLink.get(k).destination) == true) {
								iflinkexists = true;
								break;
							}
						}
						if (visited.get(i).equals(
								generatedLink.get(k).destination)
== true) {
							if (visited.get(j).equals(
```

```
    generatedLink.get(k).source) == true) {
                                iflinkexists = true;
                                break;
                            }
                        }
                        k++;
                    }
                    if (!iflinkexists) {
                        int choice = randomGenerator.nextInt(2);
                        if (choice == 1) {
                            Link newlink = new Link();
                            newlink.source = visited.get(i);
                            newlink.destination = visited.get(j);
                            newlink.bandwidth = 50 +
randomGenerator
                                    .nextInt(100);
                            newlink.cost =
100/(double)newlink.bandwidth;
                            if(newlink.cost < 1) newlink.cost = 1;
                            generatedLink.add(newlink);
                        }
                    }
                }
            }
        }
        for (int i = 0; i < generatedLink.size(); i++) {
            System.out.print("Link: source: " + generatedLink.get(i).source
                    + "    ");
            System.out.println("destination: "
                    + generatedLink.get(i).destination + " Bandwidth: "
                    + generatedLink.get(i).bandwidth + " Mbps");
        }
        return generatedLink;
    }

    public ArrayList<Router> GenerateNetworkRouter(ArrayList<Link> generatedLink)
{
        ArrayList<Router> generatedRouter = new ArrayList<Router>();
        for (int i = 0; i < generatedLink.size(); i++) {
            int j = 0;
            while (j < generatedRouter.size()) {
                if (generatedLink.get(i).source
                        .equals(generatedRouter.get(j).routerName) ==
true)
                    break;
                j++;
            }
            if (j == generatedRouter.size()) {
                Router newRouter = new Router();
                newRouter.routerName = generatedLink.get(i).source;
                generatedRouter.add(newRouter);
            }
        }
        for (int i = 0; i < generatedLink.size(); i++) {
```

```
                    int j = 0;
                    while (j < generatedRouter.size()) {
                          if
(generatedLink.get(i).destination.equals(generatedRouter
                                        .get(j).routerName) == true)
                                break;
                          j++;
                    }
                    if (j == generatedRouter.size()) {
                          Router newRouter = new Router();
                          newRouter.routerName = generatedLink.get(i).destination;
                          generatedRouter.add(newRouter);
                    }
              }
              for (int i = 0; i < generatedRouter.size(); i++) {
                    System.out.println("Router: " +
generatedRouter.get(i).routerName);
              }

              return generatedRouter;
       }

       public ArrayList<Router> GenerateRouterNeighbors(Router local,
                    ArrayList<Link> generatedLinks, ArrayList<Router>
generatedRouters) {
              ArrayList<Router> neighbors = new ArrayList<Router>();
              for (int i = 0; i < generatedLinks.size(); i++) {
                   if (local.routerName.equals(generatedLinks.get(i).source) == true)
{
                          for (int j = 0; j < generatedRouters.size(); j++) {
                                if (generatedRouters.get(j).routerName

       .equals(generatedLinks.get(i).destination) == true) {
                                        neighbors.add(generatedRouters.get(j));
                                        break;
                                }
                          }
                    } else if (local.routerName
                                .equals(generatedLinks.get(i).destination) == true)
{
                          for (int j = 0; j < generatedRouters.size(); j++) {
                                if (generatedRouters.get(j).routerName
                                            .equals(generatedLinks.get(i).source)
== true) {
                                        neighbors.add(generatedRouters.get(j));
                                        break;
                                }
                          }
                    }
              }

              return neighbors;
       }

       public ArrayList<Integer> GenerateRelatedBandwidth(Router local,
```

```
                ArrayList<Link> generatedLinks) {
            ArrayList<Integer> bandwidthSet = new ArrayList<Integer>(0);
            for (int i = 0; i < local.neighborRouters.size(); i++) {
                Link relatedlink = new GenerateNetwork().LocateLink(local,
                            local.neighborRouters.get(i), generatedLinks);
                bandwidthSet.add(relatedlink.bandwidth);
            }
            return bandwidthSet;
        }

        public Link LocateLink(Router a, Router b, ArrayList<Link> generatedLinks) {
            Link locatedlink = new Link();
            for (int i = 0; i < generatedLinks.size(); i++) {
                if ((generatedLinks.get(i).source.equals(a.routerName) &&
generatedLinks
                            .get(i).destination.equals(b.routerName))
                           || (generatedLinks.get(i).source.equals(b.routerName)
&& generatedLinks

