Comprehensive Evaluation of Shortest Path Algorithms and Highest Bottleneck Bandwidth Algorithm in Software Defined Networks

SUBMITTED BY:

Asif Sadat Sajid (14101168)

Syed Faysal Nasir Niloy (14101196)

Kh. Arham Hossain (14101195)

Tasfin Rahman (14301058)

Department of Computer Science and Engineering

SUBMITTED TO:

Dr. Amitabha Chakrabarty

Assistant Professor

Department of Computer Science and Engineering

BRAC University

Date of Submission: 17/04/2018

Declaration

We declare that this report is our own work, except for extracts and summaries for which the original references are stated herein.

Signature of Supervisor

Amitabha Chakrabarty, Ph.D

Assistant Professor

Department of Computer Science and Engineering

BRAC University

Signature of Author

Asif Sadat Sajid

Id:14101168

Kh. Arham Hossain

Id: 14101195

Syed Faysal Nasir Niloy

Id: 14101196

Tasfin Rahman

Id:14301058

ABSTRACT

Traditional network architectures are ill-suited to meet the necessities of the present enterprises, carriers, and end users. Hence, new emerging network architecture, Software Defined Network (SDN) can be used to solve the problems since it is more dynamic, manageable, adaptive and programmable. The control and data planes in SDN architecture are decoupled, network intelligence and state are legitimately centralized, and the basic network infrastructure is abstracted from applications. However, a network may reach a point where data flow which is controlled according to the bandwidth is limited by computer or network resources. When bandwidth is unable to accommodate large amounts of system data at designated data transfer rate speeds, it results in bottleneck. We utilized an existing algorithm that can improve network performance by detecting bottlenecks and route packets through the highest bottleneck bandwidth which has been implemented using java programming language. Moreover, although many algorithms have been proposed and used in various applications, a comprehensive evaluation and quantitative comparison of shortest path algorithms in SDN applications is missing. In this paper, we will narrow that gap giving an up-to-date survey of SPF Algorithms in SDN applications.

Keywords—software-defined networks; highest bottleneck bandwidth; shortest path;

Acknowledgement

Our thesis work is based on the evaluation of shortest path network algorithm and simulation of highest bottleneck bandwidth algorithm in software-defined networks. We are very pleased to acknowledge our utmost appreciation to Dr. Amitabha Chakrabarty, Assistant Professor of Department of Computer Science, BRAC University who suggested us to work on this topic. We are much honored to be able conduct this research under his supervision.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1
Introduction

In today's world where the use and exploitation of internet is growing rapidly, networks are becoming more difficult to maintain. Hence the rate of internet traffic, loads, packet loss and demand for bandwidth are increasing. Routing algorithms especially the shortest path routing algorithms [1] are used for maintaining a network's QoS (Quality of Service) [2]. With the shortest path algorithms used in network applications, packets can now reach their destinations with a very low delay time, in other words the process becomes faster. But yet the problem remains as end devices cannot be utilized to their full capacity due to a recent trending problem known as bottleneck [3]. We propose a solution with the help of an existing algorithm that detects bottleneck in every possible path from source to destination, computes the highest bottleneck bandwidth and allows the packet to take the path with the help of switches. The algorithm will be implemented in a software-defined network (SDN) [4] environment and we will evaluate the performance of the network using the algorithm and then by randomizing the algorithm function to observe the difference. The algorithm implemented in SDN environment not only detects bottlenecks, it also provide the updates of the entire network topology within constant interval of time and hence the route table gets updated which tells us whether any of the switch is dead. Since the controller holds every switch's information, it avoids taking the trouble of routing all the switches to obtain the end-to-end link information.

Chapter 2.1-2.6 gives brief information of our comprehensive survey where we subdivided our analysis with different applications and issues. Inthis paper, we principally deal with two SPF algorithms: Bellman-Ford [1] and Dijkstra [1],

including their optimized and extended versions. In Chapter 2.7, we propose an existing algorithm that computes the highest bottleneck bandwidth [5] thus preventing the packets from reaching the choke points [5]. In Chapter 5, we will display the results from both the survey and our own simulation of the algorithm. Moreover, we will draw comparisons to give a valid conclusion of the results to help evaluate the network performance in SDN environment.

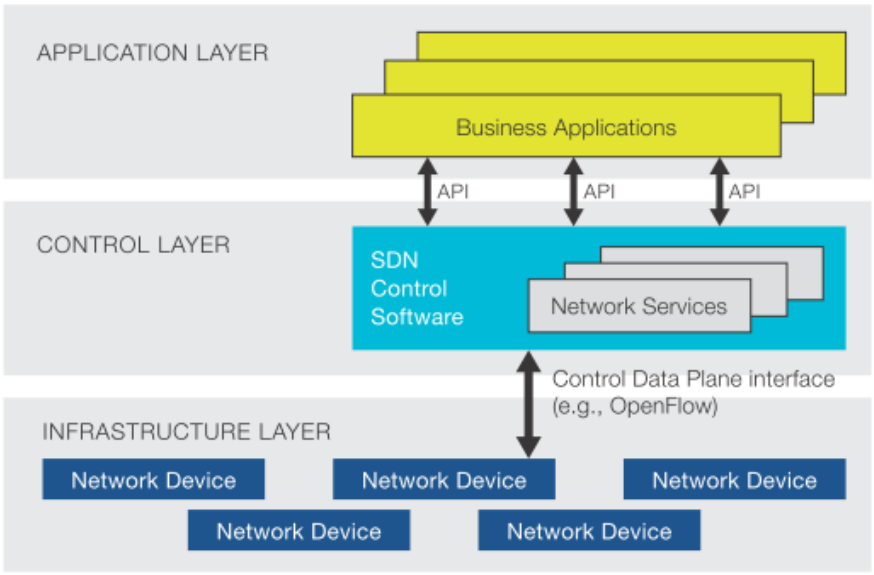

Fig 1.1: Generalized Architecture of SDN[11]

1.1 Motivation:

The emerging of Software-Defined Networking has put already put a huge change in Computer Networks. With the internet growing, it is important to maintain the huge, complex topologies to make sure that data is successfully sent to the end users and with Software-Defined Networks, it is easier to maintain because the network is more dynamic, programmable, scalable and more secured. The idea and the concept is not new but the implementations have started very recently and the success that it has brought in a short time motivated us to contribute in this field as it requires more research, more development and modification to improve further. Once it develops

further, network devices will come with software development kits and open APIs, enabling a new world of networking applications to evolve.

We contributed a comprehensive survey on the implementations of Shortest Path Algorithms in SDN Applications where we have provided the up-to-date information about the latest applications of SDN and how it is improving today's networks. Moreover, we used an existing algorithm in Software Defined Network environment that finds the Highest Bottleneck Bandwidth [5] and prevents data from reaching the Choke Points [5] of that particular topology in the network.

1.2 Methodology:

In the survey, we chose two of the most well-known Shortest Path algorithms used today which is Dijkstra's Algorithm and Bellman-Ford Algorithm [1]. From our research, we found how an existing algorithm alone cannot solve all sorts of problems such as flooding loops, load balancing issue, link failures, data loss and many more problems involving transfer of data. Hence, algorithms were modified, extended and optimized according to one's needs in order to maintain the networks' Quality of Service (QoS) [2]. Shortest Path problem is one of the four major QoS problems which we will be conveying in our research.

Bandwidth is also a QoS parameter which is widely measured. In Bottleneck bandwidth problem, we used an existing algorithm that finds highest bottleneck bandwidth from source to destination, allowing the packets to utilize the maximum bandwidth. Moreover, apart from calculating the path cost, we also keep track of the choke links and the parameters that evaluate the algorithm's performance in SDN environment.

1.3 Objective:

Our objective from the survey and the implementation of the algorithm is to reduce network traffic, link failures, data loss, faster flow rate of data, loops that affects the network. We will provide complete information about the implementation and the results of the algorithms used in Software-Defined Networks application where the algorithms are evaluated in terms of their performance parameters that ensures QoS. In bottleneck bandwidth problem, we created in total of 6 topologies and the connections between the nodes providing the path cost and delay time. We then apply the parameters to our simulation settings that evaluate an algorithm's performance. In order to justify our evaluation we changed the delay time for each node and simulated the topologies. Moreover, we randomized the algorithm to compare our results and draw conclusion.

1.4 Thesis Overview:

Chapter 1: Introduction to SDN, Research Methodology used and Objective of our Thesis

Chapter 2: Background concepts and previous works

Chapter 3: Topologies used in our paper

Chapter 4: Algorithm Implementation

Chapter 5: Results of our survey and simulations

Chapter 6: Conclusion and future plan of our research

CHAPTER 2

Literature Review

Basic understanding of this subject is very crucial because the implementation of Software-Defined Networking started very recently. SDN controllers are built using relatively small collections of tightly-coupled servers, which makes them amenable to distributed algorithms that maintain consistent versions of network-wide structures such as topology, traffic statistics, and others. SDN does not specify how controllers are implemented, it can be used to implement a variety of network algorithms, including simple ones such as shortest-path routing, and more sophisticated ones such as traffic engineering [7]. Many novel applications have been implemented with SDN including policy-based access control, adaptive traffic monitoring, wide-area traffic engineering, network virtualization, and others. SDN controllers manage the entire network, so they must often change rules on multiple switches. Although the network contains a mixture of rules from the old and new configurations during the transition, these rules have the property that any given packet will be processed according to a single version. Similar mechanisms can be used to implement per-flow consistency.

2.1 SPF Routing Algorithms in SDN Application:

Based on category of problems, there are four types of QoS based routing algorithms [2]:

- Shortest Path (SP)
- Constrained Shortest Path (CSP)
- Multi-Constrained Shortest Path (MCSP)
- Multi-Constrained Path (MCP)

We will be demonstrating our survey based Shortest Path Problems. Today's networks are dominantly based on the Shortest Path First (SPF) routing algorithms that assign weighting factors to links statically. Network traffic is often unevenly distributed, and causes link congestion even when the total load is not particularly high. This is very problematic for multimedia applications that require certain QoS [2] (Quality of Service) level for proper functioning. Providing quality of service (QoS) is an important requirement for a wide range of communication network settings and applications. Generally, the route determination (computation) is either carried out in distributed nodes, e.g., the control modules in individual distributed Internet Protocol (IP) routers, or by a centralized controller, e.g., a Software-Defined Networking (SDN) controller [4].The main goal of Shortest Path (SP) problems is to minimize a unique end-to-end QoS metric. We will be conveying our research on two of the most well-known algorithms: Bellman-Ford and Dijkstra Algorithm.

2.2 Contributions of this Survey

Our research article provides an up-to-date survey on applications of Bellman-Ford and Dijkstra's Algorithm in Software-Defined Networks. With our assessments and the results obtained, we will only present the significant and insightful information in form of evaluation table.

2.3 Basic Definitions for Shortest Path Algorithms

Dijkstra's Algorithm [1]: The Dijkstra's algorithm consists of n iterations. If all vertices have been visited, then the algorithm finishes; otherwise, from the list of unvisited vertices we have to choose the vertex which has the minimum (smallest) value at its label (At the beginning, we will choose a starting point s). After that, we will consider all neighbors of this vertex (Neighbors of a vertex are those vertices

that have common edges with the initial vertex). For each unvisited neighbor we will consider a new length, which is equal to the sum of the label's value at the initial vertex v (*d[v]*) and the length of edge *l* that connects them. If the resulting value is less than the value at the label, then we have to change the value in that label with the newly obtained value.

Bellman-Ford Algorithm [1]: In comparison to Dijkstra's algorithm, the Bellman-Ford algorithm admits or acknowledges the edges with negative weights. That is why, a graph can contain cycles of negative weights, which will generate numerous number of paths from the starting point to the final destination, where each cycle will minimize the length of the shortest path. Taking into consideration this fact let's assume that our graph does not contain cycles with negative weights. The algorithm consists of several phases, where in each phase it needs to minimize the value of all edges by replacing *d[b]* to following statement *d[a] + c*; *a* and *b* are vertices of the graph, and *c* is the corresponding edge that connects them.

Now that we are familiar with the basic algorithm and their structure, we will show how they can be utilized in different Software-Defined Networking Applications.
The purpose of both algorithms describes the way to find the nodes shortest path problem. The primary difference in these algorithms is the fact that Dijkstra carries the overall information of the network in every node. On the other hand, Bellman-Ford algorithm doesn't care about the overall network costs where each node needs to know the cost or distances with immediately neighbor nodes. Another difference is when a negative cost is available in the routes. In such case the Dijkstra algorithm cannot be useful. However, if native values are in the network, the Bellman-Ford method can handle the process to calculate the minimum route.

