A Constraint Based K-Shortest Path Searching Algorithm for Software Defined Networking

Siddhant Ray
School Of Electronics Engineering
Vellore Institute of Technology, Vellore
Vellore, Tamil Nadu, India
siddhant.r98@gmail.com

***Abstract*—Software Defined Networking (SDN) is a concept in the area of computer networks in which the control plane and data plane of traditional computer networks are separated as opposed to the mechanism in conventional routers and switches. SDN aims to provide a central control mechanism in the network through a controller known as the SDN Controller.[1] The Controller then makes use of various southbound Application Programming Interfaces (APIs) to connect to the physical switches located on the network and pass on the control information, which used to program the data plane. SDN Controller also exposes several northbound APIs to connect to the applications which can leverage the controller to orchestrate the network. The controller used with regard to this paper is the Open Network Operating System (ONOS), on which the algorithm in question is to be deployed. ONOS provides several APIs which is leveraged to connect the application to the network devices. The typical network path between any two endpoints is a shortest path fulfilling a set of constraints. The algorithm developed here is for optimal K-Shortest path searching in a given network satisfying specified constraints, controlled by ONOS.**

***Index Terms*—ONOS, OpenFlow, K-Shortest Paths, Yens Algorithm, Constrained Shortest Path First (CSPF), white edges, white vertices**

I. INTRODUCTION

The ONOS SDN controller uses OpenFlow protocol on the southbound to communicate with the switches for managing the data plane. The application on the SDN Controller calculates the network paths and translates the network paths as series of flow entries which is written to control each switch and its actions. The protocol defines the flows in a particular required format, which communicates the necessary information to the southbound devices. Each flow is required to be associated with a select mechanism which determines how to process the packet and an action mechanism which determines what to do. [2] As a part of this study MPLS Layer 2 VPNs were created to provide E2E service. OpenFlow supports Multiprotocol Label Switched (MPLS) protocol. The MPLS is a defined as a layer 2.5 protocol as it uses the best of both layer 2 and layer 3 forwarding techniques. The MPLS protocol dispenses with the need of IP packet forwarding and the huge sizes of IP routing and forwarding tables. Instead it uses a tag swapping concept which can be efficiently implemented in the switching hardware. The VPN services were defined by configuring an end to end single segment PsuedoWires. The SDN application is called on the first packet, at that point a PseudoWire (PW) mapped to MPLS Server Layer Tunnel is computed. After this, a label stack is pushed on the switches, which determines the routing mechanism of the packet. The actions of pushing and popping the labels are written within the flows of the protocol and the OpenFlow protocol maintains flow tables to effectively manage the routing. Each flow consists of a forwarding action on the packet and thus, the need for IP forwarding is removed till it reaches its destination. The sources and the destination are both defined as PW edges and ARP is only used on the PW edges, and never on the intermediate ingress and egress ports of the nodes. The efficiency of the control plane through ONOS is significantly greater as it has a complete view of the network and OpenFlow provides the necessary hooks where custom applications can be plugged in. A key factor required in MPLS networks is the presence of multiple paths between the network nodes. This is done to speed up the fault correction process in the network if at any point; a link in the current routing path fails. A point of failure in the network is defined as a Point of Local Failure. To accommodate these faults, several backup paths are calculated to ensure quick switching in the case of localized failure. The standard criterion for any path selected in a network is for it to have the lowest possible cost of routing. To determine the various paths for the MPLS routing protocol, an efficient path searching algorithm needs to be developed within the ONOS framework. The objective of this paper mainly lies in devising an efficient path searching algorithm for the MPLS routing. It aims to find a mechanism to find the first K shortest paths in the network which can be used for routing. The shortest path may be defined so on the basis of any number and type of constraints. The objective further lies to make the algorithm robust to several points of failure in the network factoring in the network constraints. The analysis will be carried out by formulating the algorithm based on a type of constraint and then further, testing it on various network topologies in order to determine the real time feasibility in terms of accuracy and output time.

A. Abbreviations and Acronyms

During the entire course of the paper, several abbreviations might be use repeatedly. A comprehensive list of the major abbreviations used is hereby provided.