        .get(i).destination.equals(a.routerName))) {
                    locatedlink = generatedLinks.get(i);
                    break;
                }
            }
            return locatedlink;
        }

        public int GetLinkIndex(Router a, Router b, ArrayList<Link> generatedLinks) {
            for (int i = 0; i < generatedLinks.size(); i++) {
                if ((generatedLinks.get(i).source.equals(a.routerName) &&
generatedLinks
                            .get(i).destination.equals(b.routerName))
                           || (generatedLinks.get(i).source.equals(b.routerName)
&& generatedLinks

        .get(i).destination.equals(a.routerName))) {
                    return i;
                }
            }
            return -1;
        }

        public Router GetRouterByName(String a, ArrayList<Router> generatedRouters) {
            for (int i = 0; i < generatedRouters.size(); i++) {
                if (generatedRouters.get(i).routerName.equals(a) == true) {
                    return generatedRouters.get(i);
                }
            }
            return null;
        }


        public static int numberOfNodes = 10;

        public static void main(String args[]) {
```

```
            System.out
                        .println("-------------------------Start test-------------
------------");
            GenerateNetwork a = new GenerateNetwork();
            ArrayList<Link> newlinks = a.GenerateNetworkLink(numberOfNodes);
            ArrayList<Router> newrouters = a.GenerateNetworkRouter(newlinks);
            Router sampleRouter = newrouters.get(3);
            ArrayList<Router> sampleNeighbor = a.GenerateRouterNeighbors(
                        sampleRouter, newlinks, newrouters);
            System.out.print("The neighbor routers of router "
                        + sampleRouter.routerName + " are: ");
            for (int i = 0; i < sampleNeighbor.size(); i++) {
                  System.out.print(sampleNeighbor.get(i).routerName + " ");
            }
            System.out.println();
            Link aaa = a.LocateLink(newrouters.get(1), newrouters.get(2), newlinks);
            System.out.println(aaa.source + " " + aaa.destination);
            int b = a.GetLinkIndex(newrouters.get(1), newrouters.get(2), newlinks);
            System.out.println(b);
            GenerateOutput generateVariable = new GenerateOutput();
            NodeOutput pathStatus = new NodeOutput();
            pathStatus = generateVariable.generateNode(numberOfNodes);
      }
}
```

********** GenerateOutput Class **********

```
import java.util.Random;

public class GenerateOutput {

      public NodeOutput generateNode(int size) {


            Random randomGenerator = new Random();
            NodeOutput output = new NodeOutput();
            output.startNode = randomGenerator.nextInt(size);

            output.finalNode = randomGenerator.nextInt(size);
            output.bandwidth = 300 + randomGenerator.nextInt(200);

            while (output.startNode == output.finalNode) {
                  output.finalNode = randomGenerator.nextInt(size);
            }

            System.out.println("Bandwidth is : " + output.bandwidth + "Mbps");

            return output;

      }

}
```

********** GenerateTable Class **********

```java
import java.util.ArrayList;

public class GenerateTable {

	public void GenerateRoutingTable(Router a, ArrayList<Link> network) {

		// initial variables
		String node = a.routerName;

		ArrayList<String> nodes = new ArrayList<String>();
		ArrayList<String> unvisited = new ArrayList<String>();
		ArrayList<String> visited = new ArrayList<String>();

		ArrayList<TableContent> routingTable = new ArrayList<TableContent>();

		int size = 0;

		// loop control
		int i = 0;
		int j = 0;

		// temp variable
		double tempcost = Double.POSITIVE_INFINITY;
		String tempnode = null;
		String tempparent = null;

		// record nodes in the network
		for (i = 0; i < network.size(); i++) {
			if (nodes.indexOf(network.get(i).source) == -1) {
				nodes.add(network.get(i).source);
			}

			if (nodes.indexOf(network.get(i).destination) == -1) {
				nodes.add(network.get(i).destination);
			}
		}

		// test nodes
		// System.out.println(nodes.size());

		// initial unvisited nodes
		for (i = 0; i < nodes.size(); i++) {
			if (nodes.get(i) != node) {
				unvisited.add(nodes.get(i));
			}
		}

		// test unvisited
		// System.out.println(unvisited.size());

		// initial visited
		visited.add(node);
		//System.out.println(node);
```

```
            // build routing table with Dijkstra
            while (unvisited.size() > 0) {

                  size = visited.size();