2.4 Demonstration of Single Link Failure Recovery:

Link failure management [8] is considered to be one of the key tools to make the network more reliable. SDN technology however holds an important role in maintaining system's fault tolerance. Shortest path algorithms holds significant importance and draws out the key factor in the execution of routing protocols with a specific end goal to give the fault tolerant arrangement in a failure recovery mechanism of a network system. In node failure recovery research, a comparative analysis of Bellman Ford and Dijkstra algorithm is performed by S. Waleed, M. Faizan, M. Iqbal and M. I. Anis (2017). These two unique algorithms show the importance of the actualized components by copying undirected mesh topology. Through repetitive connections, the introduced model furnishes a most limited way with decreased meeting deferral and offers more proficient bandwidth capacity with an ideal way without closing down any connections. [8][9]

Shortest path algorithms holds immense importance and brings out the key factor in the implementation of routing protocols in order to provide the fault tolerant solution in a failure recovery mechanism of a network system. In [10], basic concept of shortest path algorithm is implemented to find the shortest path through a graph. Recently, different algorithms and techniques are being introduced for shortest path calculation and these techniques are adding dynamic solutions to the problems at hand. In [11], heuristic methodology is utilized to rapidly reach a most ideal answer, or 'optimal solution'. This approach increases the effectiveness of this algorithm with respect to shortest path algorithm. Compute Unified Design Architecture (CUDA) is utilized for Bellman ford which decreases at least half of the shortest path calculation time making it more versatile and capable of working efficiently. The

Bellman-Ford algorithm enables unwinding to discover single source most limited ways on coordinated graphs.

Furthermore, it additionally contains negative edges. According to [12], the essential structure of bellman-ford algorithm is similar to Dijkstra algorithm. [t unwinds each one of the edges, and does this [V] - [time, where [V] is the quantity of vertices in the chart. The cost of a path is the total of edge weights in the path. Dijkstra's algorithm is an avaricious algorithm that tackles the briefest path issue from an offered source to the various nodes of a coordinated diagram. [t is generally utilized as a part of routing protocols, for example, OSPF (Open Shortest Path First). It can be utilized as a part of each link-state routing protocol: that is the point at which each node knows the entire topology of the system (and each time it changes, the entire guide is upgraded at every node).

In this paper, a comparative analysis of Bellman Ford and Dijkstra algorithm is performed with the capability of recovering from a link failure using an alternative path. These two different algorithms demonstrate the effectiveness of the implemented mechanisms by emulating undirected mesh topology. Through redundant links, the presented model provides a shortest path with reduced convergence delay and offers more efficient bandwidth with an optimum path without shutting down any links. This gives faster transmission of data with immense fault tolerance creating a better approach of migration within the network.

2.5 Extended Dijkstra for Load Balancing:

Lately, different load balancing strategies (e.g LABERIO (Load-Balancing Routing with OpenFlow) [13], LOBUS (Load-Balancing over Unstructured systems) for Data Center Networks (DCNs) utilizing the SDN world view have been presented

to decrease inertness and increase throughput. The extended Dijkstra's algorithm can be connected to infer a couple of briefest way in a SDN topology. Afterward, the Extended Dijkstra's algorithm for SDN is changed by using REST API of the controller and acquaints a clog control segment with handle movement overhead in a SDN topology.

Later the Extended Dijkstra Algorithm [14] was proposed by Ananta, M.T. & Jiang, Jehn-Ruey & Muslim, M.A. (2014) forgenerating multicast tree for a data publisher to deliver data packets to all subscribers to reduce bandwidth consumption.

Extended Dijkstra was further modified by Abdul Aziz, Abdul-hafiz &Adedokun, Emmanuel & Man-Yahya, Sani. (2017) to Improved Extended Dijkstra algorithm [15] by including a component called REST API which has to be authenticated against the controller.

2.5.1Extended Dijkstra's Algorithm for SDN:

Given a weighted, directed graph G = (V, E) and a single source node *s*, the classical Dijkstra's algorithm can return a shortest path from the source node *s* to every other node, where V is a set of nodes and E is the set of edges, each of which is associated with no weight. In the original Dijkstra algorithm nodes are associated with no weight. However, ED-SDN returns the shortest path from the single source node to every other with the consideration of the edge weight and the node weight [15].

Extended Dijkstra's Algorithm
Input: G=(V, E), ew, nw, s **Output:** d[|V|], p[|V|] d[s]←0; d[u]←∞, for each u≠s, u∈V **insert** u with key d[u] into the priority queue **while** (Q≠∅) u←Extract-Min(Q) **for** each v adjacent to u **if** d[v] >d[u]+ew[u,v]+nw[u] **then** d[v]←d[u]+ew[u,v]+nw[u] p[v]←d[u]

Fig 2.1: Extended Dijkstra Algorithm [15]

2.5.2Description of Improved Extended Dijkstra's Algorithm for SDN:

The main component of the Improved Extended Dijkstra's Algorithm for SDN (mED-SDN) is REST. The application has to be authenticated against the controller to make REST API calls to the controller. To fetch the URL through the API and retrieve some of the variables that are available,a section of the python script uses HTTP POST.For example, information about all nodes (hosts) on the network. The API will respond with an HTTP GET response message once it receives this. The HTTP traffic from host with an IP address *10.0.0.1* to destination of *10.0.0.12* will be sent out of port 1 on the switch. We set the flow priority to 3200, however the default priority, on the controller is 2990.To ensure that the traffic will use the path as set by this script rather than the default from the controller,the script puts the flow

priority higher than the default. When there is no traffic matching the flow entry,the idle timeout signifies how long the flow entry will stay in the switch. The congestion control component of this algorithm uses the bandwidth utilization as its evaluation criterion for improving congestion in an SDN topology. It reverts to the controller to search for a new path when the bandwidth utilization exceeds the threshold value set by the algorithm. In other words, to obtain the link bandwidth utilization, the congestion component proactively measures the bandwidth in the topology and utilizes the REST API on the controller to collect cumulative transmitted bytes at corresponding OpenFlow switches port.

2.6 Dynamic Programmability of Controller:

Bellman – Ford algorithm was being implemented by Shivendu, Arnav & Dhakal, Dependra & Sharma, Diwas (2015), to find the shortest path between two nodes in a network using SDN environment using POX API [17].

Controller will regularly have center administrations to help in employment of interfacing with arrange hubs and for giving a programmable interface to organize application. Information device or forwarding device receive packets, make a move on those bundles and refresh counters. Types of action include dropping of packets, modifying the header, sending bundle to single or various ports. Direction to how to deal with the bundles starts with SDN controller. This packet likewise stores this data for some time later. The most famous standard protocol used in SDN is OpenFlow. Switches are unable to function without being programmed by the controller.

SDN uses a centralized controller to generate flow tables that configures the forwarding table responsible for forwarding the packets in the network. Controller will typically have core services to aid in job of interfacing with network nodes and for providing a programmable interface to network application. Data device or

forwarding device receive packets, take action on those packets and update counters. Types of action include dropping of packets, modifying the header, sending packet to single or multiple ports. Instruction to how to handle the packets originates with SDN controller [4]. This device also caches this information for future use. Future packets of the same type can take a fast path with no need to contact the SDN controller. Application programs can be run on top of a controller to monitor and manage the network in a centralized manner.

2.7 Bottleneck Bandwidth Problem:

With global Internet traffic growing by an estimated 22% per year, the demand for bandwidth is fast outstripping providers' best efforts to supply it. Providing higher bandwidth is just not enough because that involves higher cost which both the providers and the consumers cannot afford. Therefore, to handle the issue with limited costs, the best we can do is control the bandwidth with which the data are being sent from source to the desired destination. We can eliminate the paths that have lower bandwidth (bottleneck) and select a path with the highest bottleneck bandwidth using an existing algorithm. Identifying network bottlenecks is very useful for end users and service providers. Unfortunately, it is very hard to identify the location of bottlenecks unless one has access to link load information for all the relevant links. Software Defined Networks however solves this problem and makes it much easier for both the users and operators because the controller already has the access to all the information of a topology. In this section, we used an existing algorithm that detects the bottleneck bandwidth and configure switches that allow the packets to route through the path with the highest bottleneck bandwidth. [3] [18]

2.7.1 Highest Bottleneck Bandwidth Algorithm:

The Highest Bottleneck Bandwidth (HBB) [18] algorithm is derived by modifying the Dijkstra's Algorithm except that Dijkstra's Algorithm computes the minimum distance with each edge e having a length len(e), whereas Highest Bottleneck Bandwidth computes the minimum width of any edge on the path, and for a vertex v, define widthto(v) to be the width of the widest path from s to v (say that widthto(s) = ∞).

Highest Bottleneck Bandwidth

```
1  function HBB(Graph, source):
2      for each vertex v in Graph:
3          width[v] := -infinity  ;
4          previous[v] := undefined ;
5      end for
6
7      width[source] := infinity ;
8      Q := the set of all nodes in Graph ;
9
10      while Q is not empty:
11          u := vertex in Q with largest width in width[] ;
12          remove u from Q ;
13          if width[u] = -infinity:
14              break ;
15          end if
16          for each neighbor v of u:
17        alt := max(width[v], min(width[u], width_between(u, v))) ;
18       if alt > width[v]: (u,v,a)
19                  width[v] := alt ;
20                  previous[v] := u ;
21                  decrease-key v in Q;
22              end if
23          end for
24      end while
25      return width;
26  endfunction
```

Fig 2.2: HBB algorithm [18]

2.7.2 Detect choke links:

The most important concept of the bottleneck finding algorithm is the detection of choke points [3]. Choke points are considered in case of switches that comprises of at least two edges, in other words bandwidth. Choke points or choke links concept cannot be applicable for the switch with single edges because there is nothing to compare for the packets to route to the next hop. Moreover, a single edge choke link will result in huge data loss. In practical, detection of choke links plays a significant role in improving a network's performance because with notifying, a user or an operator is aware of the reason where the transfer of packets is delayed.

The formal definition of choke link and choke point - Let us assume an end-to-end path from source S = S0 to destination D = Sn through switches S1,S2,... Sn−1. Link Li = (Si, Si+1) has available bandwidth BW i ($0 \leq i < n$). There, with the notations above, we define the set of choke links as:

$$\text{CHOKE L} = \{Lk \mid \exists j,\ 0 \leq j < n,\ k = \operatorname{argmin}_{0 \leq i \leq j} BWi\ \}$$

and the corresponding set of choke points (or choke switches) are

$$\text{CHOKE S} = \{Sk \mid Lk \in \text{CHOKE L},\ 0 \leq k < n\}$$

In the Result Analysis, we have shown some remarkable impact of choke links that can co-relate to runtime, throughput, latency and hop count to destination with the help of graphs.

CHAPTER 3

Topologies

In this chapter different topologies have been used to demonstrate the simulations. With the help of different topology size, we evaluated the performance of the algorithms and the networks. In 3.1, we have shown shortest path algorithms and in 3.2 we have demonstrated about bottlenecks.

3.1 SP Algorithm Applications:

In this chapter, we have demonstrated different types of topologies used to implement the respective simulations. In 3.1.1, 3.1.2 and 3.1.3 we have shown how the algorithms were implemented for respective applications. In 3.2, we have illustrated six different topologies for bottleneck bandwidth simulation with the help of the diagrams.

3.1.1 Link Failure Recovery:

In the figure 3.1, a random mesh network topology of 28 switches has been used. The switches and the nodes represent OpenFlow. They are decoupled in control plane and data plane. All the information and data of the switches remain in the controller which works as a forwarding tool.

The topology was set up based on Abilene core topology in Mininet OpenFlow network with 1 controller and 11 switches in Fig 3.2. Iperf was used as a testing tool to generate TCP data streams in their simulation [19].