- SDN Software Defined Networking
- ONOS- Open Network Operating System
- ONF- Open Networking Foundation
- OSPF - Open Shortest Path First
- CSPF - Constrained Shortest Path First
- MPLS- Multiprotocol Label Switching
- LSP- Label Switched Path
- KSP- K-Shortest Paths
- IP- Internet Protocol

II. BACKGROUND KNOWLEDGE OF SDN

A. *Architecture of Software Defined Networking*

Software Defined Networking as defined by Open Networking Foundation (ONF) is an architecture that is dynamic, manageable, cost-effective, and adaptable. This architecture decouples the network control and forwarding functions enabling the network control to become directly programmable and the underlying infrastructure to be abstracted for applications and network services. [3]

The SDN architecture can be viewed as a three layered architecture as depicted in the Figure 1:

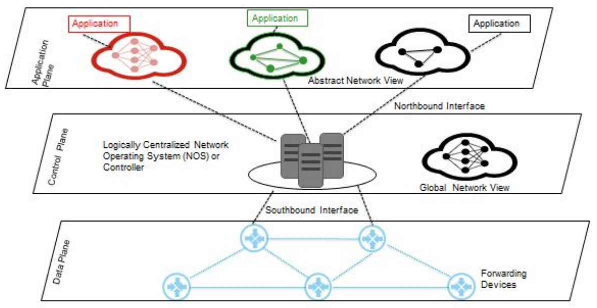

Fig. 1. Figure 1: Three Layered Architecture:

Data Plane: The lower most layer is the Data Plane. This stratum consists of forwarding devices in the form of switches and routers. These devices can be either physical or virtual. The physical switches are implemented using hardware. Most of the networking vendors today support e.g. Juniper and Cisco have support SDN protocols as a part their merchant switches. These switches are responsible for forwarding, modifying and dropping packets. It should also be mentioned that there are several implementations of the data plane in software. One of the most popular implementations of the software switch, which runs on common operating systems such Linux, is known as the Open vSwitch (OvS package) which has been extensively used during this study. The policies for applying the rules on the packets are configured by the control plane over South Bound Interface (SBI). Thus the control plane can control data planes processing and forwarding capabilities leveraging the SBI. The OpenFlow protocol is a foundational element for building SDN solutions and was defined by ONF. It happens to be one of the protocols of choice for the controller to control the behaviour of the forwarding plane. However it is not the only protocol that this supported over this interface. There are many other protocols like NETCONF, OVSDB, etc. which can be leveraged by the controller to orchestrate the forwarding layer. However OpenFlow is more than a secure control protocol running over TCP using Transport Layer Security (TLS) between the Forwarding and the Control Plane. It also defines the packet forwarding tables, through a mechanism of adding, modifying and removing packet matching rules and actions which are implemented in the dataplane. These rules are defined for entities which are termed as Flows, classified by the tuples in the packet header. A flow is defined as a set of packet field values acting as a match (filter) criterion and a set of actions (instructions). Thus using this architecture the control plane is able populate the flow table rules in the data plane to enable wirespeed forwarding. The packets which are unmatched are forwarded to the controller so that the controller can take the decision and populate the appropriated flow control rules to the dataplane.

Control Plane: The Control Plane should be viewed as the Networking Operating System (NOS). It is primarily responsible for the following

- Network Topology
- State Information
- Notification and Device(Switch) Management
- Abstraction of Data Plane
- Exposes NBI to the Application Layer

The control plane has been main objective of this study. The control plane used for this study is the ONOS.

The Northbound Interfaces (NBIs) are defined between the control plane and the application plane. Using NBIs, applications can exploit the abstract network views provided by the Control Plane to express network behaviour and requirements, and facilitate automation, innovation and management of SDN networks. The ONF and other standards bodies are trying to define the standard NBIs and a common information model.

Application Layer: The Application Layer consists of business application that consume Consumer Network APIs for Business Intelligence and Network Optimization.

This new architecture which is a deviation from the traditional IP networks by removing the network control functions from the underlying routers and switches, thereby promoting the logical centralization of network control and introduces the network programmability. Thus it promotes the concept as defined by ONF where network is centrally managed using a SDN Controller and programmatically configured by open standards based interfaces. This programmable aspect of the network opens possibility of developing applications like Security Applications and Monitoring Business Applications which leads to better monetization of the infrastructure.