                  // System.out.println(size);
                  // System.out.println(node);

                  tempcost = Double.POSITIVE_INFINITY;

                  for (i = 0; i < size; i++) {
                        for (j = 0; j < network.size(); j++) {
                              if (network.get(j).source == visited.get(i)
                                          &&
visited.indexOf(network.get(j).destination) == -1) {
                                    if (visited.get(i) == node) {
                                          if (network.get(j).cost < tempcost) {
                                                tempcost = network.get(j).cost;
                                                tempparent = node;
                                                tempnode =
network.get(j).destination;
                                          }
                                    } else {
                                          if (network.get(j).cost
                                                      + routingTable.get(i -
1).cost < tempcost) {
                                                tempcost = network.get(j).cost
                                                            +
routingTable.get(i - 1).cost;
                                                tempparent =
network.get(j).source;
                                                tempnode =
network.get(j).destination;
                                          }
                                    }
                              }

                              if (network.get(j).destination == visited.get(i)
                                          &&
visited.indexOf(network.get(j).source) == -1) {
                                    if (visited.get(i) == node) {
                                          if (network.get(j).cost < tempcost) {
                                                tempcost = network.get(j).cost;
                                                tempparent = node;
                                               tempnode  =  network.get(j).source;
                                          }
                                    } else {
                                          if (network.get(j).cost
                                                      + routingTable.get(i -
1).cost < tempcost) {
                                                tempcost = network.get(j).cost
                                                            +
routingTable.get(i - 1).cost;
```

```
                                                        tempparent =
network.get(j).destination;
                                                       tempnode = network.get(j).source;
                                                }
                                        }
                                }
                        }
                }

                TableContent content = new TableContent();
                content.cost = tempcost;
                content.destination = tempnode;
                content.parent = tempparent;

                //System.out.println(content.cost);
                //System.out.println(content.destination);
                //System.out.println(content.parent);

                routingTable.add(content);

                unvisited.remove(tempnode);

                visited.add(tempnode);
        }

        // track parent to find port
        for (i = 0; i < routingTable.size(); i++) {
                        if (routingTable.get(i).parent == node) {
                                routingTable.get(i).port =
routingTable.get(i).destination;
                        } else {
                                for(j=0;j<routingTable.size();j++){
                                        if(routingTable.get(j).destination ==
routingTable.get(i).parent){
                                                routingTable.get(i).port =
routingTable.get(j).port;
                                        }
                        }
                }
        }

        a.routingTable = routingTable;

    }

}

****************************** Link Class *************************

public class Link {
    public String source;
    public String destination;
    public Integer bandwidth = 0;
    public double cost = 0;
}
```

********** MultipathAlgorithm Class (Main Class) **********

```java
import java.util.ArrayList;

public class MultipathAlgorithm {
	public boolean isFinished;
	public boolean isDeadEnd;
	public ArrayList<Router> visitedRouters;

	public ArrayList<ArrayList<Link>> searchpath(Router source_,
		Router destination_, ArrayList<Link> networklinks,
		ArrayList<Router> networkrouters, int pathnumber) {
		System.out.println("Start searchpath");
		ArrayList<ArrayList<Link>> pathSet = new ArrayList<ArrayList<Link>>();
		System.out.println("We have " + pathnumber + " path at most!");
		for (int i = 1; i <= pathnumber; i++) {
			System.out.print("The " + i + "th path is: ");
			ArrayList<Link> path = new ArrayList<Link>(0);
			isDeadEnd = false;
			isFinished = false;
			visitedRouters = new ArrayList<Router>(0);
		       path = searchsinglepath(source_, destination_, networklinks, path,
				networkrouters);
			if (path.size() == 0) {
				System.out.print("No more ways!!");
				break;
			}
			System.out.print("[");
			for (int j = 0; j < path.size(); j++) {
				System.out.print(path.get(j).source + " "
					+ path.get(j).destination + "  ");
			}
			System.out.println("]");
			pathSet.add(path);
			for (int j = 0; j < path.size(); j++) {
				Router left = new GenerateNetwork().GetRouterByName(
					path.get(j).source, networkrouters);
				Router right = new GenerateNetwork().GetRouterByName(
					path.get(j).destination, networkrouters);
				if (right != null && left != null) {
					left.neighborRouters.remove(right);
					right.neighborRouters.remove(left);
					networklinks.remove(path.get(j));
				}
			}
			System.out.print("\tThe left network is: ");
			for (int j = 0; j < networklinks.size(); j++) {
				System.out.print(networklinks.get(j).source + " "
					+ networklinks.get(j).destination + " \t");
			}
			System.out.println();
		}
		// find pathnumber ways
		return pathSet;
	}
```