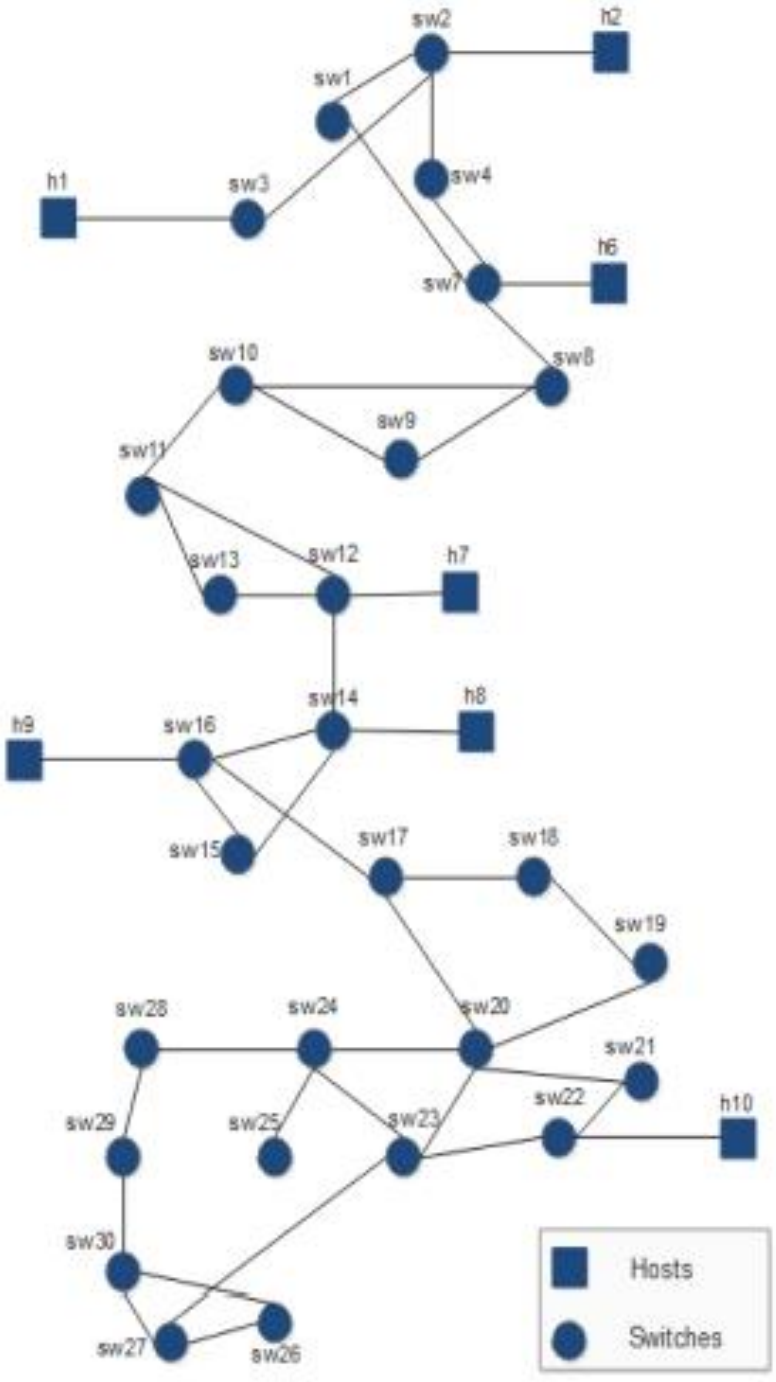

Fig 3.1: Topology of 28 switches [8]

3.1.2 Load Balancing:

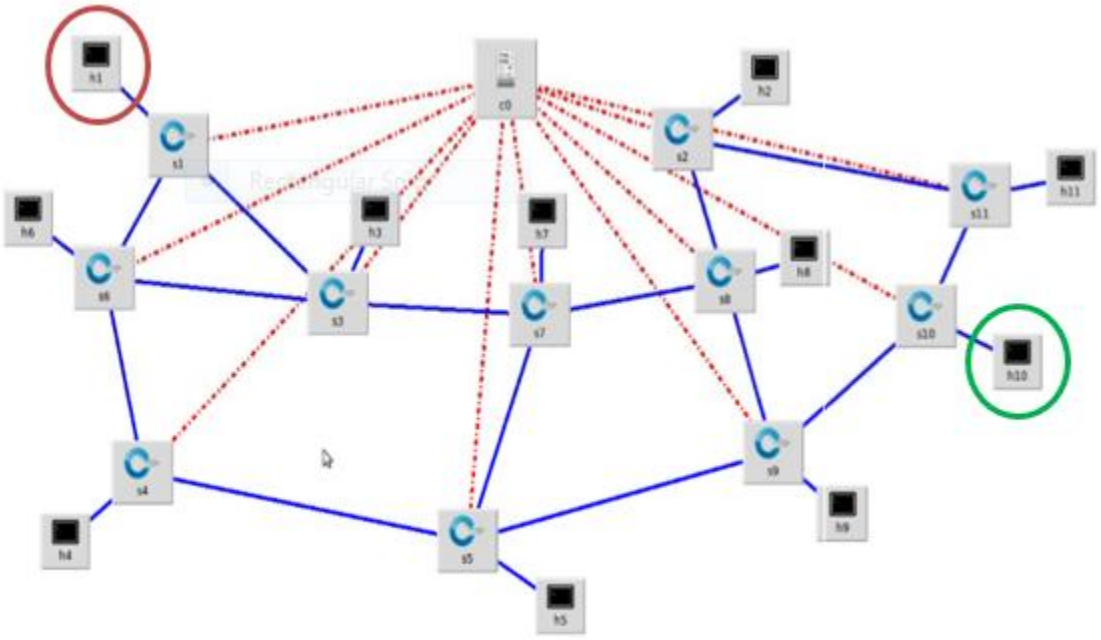

Fig 3.2: Abilene Topology for Load Balancing Demonstration [10]

3.1.3 Dynamic Programmability:

Mininet topology was set up as shown in figure below with 2 hosts, 8 switches and a controller Fig 3.3.

3.2 Bottleneck Bandwidth:

In order to make an SDN based simulation environment, we need links between the nodes, switches enabled by SDN, controller and a host. To reduce the hassle, here we used the local host address, which is 127.0.0.1. In our codes, edge weight represents bandwidth and nodes represent the switches. Switch-1 is considered as the source while the Switch-n (last switch) is considered as the destination. We created 6 different topologies for this simulation in order to evaluate the performance of the algorithm in the network with SDN environment. We used 6 different

topologies in order to check how well the algorithm's performance alters with changes in complexity of the topology. We have displayed all 6 different topology diagrams below.

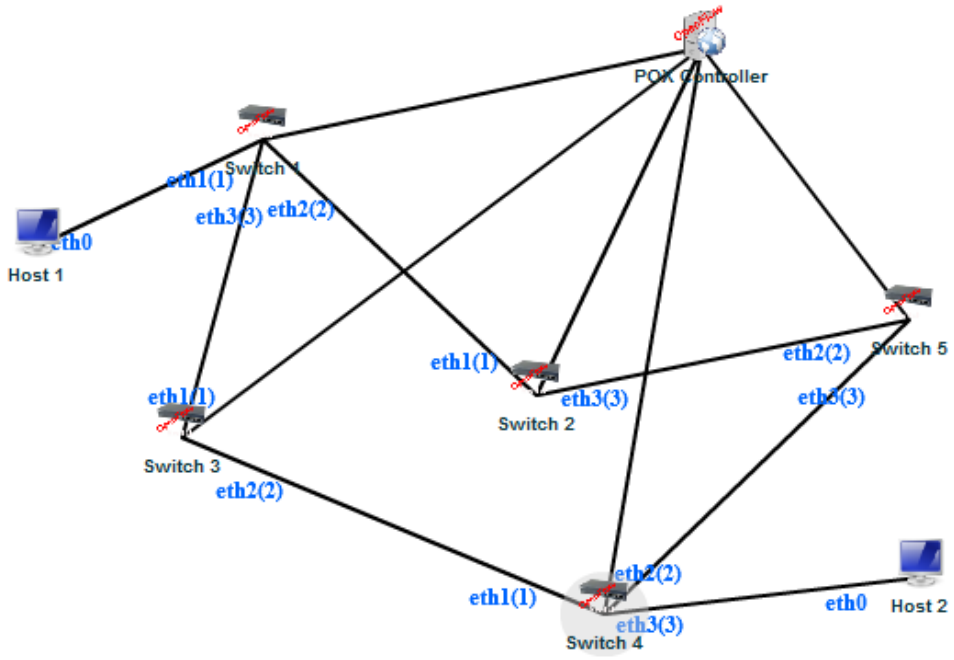

Fig 3.3: Mininet Topology for simulation [20]

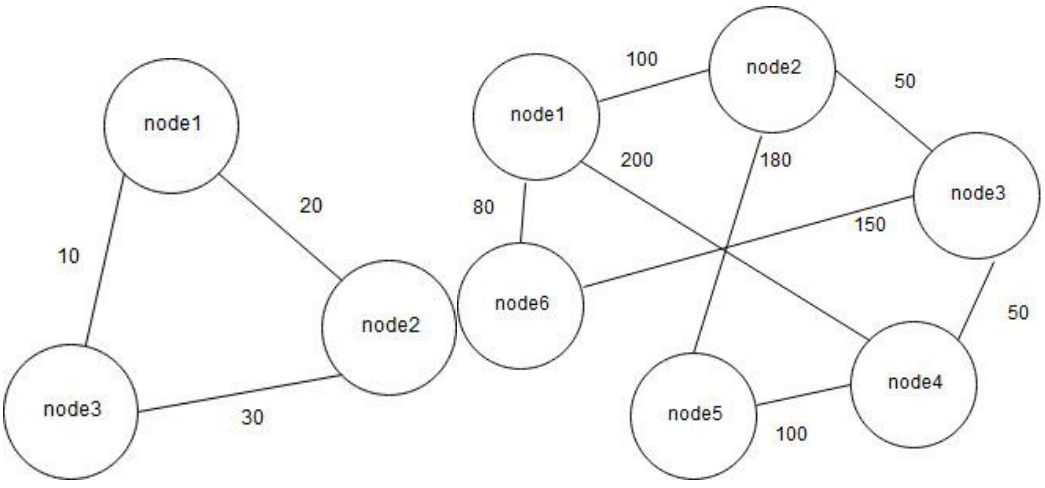

Fig 3.4: (a) Topology-3 (b) Topology-6

Fig 3.4 (a) is a topology with 3 nodes, 1 host (localhost) and a controller. Fig 3.4 (b) is a topology with 6 nodes, 1 host (localhost) and a controller.

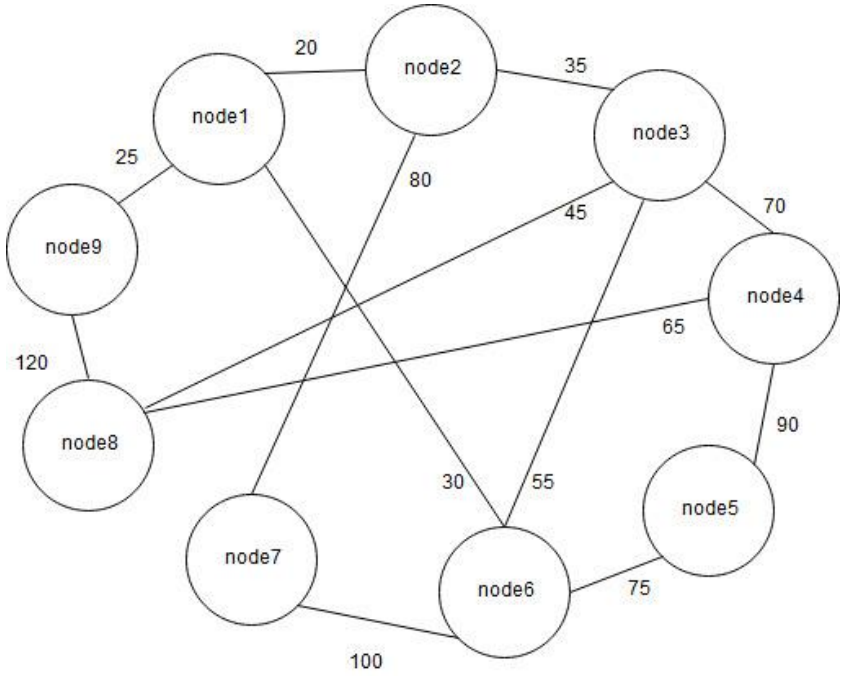

Fig 3.5 Topology – 9

Fig 3.5 is a topology with 9 nodes, 1 host (localhost) and a controller.

Fig 3.6 is a topology with 12 nodes, 1 host (localhost) and a controller. Fig 3.7 is a topology with 18 nodes, 1 host (localhost) and a controller. Fig 3.8 is a topology with 3 nodes, 1 host (localhost) and a controller.