B. Control Plane : SDN Controller

The SDN Controller is responsible for multiple functions. To facilitate the agile manipulation of the network by applications, it is imperative that is able to provide a logical abstracted view of the network. [4] A typical SDN Controller e.g. ONOS supports the following functions:

- Topology Service: Retrieve Topology, Node, Link Edge-Point details so that network connectivity map is available to the applications
- Connectivity Service: Request and Retrieve P2P, P2MP, Connectivity. There may be multiple forwarding technologies can be supported. For this study the P2P connectivity using PseudoWire (PW) over an MPLS tunnels was considered.
- Notification Service: Subscription and filtering / Autonomous event notification. This will ensure that the SDN Controller is real time view of the state of the network switches so that it can react in real time in case of failures or changes in the forwarding layers.
- Path Computation Service: Request for Computation and Optimization of paths so that the end to end network path. This was the focus area of the study.
- Network Service: Create, update, and delete network topologies. The Network Path that was calculated was programmed to the devices using the Network Services

One of the main functions of the control plane is to dynamically compute the end to end path between two end points in the network. For this study the type of service chosen was a Layer 2 VPN. The path computation algorithm of the Controller was augmented to calculate the constraint based shortest path between the two service end points. The constraints can be any service intent (parameters like bandwidth etc.) as depicted in figure 2.

The SDN architecture provides an additional benefit. The centralization of path computation enables customization of the control routing policies and algorithms from a single location in network. This is different from the traditional networks where the path computation is distributed on the network elements. Thus we see that the SDN controller enables the functionality of Path Computation Element (PCE) for the network it controls. The network applications can enable the calculation of customisable routing plans based on the

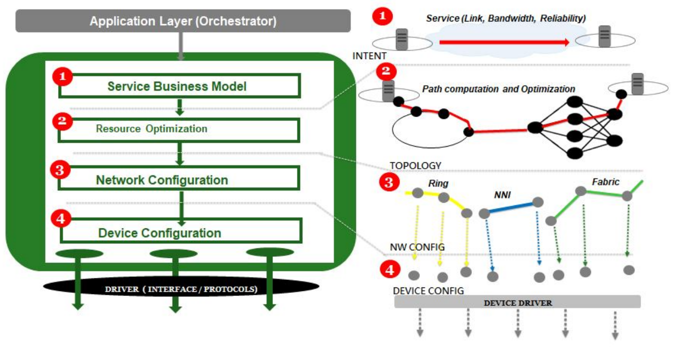

Fig. 2. Figure 2: Functionality of the SDN Controller

constraints. Any network can invoke the path computation algorithm through the North Bound API to pass the parameters and constraints. Applying the constraints, the SDN controller is capable of determining and finding a suitable route for conveying data between a source and a destination. The centralized approach is very efficient when multiple paths are being simlutaneoulsy computed as resource contention is minimised when using a distributed approach.

C. ONOS Architecture:

The ONS software is written in Java and provides a distributed SDN applications platform atop Apache Karaf OSGi container [3]. ONOS can run as a distributed system across multiple servers, allowing it to use the CPU and memory resources of multiple servers while providing fault tolerance in the face of server failure and potentially supporting live/rolling upgrades of hardware and software without interrupting network traffic. Thus is essentially provides a platform to build network applications that can be deployed in an Operators Network. The SDN applications are written are written in Java as bundles that are loaded into the Karaf OSGi container.

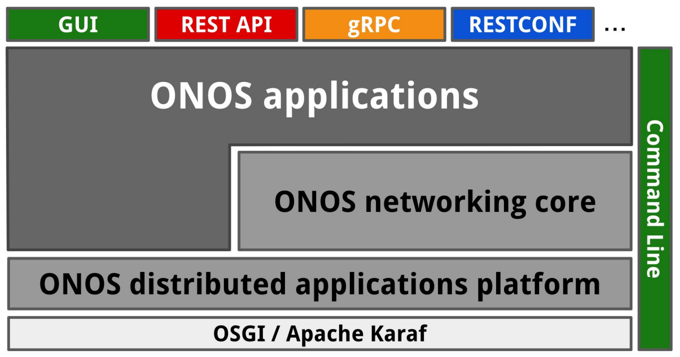

Fig. 3. Figure 3: ONOS Architecture:

The platform provides applications with a number of high-level abstractions, through which the applications can learn about the state of the network and through which they can control the flow of traffic through the network. The network graph abstraction provides information about the structure and topology of the network. As a part of this study the network path calculation module of ONOS was extended to take in meaningful constraints. In any real networks constraints are essential as they allow for creating network paths that support different criterion, e.g. disjoint paths for path protection switching. The intent is a network-centric abstraction that gives application programmers the ability to control network by specifying what they wish to accomplish rather than specifying how they want to accomplish it and in some cases can be specified as the constraints. The developed applications (core extensions) were loaded and unloaded dynamically, via REST API or GUI, and without the need to restart the cluster or its individual nodes. ONOS application management subsystem assumed the responsibility for distributing the application artefacts throughout the cluster to assure that all nodes are running the same application software.