```
	public ArrayList<Link> searchsinglepath(Router start, Router end,
		ArrayList<Link> way, ArrayList<Link> currentpath,
		ArrayList<Router> nodes) {
		if (isFinished)
			return currentpath;
		for (int i = 0; i < visitedRouters.size(); i++) {
			if (visitedRouters.get(i).routerName.equals(start.routerName) ==
true) {
				System.out.print("Circult!");
				isDeadEnd = true;
				return currentpath;
			}
		}
		visitedRouters.add(start);
		if (start.routerName.equals(end.routerName)) {
			isFinished = true;
			return currentpath;
		}
		for (int i = 0; i < start.neighborRouters.size(); i++) {
			if (start.neighborRouters.get(i).routerName.equals(end.routerName)
== true) {
				Link finalLink = new GenerateNetwork().LocateLink(start,
end,
						way);
				currentpath.add(finalLink);
				visitedRouters.add(end);
				System.out.print(finalLink.source + " " +
finalLink.destination
						+ "end \t");
				isFinished = true;
				return currentpath;
			}
		}
		if (start.neighborRouters.isEmpty() || end.neighborRouters.isEmpty()) {
			// remove extra links
			System.out.print("Dead!");
			isDeadEnd = true;
			return currentpath;
		}
		while (start.neighborRouters.size() != 0) {
			isDeadEnd = false;
			Link templink;
			templink = new GenerateNetwork().LocateLink(start,
					start.neighborRouters.get(0), way);
			if (templink != null) {
				currentpath.add(templink);
				System.out.print(templink.source + " " +
templink.destination
						+ "\t");
			} else
				break;

			int indexOflinkToRemove = new
GenerateNetwork().GetLinkIndex(start,
```

```
                        start.neighborRouters.get(0), way);
                start.neighborRouters.get(0).neighborRouters.remove(start);
                if (indexOflinkToRemove >= 0)
                        way.remove(indexOflinkToRemove);
                searchsinglepath(start.neighborRouters.get(0), end, way,
                        currentpath, nodes);
                if (isDeadEnd) {
                        currentpath.remove(templink);
                }
                if (isFinished) {
                        System.out.print("over");
                        break;
                }
                start.neighborRouters.remove(0);
        }
        return currentpath;
    }

    public static void main(String[] args) {
        ArrayList<Link> testNetworkLinks = new GenerateNetwork()
                .GenerateNetworkLink(50);
        ArrayList<Router> testNetworkRouters = new GenerateNetwork()
                .GenerateNetworkRouter(testNetworkLinks);
        for (int i = 0; i < testNetworkRouters.size(); i++) {
                testNetworkRouters.get(i).neighborRouters = new GenerateNetwork()
                        .GenerateRouterNeighbors(testNetworkRouters.get(i),
                                testNetworkLinks, testNetworkRouters);
        }
        for (int i = 0; i < testNetworkRouters.size(); i++) {
                System.out.print("The Router "
                        + testNetworkRouters.get(i).routerName
                        + "'s neighbors are: ");
                for (int j = 0; j < testNetworkRouters.get(i).neighborRouters
                        .size(); j++) {
                        System.out.print(testNetworkRouters.get(i).neighborRouters
                                .get(j).routerName + " ");
                }
                System.out.println();
        }
        for (int i = 0; i < testNetworkRouters.size(); i++) {
                testNetworkRouters.get(i).bandwidth = new GenerateNetwork()
                        .GenerateRelatedBandwidth(testNetworkRouters.get(i),
                                testNetworkLinks);
                System.out.print("Router " + testNetworkRouters.get(i).routerName
                        + "'s bandwithslist is: ");
                for (int j = 0; j < testNetworkRouters.get(i).bandwidth.size();
j++) {
                        System.out.print(" "
                                + testNetworkRouters.get(i).bandwidth.get(j)
+ " ");
                }
                System.out.println();
        }
        GenerateTable b = new GenerateTable();
        for (int i = 0; i < testNetworkRouters.size(); i++) {
```

```
                    b.GenerateRoutingTable(testNetworkRouters.get(i),
testNetworkLinks);
            }

            // output routing table for test
            /*
             * for (int i = 0; i < testNetworkRouters.get(1).routingTable.size();
             * i++) {
             * System.out.print(testNetworkRouters.get(1).routingTable.get(i).cost +
             * " "); System.out
             * .print(testNetworkRouters.get(1).routingTable.get(i).destination +
             * " "); System.out
             * .print(testNetworkRouters.get(1).routingTable.get(i).parent + " ");
             * System.out
             * .println(testNetworkRouters.get(1).routingTable.get(i).port); }
             */