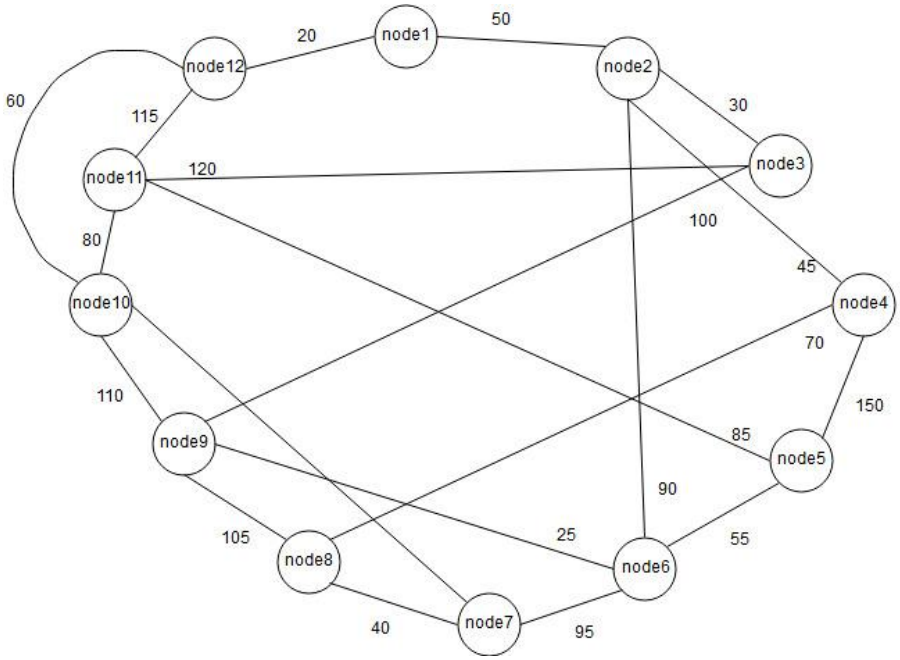

Fig 3.6: Topology-12

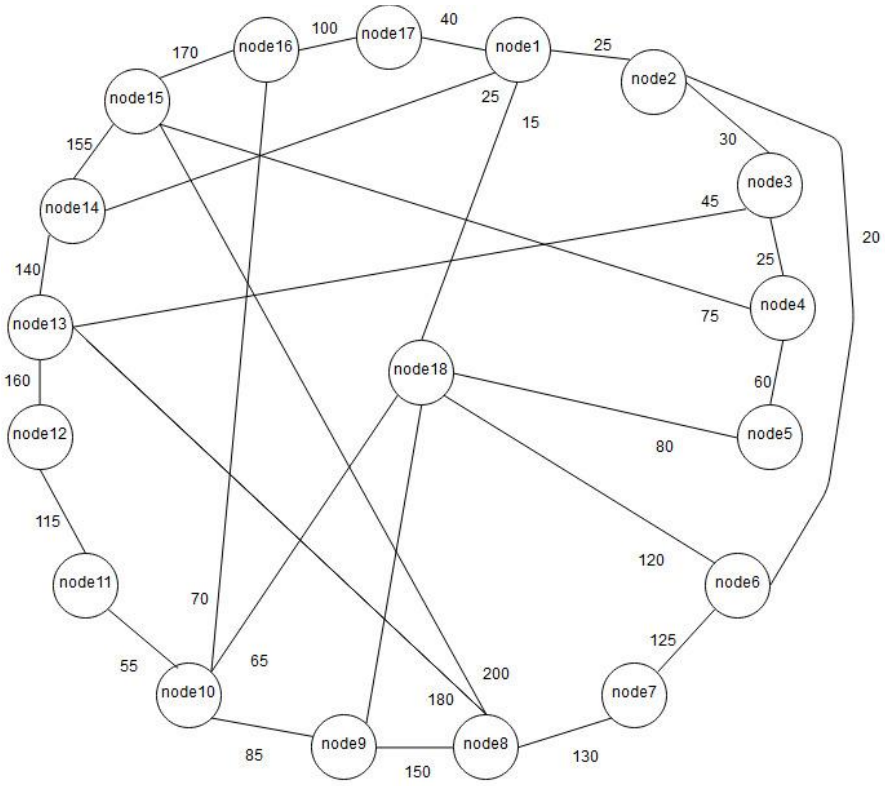

Fig 3.7: Topology-18

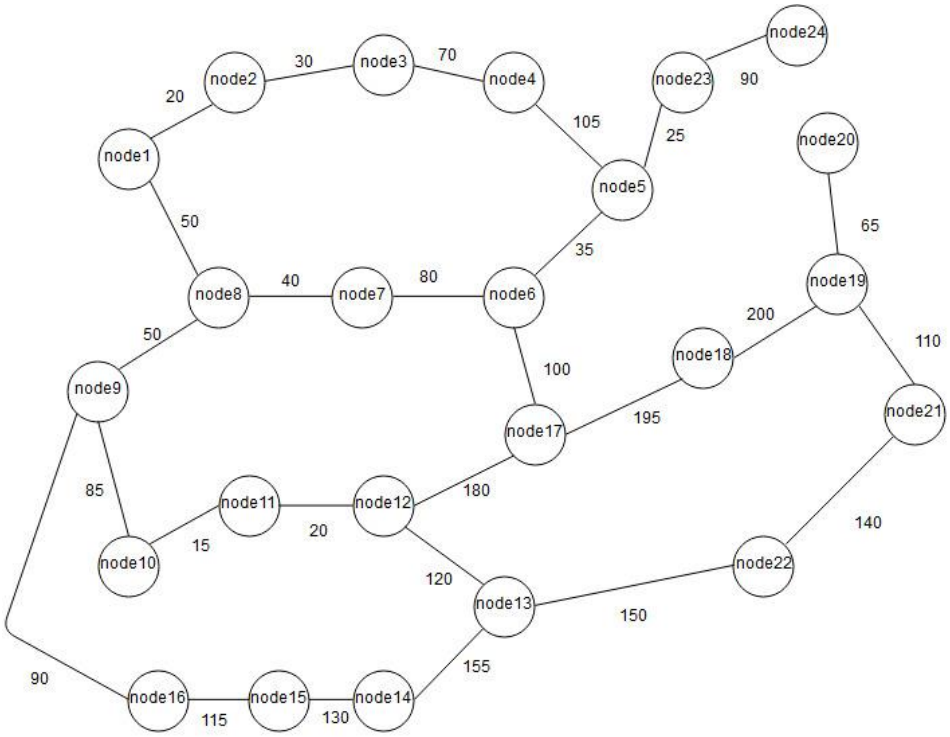

Fig 3.8: Topology-24

CHAPTER 4

Algorithm Implementation

In this chapter, we have demonstrated the algorithms implemented for the comprehensive evaluation and bottleneck finding algorithm. In case of comprehensive survey, the algorithms used were Bellman-Ford, Dijkstra and their modifications (Extended Dijkstra and Improved Extended Dijkstra). In case of Bottleneck finding, we used the Highest Bottleneck Bandwidth algorithm.

4.1 Shortest Path Algorithms:

We have shown how the other simulators implemented shortest path algorithms for simulation.

4.1.1 Link Failure Recovery:

Dijkstra and Bellman-Ford Algorithm has been implemented with the help of Mininet Emulator keeping the topology of 28 switches constant to make sure the simulation results are fairly judged. From the simulation, for each algorithm their shortest route to destination is measured. In case of Dijkstra algorithm, V[s] is described as an array containing all the node switches of the network. The distance associated with all these nodes is set by referring dist[s] to infinity. Dist[src] which is the starting point is the source being set to zero distance along with minimum distance(dist, Q) that computes the minimum distance adjacent node.

In case of Bellman-Ford, each node in the topology is considered such that D[u] contains the distances to infinity. The adjacent nodes are being accessed with D[u] + weight < D[v], that represents if distance D[u] of a single node added to its weight produces a lesser distance than its D[v] adjacent nodes distance, thus, D[v] = D[u]

+ weight would be updated. All the adjacent nodes would be processed including the nodes having minimum distance.

4.1.2 Load Balancing:

Node Weight & Edge Weight: [20]

According to Eq. (1) - The node weight nw[v] of v is defined as

$$nw[v] = \frac{\sum f \in Flow(v) Bits(f)}{Capacity(v)} \qquad (1)$$

Where Bits(f) stands for the number of f's bits processed by node v per second.

According to Eq. (2) - The edge weight ew[e] of e is defined as

$$nw[e] = \frac{\sum f \in Flow(e) Bits(f)}{Bandwidth(e)} \qquad (2)$$

Where Bits(f) stands for the number of f's bits passing through edge e per second.

The number of flow bits can be easily obtained by a node or passing through an edge of the "counters field" of the OpenFlow switches' flow tables. Moreover, the numerators in 1 and 2 equations are the unit of "bits" and denominators are units of "bits per second". Node weights and Edge Weights are unit of "seconds" and with the help of their values we can attain end-to-end latency from one end to the other end of the path. [10]

4.1.3 Dynamic Programmability:

POX controller is used in Mininet Simulator [21]. The python script is being run through the simulator. Links between the switches takes time to be detected first and then the Bellman-Ford algorithm is executed in the simulator as transmission of

packets is now available. Bellman - Ford algorithm finds the shortest path for the transmission. The packet starts getting transmitted one by one along the path [20].

4.2Highest Bottleneck Bandwidth:

The Highest Bottleneck Bandwidth (HBB) algorithm [18] has been implemented by modifying the Dijkstra algorithm where it considers the shortest edge path from source (s(p)) to destination (d(p)). In case of HBB, the algorithm first considers the narrowest path from each route to destination and finds the highest amongst them and chooses that route to destination.

CHAPTER 5

Result Analysis of Comprehensive Survey of Shortest Path Algorithms

In this chapter, we have analyzed the results for both comprehensive evaluations. From the results obtained in the comprehensive survey, we have evaluated the nature of the algorithms: Bellman-Ford, Dijkstra, Extended Dijkstra and Improved Extended Dijkstra. From the results we have tabulated the algorithms and categorized them to display which algorithm is better.

5.1 Load Balancing:

5.1.1 Extending Dijkstra:

A. Throughput Test on Larger Abilene Network: The number of nodes was increased by four more nodes to test for the effectiveness of the algorithms – DA, ED-SDN and mED-SDN. Throughput tests were carried out for the three approaches. [12]

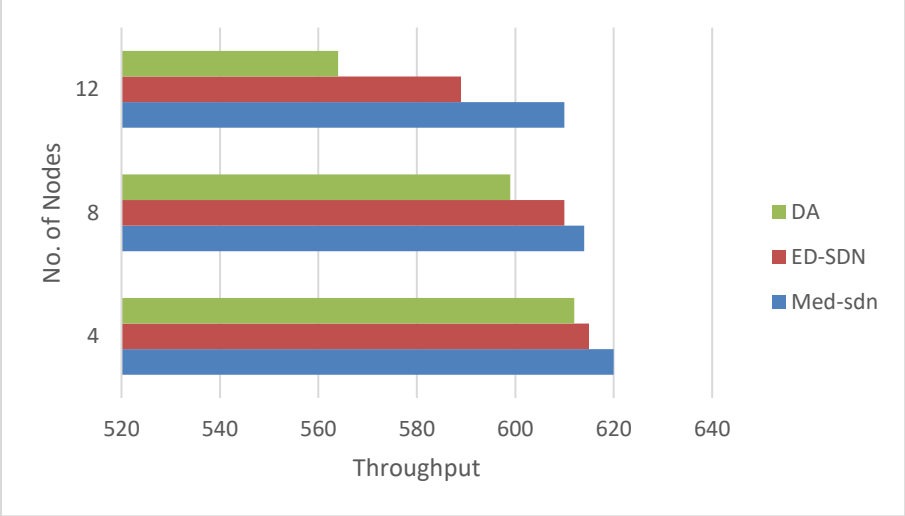

Fig 5.1: Throughput test

Nodes	mED-SDN	ED-SDN	DA
4	620 Mbps	615 Mbps	612 Mbps
8	614 Mbps	610 Mbps	599 Mbps
12	610 Mbps	589 Mbps	564 Mbps

Table 1: Throughput test results

B. Latency Test on Abilene Network Topology: The bandwidth of an edge and the capacity of a node were set randomly to be within the range [12].

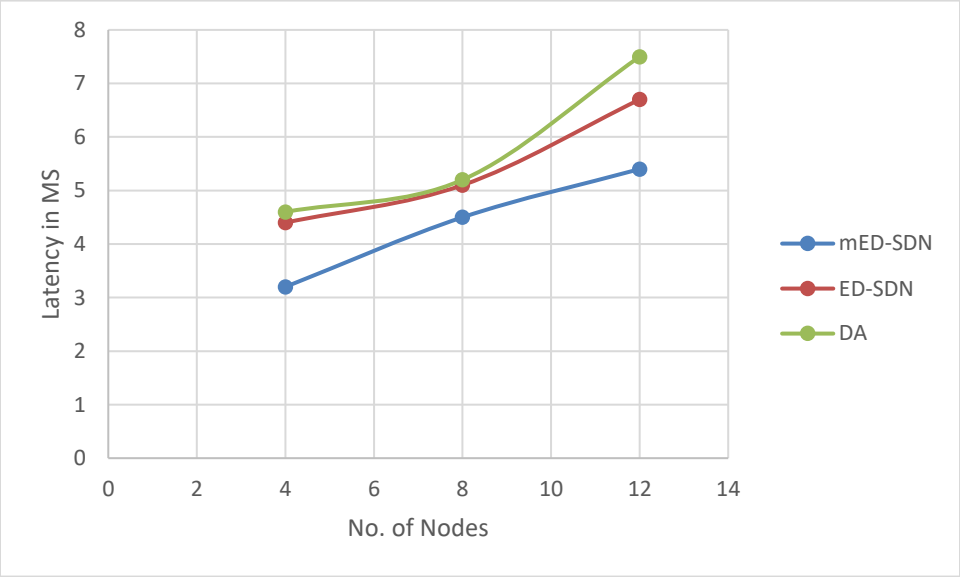

Fig 5.2: Latency test

We can notice from fig 5.2 that mED- SDN has less end-to-end latency than the original ED-SDN, and when the nodes increase, DA experiences significant degradation of latency, partly because of the increase in the number of hops a packet travels from the server node. [12]

Nodes	mED-SDN	ED-SDN	DA
4	3.2 ms	4.4 ms	4.6 ms
8	4.5 ms	5.1 ms	5.2 ms
12	5.4 ms	6.7 ms	7.5 ms

Table 2: Latency test results

5.2 Link Failure Recovery:

After successfully conducting the tests with both the algorithms we establish the comparative analysis of the results based on response time of Bellman Ford and Dijikstra algorithms implemented from single node failure to multiple node failure.