III. PATH COMPUTATION

Borrowing the definition from RFC 4655 a Path Computation Element (PCE) is an entity that is capable of computing a network path or route based on a network graph, and of applying computational constraints during the computation. The PCE entity is an application that can be located within a network node or component, on an out-of-network server, for example in SDN Controller. The computation algorithm operates on the following

- A topology map (the Traffic Engineering Database TE DB) learnt from the network through a passive participation in the IGP.
- A connectivity graph formed by Nodes and Links describing the network
- Link and Node constraints which are referred to as metrics

The traffic engineering capabilities mandates that flows are routed across the network based on the resources a flow requires. To achieve this path computation engine needs to weigh the resources that are available on each link of the connectivity path and return a path that comprises of a collection individual links that has the resource capability to service the requirements for that flow. This brings in the concept of the constraints. Thus in network traffic engineering is viewed as mapping the flows over a physical topology. The shortest-path algorithm is very widely used because of its efficiency and stability. However networks are aware of Quality of Service (QoS). So it is imperative to find paths between a source destination pair that are different from the shortest path(s) returned by a minimum distance algorithm (Min-Dist1). This is because the shortest path may be highly congested while alternate paths between the source and the destination are running at relatively lower utilization. In addition, since flows now have different QoS requirements, a single path may not be able to satisfy requirements of all flows between two end points.

To achieve this ONOS SDN controller generates a network-wide MPLS LSP database (LSP DB). The PCE application computes the path and use information stored in the LSP DB to establish LSPs over the optimized path. [5]

The study was carried out in the following environment of ONOS SDN Controller.

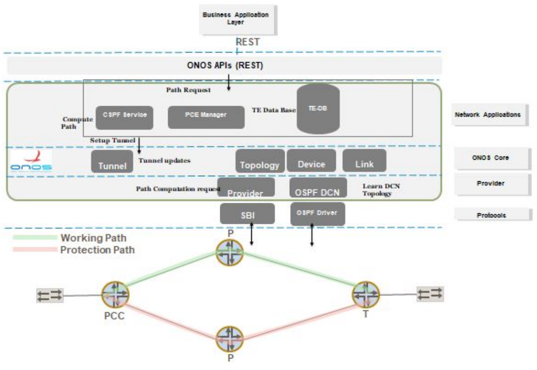

Fig. 4. Figure 4: Path Computation Application

1) The path request was received in the SDN Controller through an API. The request was received for the calculation of a number of network paths for which specific constraints were specified. Example Node Disjoint Paths for establishing protected paths in the network.

2) The PCE Manager application was responsible for handling the request to and from the North Bound Interface.

3) The network topology was learnt by the SDN controller. In the OpenFlow Environment the Traffic Engineering TE Database was built up by the functionality provided by the ONOS SDN Controller. This provided the network topology on which the Path Computation Algorithm was run.

4) The CSPF Services Application computed the path and reserved the bandwidth in the TE-DB. Multiple Paths were computed based on the constraints that were provided by a business application. The algorithm under study was implemented as part of this application.

5) Once the Paths were computed they were programmed in the network using the south bound interface of the

SDN controller

IV. Algorithm Overview

A computer network can typically be modelled as a dual sided directed graph. Thus calculating the path between any two nodes of the network is analogous to finding the path between two vertices of the graph. Over the course of time, several graph path searching algorithms have been developed each with the objective of finding the shortest path between a given pair of nodes. For graphs, like computer networks, the constraints or the weights of the edges are usually non zero and positive quantities, computer networks do not generally have negative quantities as weights or constraints. For shortest path searching in such a case, the most famous algorithm is the Dijkstra's path searching algorithm for graphs with non- negative weights. Dijkstra's algorithm returns the single shortest possible path between a pair of nodes in a graph. The worst case time complexity in the big O notation, of the Dijkstra's algorithm is **O(V2)** where V represents the number of nodes and E the number of edges. However, using the concept of a Fibonacci Heap, it can be shown that the complexity of the algorithm reduces to **O(E +VlogV)** if the graph is represented in an adjacency list form.