            System.out.println("------------------------------");
            NodeOutput output = new GenerateOutput()
                        .generateNode(testNetworkRouters.size());
            int pathnumber = new PathConstraint().GenerateConstraint(
                        testNetworkRouters.get(output.startNode).routerName,
                        testNetworkRouters.get(output.finalNode).routerName,
                        output.bandwidth, testNetworkRouters);
            System.out.println("From "
                        + testNetworkRouters.get(output.startNode).routerName + "
to "
                        + testNetworkRouters.get(output.finalNode).routerName +
":");
            ArrayList<ArrayList<Link>> testpath = new MultipathAlgorithm()
                        .searchpath(testNetworkRouters.get(output.startNode),
                                    testNetworkRouters.get(output.finalNode),
                                    testNetworkLinks, testNetworkRouters,
pathnumber);
            ArrayList<Double> sp = new SplitPath()
                        .Split(testpath, output.bandwidth);
            for (int i = 0; i < sp.size(); i++) {
                  System.out.println("The " + (i + 1) + "th path gets " + sp.get(i)
                              + " Mbps");
            }

      }
}
```

********** NodeOutput Class **********

```
public class NodeOutput {
      public int startNode;
      public int finalNode;
      public int bandwidth;
}
```

********** PathConstraint Class **********

```
import java.util.ArrayList;

public class PathConstraint {
```

```
	public int GenerateConstraint(String source, String destination, int flow,
		ArrayList<Router> routers) {
		int i = 0;
		int j = 0;
		int k = 0;
		int l = 0;

		int pathconstraint = 1;
		int p = 0;

		int routersrc = 0;
		int routerdest = 0;

		double tempbandwidth = 0;

		int nexthop = 0;

		for (i = 0; i < routers.size(); i++) {
			if (routers.get(i).routerName.equals(source)) {
				routersrc = i;
			}

			if (routers.get(i).routerName.equals(destination)) {
				routerdest = i;
			}
		}

		nexthop = routersrc;

		for (i = 0; i < routers.size(); i++) {
			if (nexthop == routerdest) {
				break;
			}
			for (j = 0; j < routers.get(nexthop).routingTable.size(); j++) {
				if (routers.get(nexthop).routingTable.get(j).destination
						.equals(routers.get(routerdest).routerName))
{
					for (k = 0; k < routers.size(); k++) {
						if (routers.get(k).routerName.equals(routers

	.get(nexthop).routingTable.get(j).port)) {
							for (l = 0; l <
routers.get(nexthop).neighborRouters
								.size(); l++) {
								if
(routers.get(nexthop).neighborRouters.get(l).routerName == routers
								.get(k).routerName)
{
								if
(routers.get(nexthop).bandwidth.get(l) > tempbandwidth) {
								tempbandwidth =
routers.get(nexthop).bandwidth

	.get(l);
								}
```

```
                                                                nexthop = k;
                                                        }
                                                }
                                        }
                                }
                        }
                }
        }

        p = (int) Math.rint(flow * 1.8 / tempbandwidth);

        if (p <= 0) {
                pathconstraint = 1;
        }else{
                pathconstraint = p;
        }

        return pathconstraint;
    }

}
```

********** Router Class **********

```
import java.util.ArrayList;

public class Router {
    String routerName;
    ArrayList<Router> neighborRouters;
    ArrayList<Integer> bandwidth;
    ArrayList<TableContent> routingTable;
}
```

********** SplitPath Class **********

```
import java.util.ArrayList;

public class SplitPath {
    public ArrayList<Double> Split(ArrayList<ArrayList<Link>> pathSet, int flow) {
        ArrayList<Double> setways = new ArrayList<Double>(0);
        ArrayList<Integer> limit = new ArrayList<Integer>(0);
        for (int i = 0; i < pathSet.size(); i++) {
            Integer templimit = 10000;
            for (int j = 0; j < pathSet.get(i).size(); j++) {
                if (pathSet.get(i).get(j).bandwidth < templimit)
                    templimit = pathSet.get(i).get(j).bandwidth;
            }
            limit.add(templimit);
            System.out.println("The " + (i+1) + "th path limit is " +
templimit + " Mbps");
        }
        Integer sum = 0;
        for (int i = 0; i < limit.size(); i++) {
            sum += limit.get(i);
        }
        for (int i = 0; i < limit.size(); i++) {
            Double tempRatio;
```

```
                tempRatio = (double) limit.get(i) / (double) sum;
                setways.add(tempRatio * flow);
            }

            return setways;
        }
}
```

********** TableContent Class **********

```
public class TableContent {

    //destination
    String destination;

    //port
    String port;

    //Dijkstra parent
    String parent;

    //total cost
    double cost;

}
```