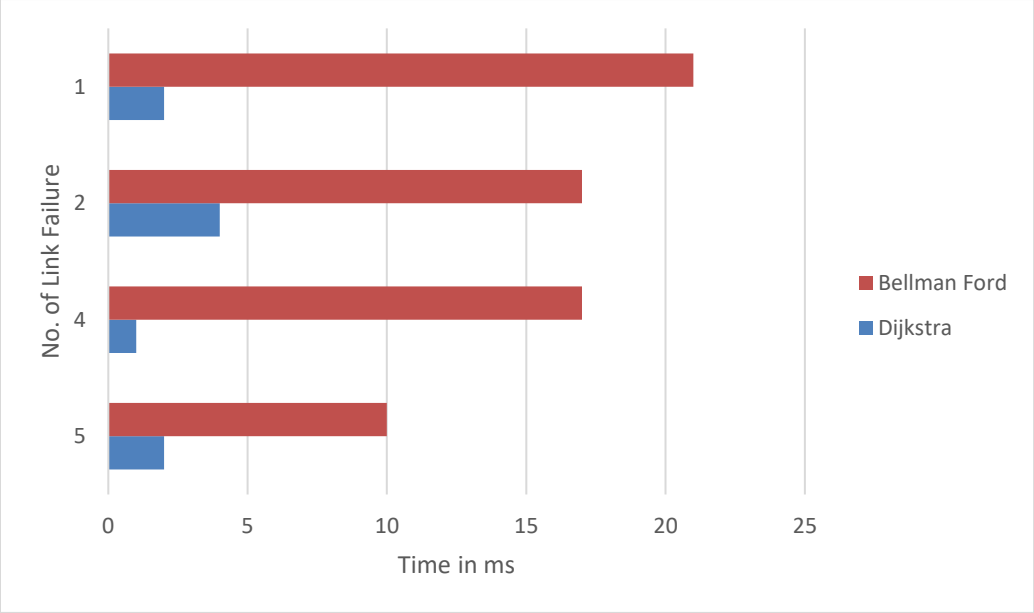

Fig 5.3: Comparative analysis

The response bar graph in Figure 5.3 gives a clear comparison picture showing the efficiency of operation between Bellman Ford and Dijkstra algorithm. The network is comprised of multiple links maintained between resulting source and destination

hosts. A reasonable amount of time is observed from the results obtained showing a reasonable amount of difference in time making Dijkstra more robust. [3]

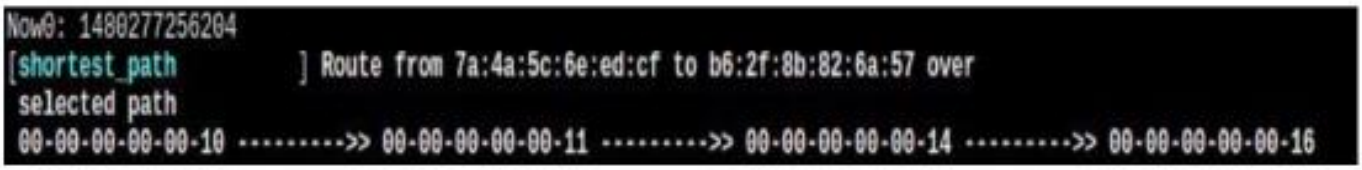

Fig 5.4: Shortest Route Calculation

Fig 5.4 provides information about the calculated least amount path from the source towards the destination.

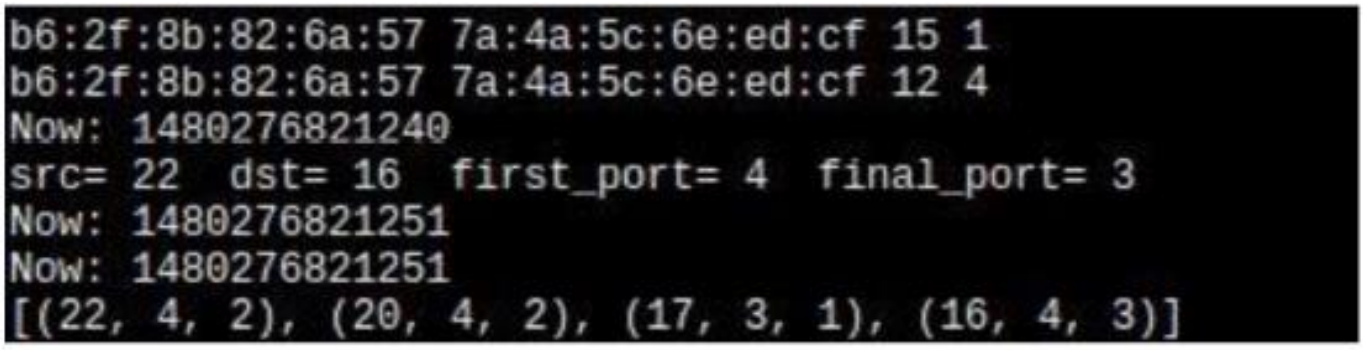

Fig 5.5: Optimum Minimum Path Calculated using Dijkstra

Fig 5.5 shows us that Bellman-Ford presents a single path until link failure occurred. This is when Dijkstra algorithm represented optimum minimum paths that can be used as an alternative in case of any failure. [8]

5.3 Dynamic Programmability of Controller using Bellman-Ford:

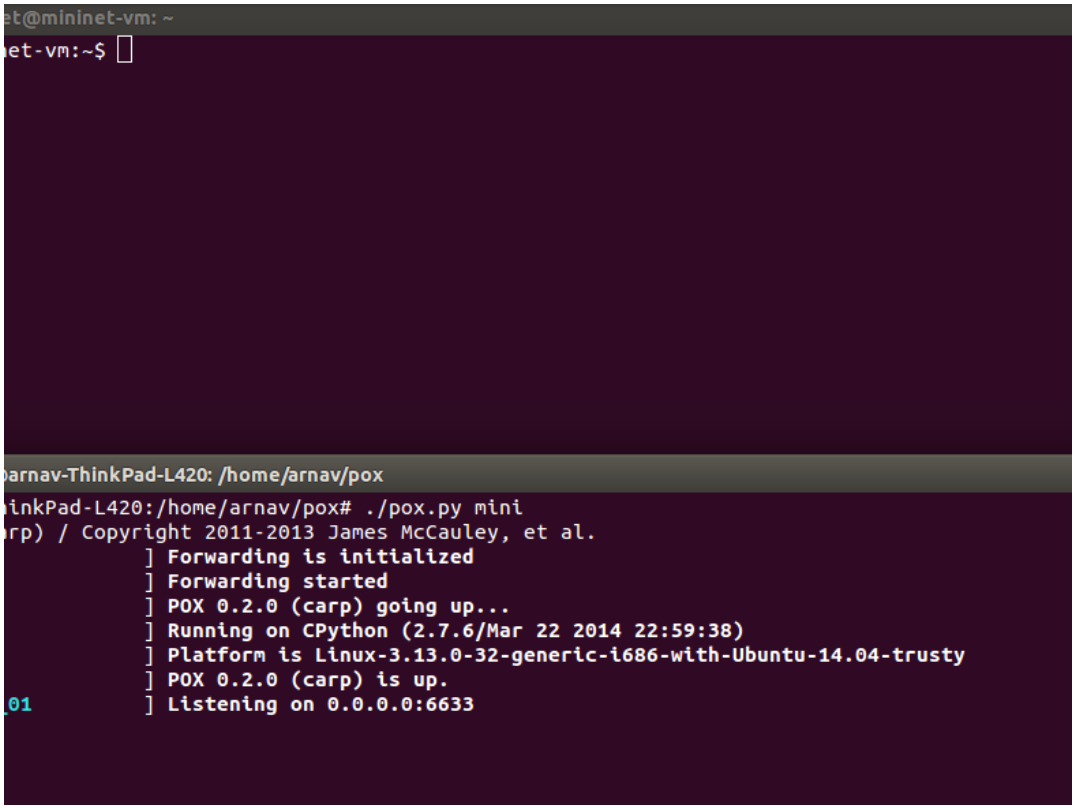

Fig 5.6 Packet Forwarding update

Transmission of packets is now possible once all the switches and links have been detected. In Fig above, Bellman - Ford algorithm finds the shortest path for the transmission the packet begins getting transmitted one by one along the path. At first ARP packet is sent to find the IP locations of every one of the system gadgets included and the stream table of the switches involved.

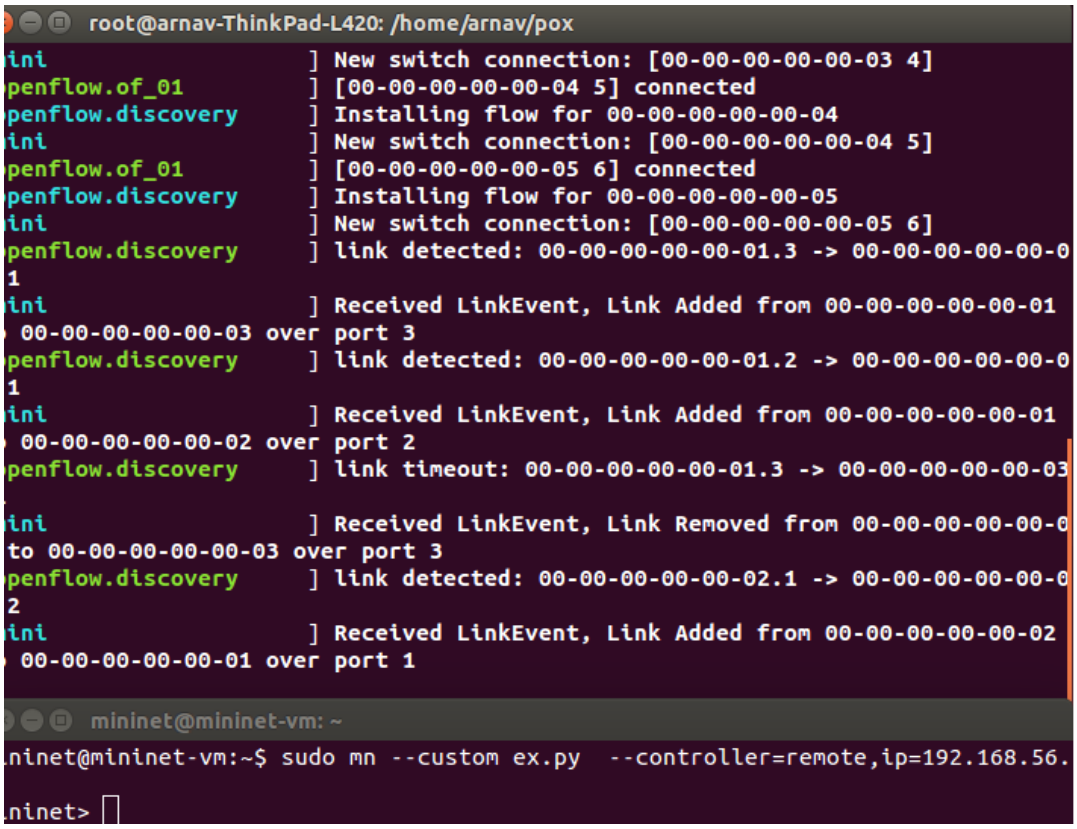

Fig 5.7: Adding of switches and links dynamically

In Fig 5.7, it is observed that whenever a switch is added to the network after running a python script called “ex.py” containing the specification of the network it generates a “Switch connection event”.

Evaluation:

Algorithms	Bellman-Ford Algorithm	Dijkstra		
Modifications & Extensions	–	General Dijkstra	Extended Dijkstra	Improved Extended Dijkstra
Optimal	Medium	Medium	High	High

Completeness	Complete	Complete	Complete	Complete
Delay Constraints	High	Medium	Low	Low
Applications	Single Link Failure Recovery, Dynamic SDN Controller	Single Link Failure Recovery	Load Balancing	Load Balancing, Multicasting

Table 3: Evaluation of the algorithms

In table 3, we have summarized the behavior of the two algorithms and their modifications. Dijkstra's contribution in the SDN application is higher compared to Bellman Ford along with their behaviors.

CHAPTER 6

Result Analysis of Highest Bottleneck Bandwidth

In this chapter, we have simulated and displayed our simulation results in form of graphs. With the simulation results, we evaluated the performance of the network with different topology size which helps to evaluate the nature of the algorithm with the change in complex topology. For evaluating the network performance, the parameters used were: runtime, throughput and latency. The parameters and number of choke points were graphically displayed with respect to the number of nodes.

6.1- Highest Bottleneck Bandwidth

6.1.1 - Simulation Settings

Parameters	**Setting**					
Bandwidth on Edges(mbps)	(10-30)	(50-200)	(20-120)	(20-155)	(15-200)	(15-200)
Number of Switches	3	6	9	12	18	24
Number of Edges	3	8	13	20	28	28
Delay Time (ms)	5	(10-30)	(5-40)	(5-50)	(10-65)	(5-50)
Testing Tool	Controller(Java Main class)					
Environment	Eclipse For Java (Oxygen)					

Table 4: Simulations Settings

To implement this concept, we used java programming language and to be specific the type of programming is known as Java Socket Programming [18]. For Socket, it must have two parts – IP address and Port number. Local host defines the IP address in our program (127.0.0.1) and the port number was 30000. Here, we defined the datagram packets along with the buffer size. Datagram packets can be called from the library directly.

With the help of HashMap, we dynamically input the text files of the topology into the code which is read by the buffer reader. Text file consists of the node connections, bandwidth and delay time. The main class of the java file is programmed to act as a controller here since any java code runs in the main class. As SDN environment requires protocol to connect the switches with the Controller, command line acted as the main protocol for the environment.

In order to get the updates from time to time, we implemented threading which is connected to the controller to send updates of all the switches and their connectivity status along with the routing table. If any switch's connection is lost, with the help of thread the message is being received by the controller which says "Switch is dead".

From the code, the parameters we generated were runtime, throughput, latency, choke points, and hop count to destination and then using the values, we generated graphs to justify the pattern. In order to infer if the network performance has improved or not, we randomized the algorithm. Randomizing the algorithm simply means, we stopped the HBB algorithm's functions which now gives freedom of transfer of packets from source to destination. Below in figure, the comparative results are displayed and explained.

The results are saved in LOG file which can be defined inside the java program using Bufferedwrite.

Once the execution starts, the Controller calls the switch for registering request. With the help of command line, every switch is called. Then the Controller receives update in form of the routing table.

6.1.2 Simulation Results in Graphs:

A. Runtime vs. Number of Nodes -

Fig 6.1: Runtime vs. Number of Nodes

In case of runtime, if we randomize the algorithm, significant changes is not observed. The graph is approximately positive linear and the runtime increases with size of the topology. With HBB algorithm executed, we observe that the runtime is relatively lower in case of small topology size but significantly rise till 18 nodes and then decreases at 24. The increase observed is due to number of checking of the bandwidth which consumes time. The decrease in runtime observed is due to the

orientation of the node connections in the topologies where the packets found high bandwidth paths in HBB unlike Randomized state.

B. Throughput vs. Number of Nodes -

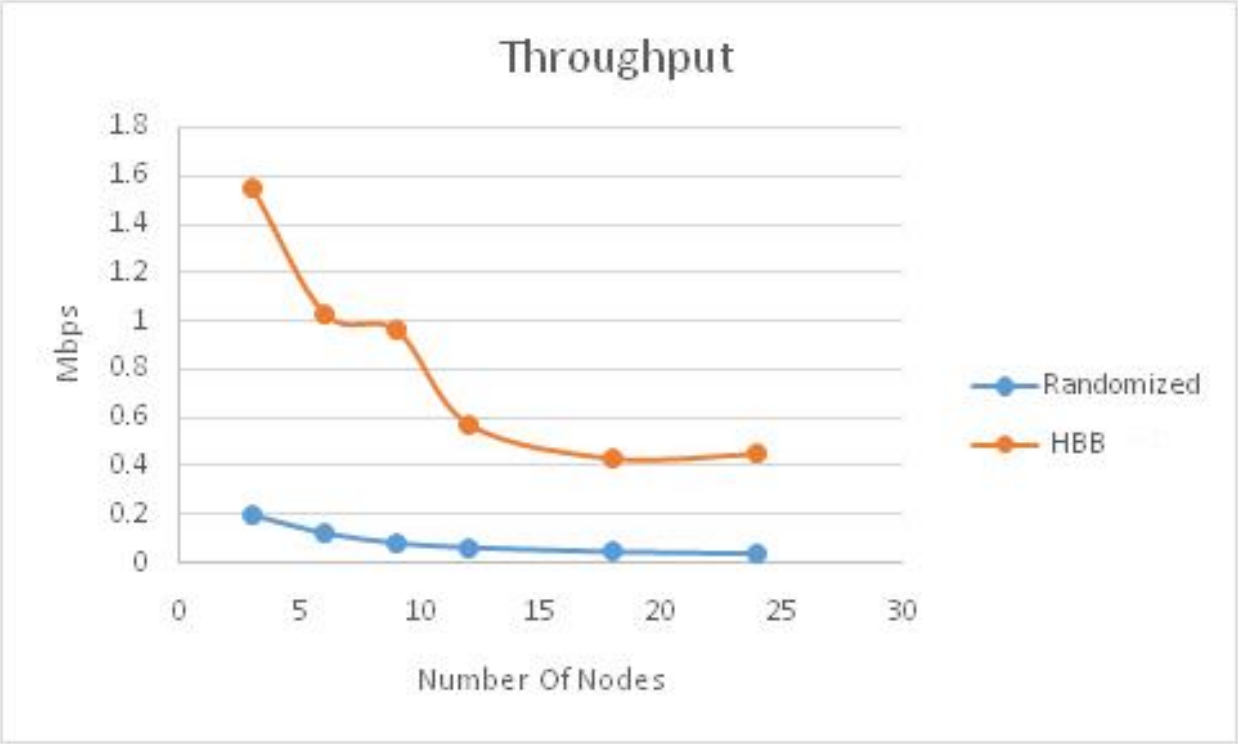

Fig 6.2: Throughput vs. Number of Nodes

In case of throughput, randomizing the algorithm significantly lowers network performance compared to HBB algorithm as observed from the above figure. The decrease in throughput with change of the size of the topology is quite negligible compared to the change observed in HBB algorithm. Yet, we can clearly find a significant difference in the throughput value in presence of the HBB algorithm. Hence network performance is guaranteed to improve.

A. Latency vs. Number Of Nodes -

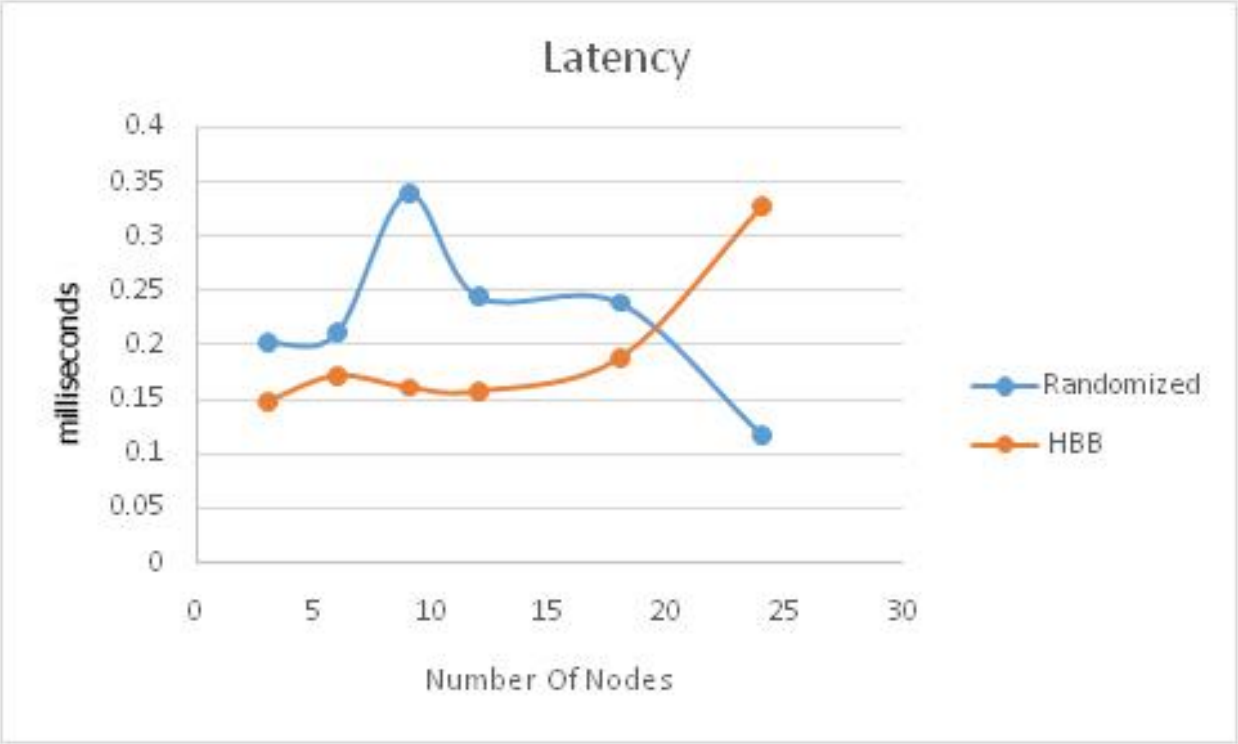

Fig 6.3: Latency vs. Number of Nodes

In case of comparing latency, with increase in topology size, the latency is higher in randomized state of the algorithm compared to the presence of HBB algorithm. This is due to the packets passing through the network topology with freedom. Hence, the chances of reaching the choke points are very high now compared to HBB algorithm. Therefore the rate of transfer of packets is significantly slowed down which effects the network performance. However, an exceptional case is observed when we simulated using 24 nodes. Latency decreases significantly in randomized due to the structural orientation of the edges of the graph. Most of the packets were able to route through the bandwidth edge at a lower hop count to destination which decreased the amount of time required to transfer the packets. Hence latency decreased. In case of HBB algorithm, latency increases remarkably because in order to avoid choke links, the hop count to destination increased. Therefore more time

was required to transfer the packets which resulted in a high latency. The result infers that the algorithm is also dependent on the structure of the network topology.

6.1.3 - Relating choke points or choke links:

A. Choke Points vs. Hop Count to Destination

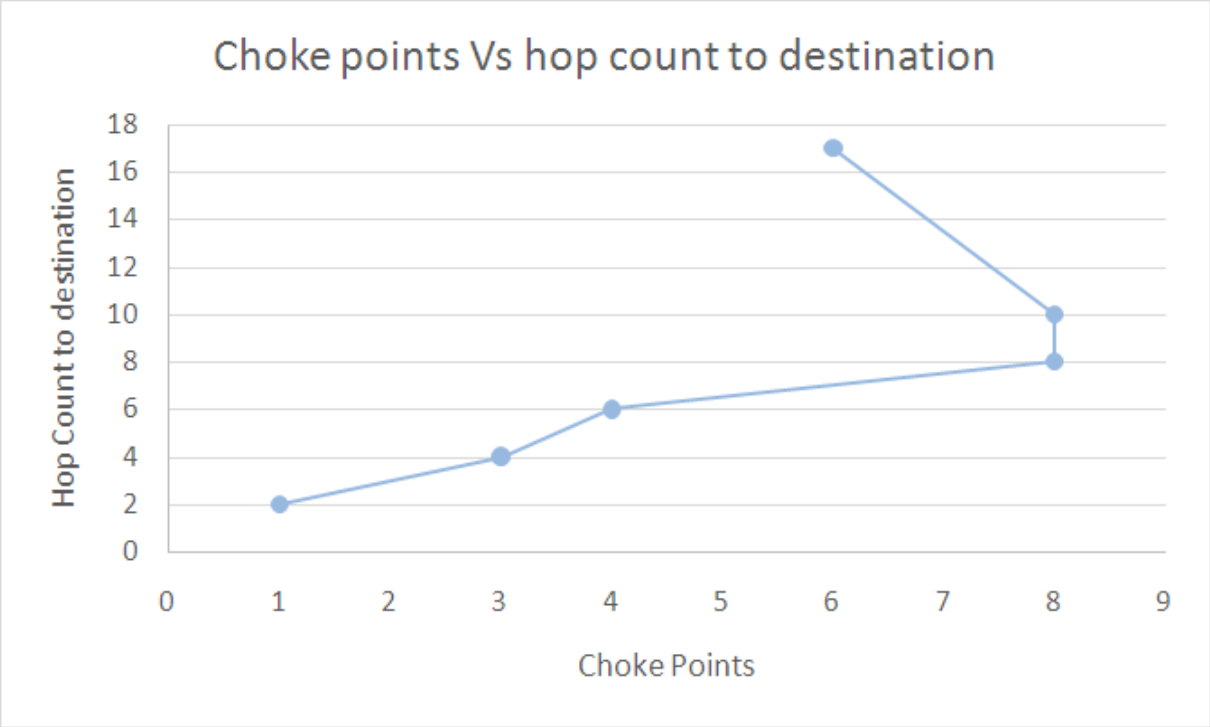

Fig 6.4: Choke Points vs. Hop Count to Destination

The graph of Number of Choke Points against Hop Count to Destination shows a linear relationship where hop count increases with increase in choke points from topology-3 to topology-18. In case topology-24, we observed before that the network topology structure was different from other topologies which caused the packets to take a path involving higher hop counts.

B. Choke Points vs. Throughput

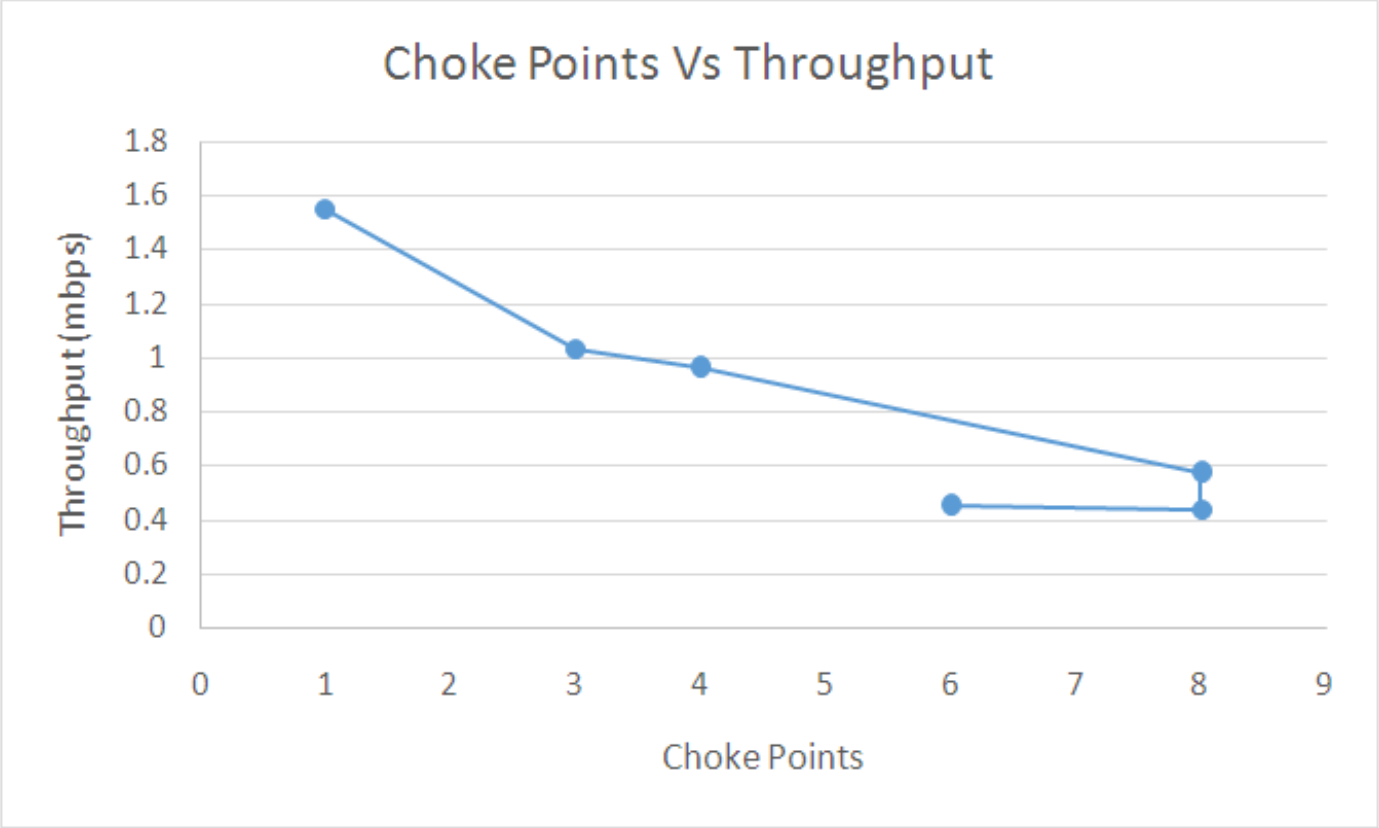

Fig 6.5: Choke Points vs. Throughput

From the graph above, we can infer that with increase in number of choke points, the network performance will be lower than before because now there are more edges with lower bandwidths. Hence the throughput decreases.

C. Choke Points vs. Runtime

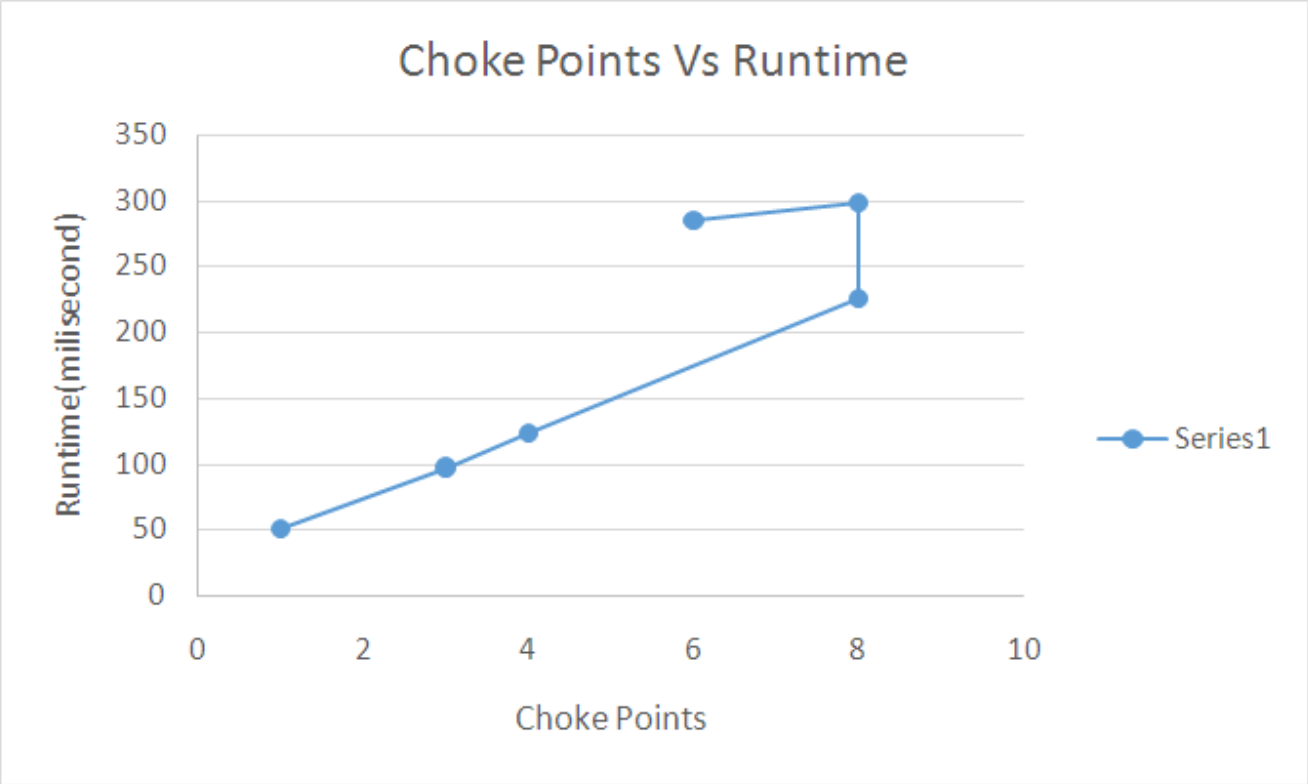

Fig 6.6: Choke Points vs. Runtime

With increase in choke points, the packets will have to change routes and the probability of the packets routing through longer paths is higher. Hence the runtime of the network will normally rise.

No clear relationship is observed from the graph in Fig 6.7 to find the relationship between the choke links and latency of the network.

D. Choke Points vs. Latency

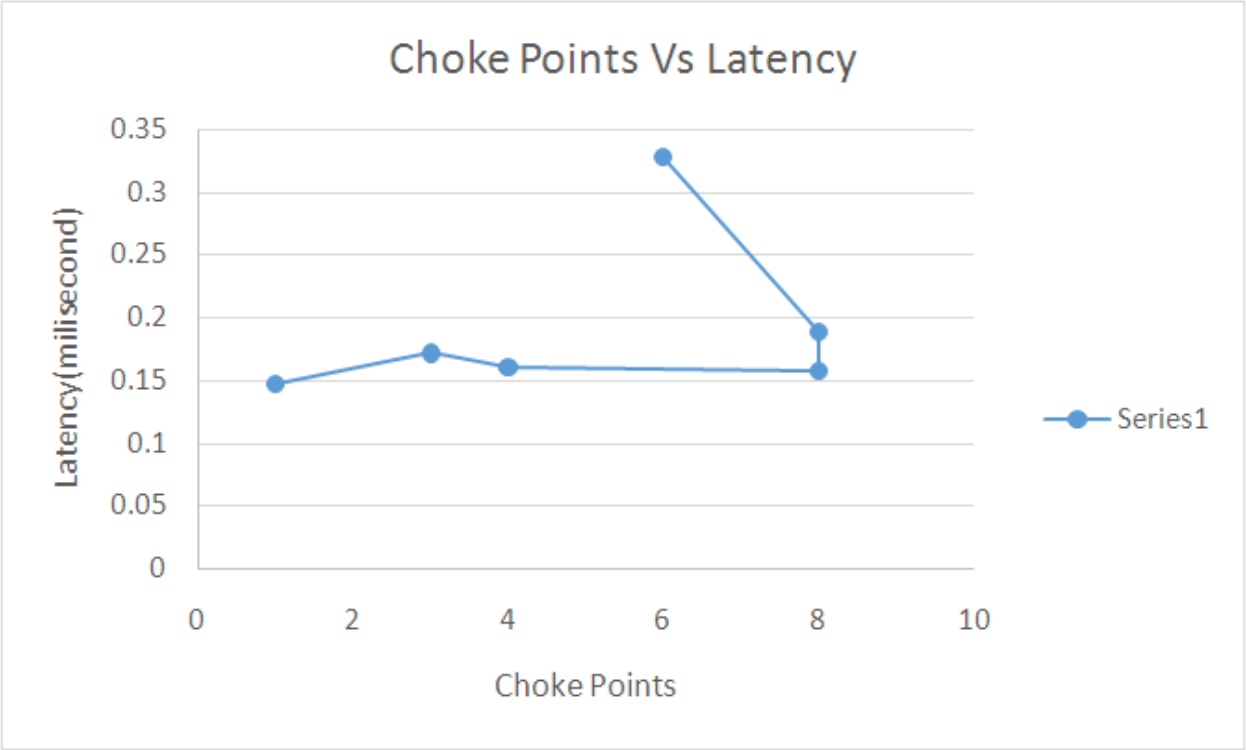

Fig 6.7: Choke Points vs. Latency

6.2 Running the HBB Algorithm:

The figure 6.8 is a demonstration of how we simulated our topology using the algorithm. In order to make the code run, first we run the eclipse (the environment we used for simulation). Next, with the help of the command line, we call all the switches we gave as input via text file. For instance, if the size of the topology comprised of 6 nodes, it means there are 6 switches practically. Hence, we call 6 times with 6 different command lines. The output is displayed within the command line, console and a log file.

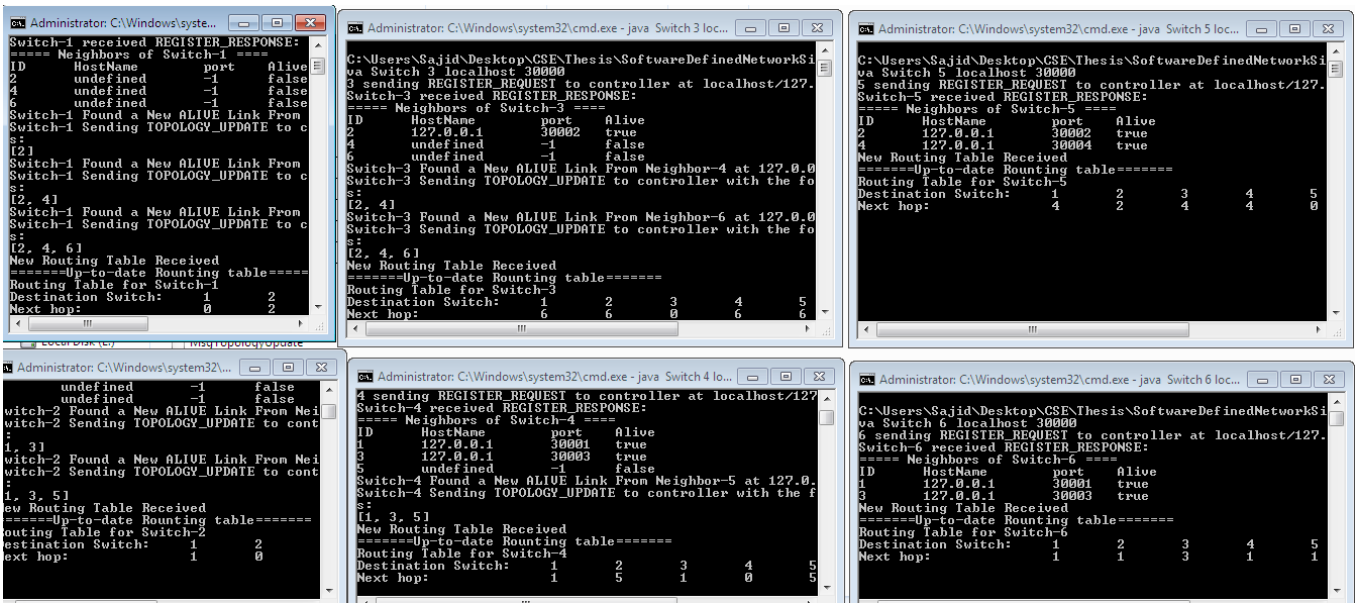

Fig 6.8: Command Lines for Topology-6

Figures 6.9 and 6.10 are a LOG file which has been created automatically after each simulation of the bottleneck bandwidth algorithm. Here, we have shown LOG file for Topology-6. First part describes the topology in form of 2D adjacency matrix. From second part, it describes the state of the switches after Switch-1 has been called. The process repeats until all the 6 Switches have been called. After the 6th switch, a routing table is generated for every switch as shown in 6.8 and we get to know if the switches are dead or alive.

```
2018-02-27T02:30:28.840: LOG file created.
2018-02-27T02:30:29.150: Adjacency Matrix of Switch Link Bandwidth
BW      1       2       3       4       5       6
1       Inf     100     0       200     0       80
2       100     Inf     50      0       180     0
3       0       50      Inf     50      0       150
4       200     0       50      Inf     100     0
5       0       180     0       100     Inf     0
6       80      0       150     0       0       Inf
2018-02-27T02:30:51.784: Received REGISTER_REQUEST From Switch-1
2018-02-27T02:30:51.873: Controller Sending REGISTER_RESPONSE To Switch-1
2018-02-27T02:30:51.873: ===== Neighbors of Switch-1 ====
2018-02-27T02:30:51.874: ID     HostName        Port    Alive
2018-02-27T02:30:51.874: 2      undefined       -1      false
2018-02-27T02:30:51.874: 4      undefined       -1      false
2018-02-27T02:30:51.874: 6      undefined       -1      false
2018-02-27T02:31:09.763: Received REGISTER_REQUEST From Switch-2
2018-02-27T02:31:09.764: Controller Sending REGISTER_RESPONSE To Switch-2
2018-02-27T02:31:09.766: ===== Neighbors of Switch-2 ====
2018-02-27T02:31:09.766: ID     HostName        Port    Alive
2018-02-27T02:31:09.766: 1      127.0.0.1       30001   true
2018-02-27T02:31:09.766: 3      undefined       -1      false
2018-02-27T02:31:09.766: 5      undefined       -1      false
2018-02-27T02:31:09.842: Controller Received TOPOLOGY_UPDATE from Switch ID: 1
2018-02-27T02:31:09.843: Waiting for all nodes registering
2018-02-27T02:31:26.397: Received REGISTER_REQUEST From Switch-3
2018-02-27T02:31:26.397: Controller Sending REGISTER_RESPONSE To Switch-3
2018-02-27T02:31:26.398: ===== Neighbors of Switch-3 ====
2018-02-27T02:31:26.398: ID     HostName        Port    Alive
2018-02-27T02:31:26.398: 2      127.0.0.1       30002   true
2018-02-27T02:31:26.398: 4      undefined       -1      false
2018-02-27T02:31:26.398: 6      undefined       -1      false
2018-02-27T02:31:26.433: Controller Received TOPOLOGY_UPDATE from Switch ID: 2
2018-02-27T02:31:26.433: Waiting for all nodes registering
2018-02-27T02:31:41.571: Received REGISTER_REQUEST From Switch-4
2018-02-27T02:31:41.572: Controller Sending REGISTER_RESPONSE To Switch-4
2018-02-27T02:31:41.572: ===== Neighbors of Switch-4 ====
2018-02-27T02:31:41.572: ID     HostName        Port    Alive
2018-02-27T02:31:41.572: 1      127.0.0.1       30001   true
2018-02-27T02:31:41.572: 3      127.0.0.1       30003   true
2018-02-27T02:31:41.572: 5      undefined       -1      false
2018-02-27T02:31:41.607: Controller Received TOPOLOGY_UPDATE from Switch ID: 1
2018-02-27T02:31:41.607: Waiting for all nodes registering
2018-02-27T02:31:41.616: Controller Received TOPOLOGY_UPDATE from Switch ID: 3
2018-02-27T02:31:41.616: Waiting for all nodes registering
2018-02-27T02:31:58.340: Received REGISTER_REQUEST From Switch-5
2018-02-27T02:31:58.340: Controller Sending REGISTER_RESPONSE To Switch-5
2018-02-27T02:31:58.340: ===== Neighbors of Switch-5 ====
2018-02-27T02:31:58.340: ID     HostName        Port    Alive
2018-02-27T02:31:58.340: 2      127.0.0.1       30002   true
2018-02-27T02:31:58.341: 4      127.0.0.1       30004   true
```

Fig 6.9: LOG File for Topology-6 (part-1)

```
2018-02-27T02:31:41.607: Waiting for all nodes registering
2018-02-27T02:31:41.616: Controller Received TOPOLOGY_UPDATE from Switch ID: 3
2018-02-27T02:31:41.616: Waiting for all nodes registering
2018-02-27T02:31:58.340: Received REGISTER_REQUEST From Switch-5
2018-02-27T02:31:58.340: Controller Sending REGISTER_RESPONSE To Switch-5
2018-02-27T02:31:58.340: ===== Neighbors of Switch-5 ====
2018-02-27T02:31:58.340: ID     HostName        Port    Alive
2018-02-27T02:31:58.340: 2      127.0.0.1       30002   true
2018-02-27T02:31:58.341: 4      127.0.0.1       30004   true
2018-02-27T02:31:58.373: Controller Received TOPOLOGY_UPDATE from Switch ID: 2
2018-02-27T02:31:58.374: Waiting for all nodes registering
2018-02-27T02:31:58.380: Controller Received TOPOLOGY_UPDATE from Switch ID: 4
2018-02-27T02:31:58.380: Waiting for all nodes registering
2018-02-27T02:32:12.893: Received REGISTER_REQUEST From Switch-6
2018-02-27T02:32:12.893: Controller Sending REGISTER_RESPONSE To Switch-6
2018-02-27T02:32:12.893: ===== Neighbors of Switch-6 ====
2018-02-27T02:32:12.893: ID     HostName        Port    Alive
2018-02-27T02:32:12.893: 1      127.0.0.1       30001   true
2018-02-27T02:32:12.893: 3      127.0.0.1       30003   true
2018-02-27T02:32:12.895: All Switches Registered
2018-02-27T02:32:12.896: The routing table for every node is as below:
2018-02-27T02:32:12.896: ===Up-to-date Routing Table for all Switches===
2018-02-27T02:32:12.896: Routing Table for Switch-1
Destination Switch:             1       2       3       4       5       6
Next hop:                       0       2       6       4       2       6

2018-02-27T02:32:12.896: Routing Table for Switch-2
Destination Switch:             1       2       3       4       5       6
Next hop:                       1       0       1       1       5       1

2018-02-27T02:32:12.897: Routing Table for Switch-3
Destination Switch:             1       2       3       4       5       6
Next hop:                       6       6       0       6       6       6

2018-02-27T02:32:12.897: Routing Table for Switch-4
Destination Switch:             1       2       3       4       5       6
Next hop:                       1       5       1       0       5       1

2018-02-27T02:32:12.898: Routing Table for Switch-5
Destination Switch:             1       2       3       4       5       6
Next hop:                       4       2       4       4       0       4

2018-02-27T02:32:12.898: Routing Table for Switch-6
Destination Switch:             1       2       3       4       5       6
Next hop:                       1       1       3       1       1       0

2018-02-27T02:32:13.001: Controller Received TOPOLOGY_UPDATE from Switch ID: 1
2018-02-27T02:32:13.001: Route table already updated
2018-02-27T02:32:13.003: Controller Received TOPOLOGY_UPDATE from Switch ID: 3
2018-02-27T02:32:13.003: Route table already updated
```

Fig 6.10: LOG File for Topology-6 (part-2)

CHAPTER 7

Conclusion

Many communication networks and applications strictly require and maintain QoS but also many devices cannot perform to their full capacity due insufficient bandwidth through which data is transferred. In this paper, we have shown some of the implementations of Bellman-Ford and Dijkstra algorithm in SDN applications along with their performance evaluation as well as simulate and display results to test network's performance using HBB algorithm. From the results analysis, we can conclude the effectiveness of the algorithm that contributes in the network in terms of throughput and runtime. Although the latency may vary, it also depends on the topology that we have defined but network performance is guaranteed to improve.

7.1 Future Plan:

We would like to make our algorithm work as an application or a tool to detect network bottlenecks and implement the application based on firewall [23]. Firewall system allows packets to route with more security and hence this will avoid data loss or hijack.

7.2 Challenges:

Since software-defined networking is a very new concept in computer networks, we had to thoroughly go through the functionalities in order to make this project work. While simulating an algorithm, the challenges we faced –

- With Java Socket Programming, it is possible to simulate basic network topologies but not flexible for large, complicated networks. Hence, a simulator build for networking is required for simulation.

- Packet loss rate calculation is not possible in java network programming because we have defined the datagram packets with the help of the java library. A special case where packet losses may happen is if there is a bug in our code.
- We needed to call every switch for this simulation so for large networks of 24 or 30 switches, so this method is not dynamic.
- The results have been generated and displayed with the help of a LOG file for every different topology.

REFERENCES

[1] Madkour, Amgad & Aref, Walid&Rehman, Faizan& Rahman, Abdur&Basalamah, Saleh. (2017). "A Survey of Shortest-Path Algorithms".

[2] J. W. Guck, A. Van Bemten, M. Reisslein and W. Kellerer, "Unicast QoS Routing Algorithms for SDN: A Comprehensive Survey and Performance Evaluation," in *IEEE Communications Surveys & Tutorials*, vol. 20, no. 1, pp. 388-415, Firstquarter 2018.

[3] Network Bottleneck. (n.d.). Retrieved from https://www.techopedia.com/definition/24819/network-bottleneck

[4] M. Cascado, N.Foster andA.Guha,"Abstractions for Software-Defined Networks", Communications of the ACM,2014, Vol. 57,Issue 10, pp. 86-95.

[5] Hu, Ningning& L. Li, Erran& Mao, Zhuoqing&Steenkiste, Peter & Wang, Jia. (2004)," Locating Internet Bottlenecks: Algorithms, Measurements, and Implications", 34. 41-54. 10.1145/1015467.1015474.

[6] S. Tomovic, I. Radusinovic and N. Prasad, "Performance comparison of QoS routing algorithms applicable to large-scale SDN networks," IEEE EUROCON 2015 - International Conference on Computer as a Tool (EUROCON), Salamanca, 2015, pp. 1-6.

[7] Traffic Engineering. (n.d.). Retieved from http://searchtelecom.techtarget.com/definition/traffic-engineering

[8] S. Waleed, M. Faizan, M. Iqbal and M. I. Anis, "Demonstration of single link failure recovery using Bellman Ford and Dijikstra algorithm in SDN," 2017 International Conference on Innovations in Electrical Engineering and Computational Technologies (ICIEECT), Karachi, 2017, pp. 1-4.

[9] Ku'zniar, Maciej&Perevs'ini, Peter &Vasi'c, Nedeljko&Canini, Marco &Kostic, Dejan. (2013),"Automatic Failure Recovery for Software-defined Networks". 159-160. 10.1145/2491185.2491218.

[10] Kairanbay, Magzhan& Mat Jani, Hajar. (2013),"A Review and Evaluations of Shortest Path Algorithms. International Journal of Scientific & Technology Research", 2. 99-104

[11] J. R. Jiang, H. W. Huang, J. H. Liao and S. Y. Chen, "Extending Dijkstra's shortest path algorithm for software defined networking," *The 16th Asia-Pacific Network Operations and Management Symposium*, Hsinchu, 2014, pp. 1-4.

[12] Abdulaziz, Abdul-hafiz &Adedokun, Emmanuel & Man-Yahya, Sani. (2017),"Improved Extended Dijkstra's Algorithm for Software Defined Networks", International Journal of Applied Information Systems, 12. 22-26. 10.5120/ijais2017451714

[13] What is OpenFlow (n.d.). Retieved from https://www.sdxcentral.com/sdn/definitions/what-is-openflow/

[14] M. Saxena and R. Kumar, "A recent trends in software defined networking (SDN) security," *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, New Delhi, 2016, pp. 851-855.

[15] Ananta, M.T. & Jiang, Jehn-Ruey& Muslim, M.A.. (2014),"Multicasting with the extended dijkstra's shortest path algorithm for software defined networking", 9. 21017-21030.

[16] POX API (n.d.). Retrieved from https://github.com/noxrepo/pox

[17] M.A. Othman, H.A. Sulaiman, M.M. Ismail, M.H. Misran, M.A.B.M. Said and R.A. Ramlee, "Analysis of Least-Cost Routing using Bellman Ford and

Dijkstra Algorithms in Wireless Routing Network", International Journal of Advancements in Computing Technology, Jun2013, Vol. 5 Issue 10, pp. 109.

[18] Java Socket programming. (n.d.). Retrieved from https://www.javatpoint.com/socket-programming

[19] Gao, Jim. "Widest Path Algorithm (Variation of Dijkstra)." *Jims Programming* , 11 Aug. 2015, jimgao.tk/wordpress/?p=26

[20] Iperf Website, http://iperf.fr/, last accessed on June 2014.

[21] Shivendu, Arnav &Dhakal, Dependra& Sharma, Diwas. (2015)," Emulation of Shortest Path Algorithm in Software Defined Networking Environment", International Journal of Computer Applications. 116. 47-49. 10.5120/20304-2344.

[22] Mininet Overview (n.d.). Retrieved from http://mininet.org/overview/

[23] Firewall (n.d.). Retrieved from http://searchsecurity.techtarget.com/definition/firewall