The Dijkstra's algorithm only returns a single possible shortest path between a pair of network nodes. The problem at hand requires us to find a K number of shortest paths in the networks, which can then be used in conjunction with a series of constraints in order to provide the required set of paths. The classic implementation of the K- Shortest Paths in a given graph is the Yen's Algorithm.[6] The Yen's algorithm returns the first K shortest loopless paths in a given graph in increasing order. Since the paths in computer networks are also required to be loopless, the choice of Yen's K Shortest path algorithm serves the purpose well in the scenario.[7] The alternative classical implementation of K Shortest Paths is the Eppstein's algorithm. However, the Eppstein's algorithm does not return only loopless paths, it returns paths containing loops which is not feasible in the case of computer networks.[8] For further discussion, the standard algorithm used in K shortest pathing searching will be based on the Yen model and not the Eppstein model. The Yen's model will be used as a reference for deriving the algorithm in question in this paper.

The Yen's Algorithm uses a shortest path searching algorithm for obtaining the first shortest path in the network. It may be any shortest path algorithm; in this case, the Dijkstra's algorithm is used. After finding the first shortest path, it recursively calls the Dijkstra's algorithm to find the next shortest path. It defines a root path and spur path at each stage, both being derived from the previous shortest path, by removing one edge at a time. The Dijkstra's algorithm is then used on the spur paths to generate the next possible shortest paths, the shortest of which becomes the required path. This process is carried out recursively to find K paths.

The following formulation is the pseudo code of the Yen's K Shortest Path algorithm.

```
Number of paths: K


function KSP (Graph, source, destination, K):

A[0] = Dijkstra(Graph, source, destination);
//Determine the shortest path from the source to the sink.

B = [];
// Initialize the set to store the potential kth shortest path.

foreach k in K do
    // The spur node ranges from the first node to the
      next to last node in the previous k-shortest path.

    foreach i in 0 to (A[k  1])  2: do
        // Spur node is retrieved from the previous
          k-shortest path, k  1.

        spurNode = A[k-1].node(i);
        // The spur node ranges from the first node to the
          next to last node in the previous k-shortest path.

        rootPath = A[k-1].nodes(0, i);
        foreach path p in A do
            if rootPath == p.nodes(0, i): then
                remove p.edge(i,i + 1) from Graph;
                // Remove the links that are part of the
                  previous shortest paths which share the
                  same root path.

            else
                continue;
            foreach node rootPathNode in rootPath
              except spurNode: do
                remove rootPathNode from Graph;
                // Calculate the spur path from the spur
                  node to the sink.

            spurPath = Dijkstra(Graph, spurNode,
              destination);
            // Entire path is made up of the root path and
              spur path.

            totalPath = rootPath + spurPath;
            // Add the potential k-shortest path to the
              heap.
            B.append(totalPath);
            // Add back the edges and nodes that were
              removed from the graph.

            restore edges to Graph;
            restore nodes in rootPath to Graph;

        if B is empty:
        // This handles the case of there being no spur
        paths, or no spur paths left.
        // This could happen if the spur paths have
          already been exhausted (added to A),

            break;

        B.sort();
        // Sort the potential k-shortest paths by cost.
        path.
        A[k] = B[0];
        // Add the lowest cost path becomes the k-shortest
        B.pop();
        //Remove the first element from the set B

    return A;
```

The Yens algorithm makes KT calls to the Dijkstra's algorithm where T is the length of the spur paths. The worst case time complexity of the Dijkstra's algorithm as mentioned earlier, was **O(N2)** where N is the number of nodes but was shown to be reduced to **O(M + NlogN)** using a Fibonacci heap, where M is the number of edges.[9]

Thus T can have a best case value of **O(logN)** and a worst case value of N. Thus the worst case time complexity for Yens algorithm is **O(KN(M + NlogN))**. If the Fibonacci heap is not used in the Dijkstra call, the worst case time complexity of Yen's algorithm increases to **O(KN3)**.

V. MODIFICATIONS IN THE ALGORITHM INVOLVING CONSTRAINTS

Practical applications of K shortest paths in computer networks may often require the paths to incorporate a certain number of constraints, in addition to being the shortest possible paths in terms of routing costs. We will now try to modify the existing Yens algorithm in incorporate a set of constraints while calculating the paths and returning only those paths which satisfy the given constraints.

For the purpose of convenience, the constraints will be treated as two sets of white edges and white vertices each. The idea is that the set of white edges and vertices is the set of those edges and vertices which must be included in every path returned by the algorithm. It may so happen that this results sometimes in a list of paths which are less than K in size. However, practically speaking, constraints are of much greater importance as opposed to a mere number of paths. Based on the fact that network solutions almost always require a set of constraints to be applied on the paths calculated for packet forwarding, this is not a very significant problem. If required, a number greater than K may be entered to return K paths but the additional benefit is that the constraints will be applied on each path, which is of utmost importance.

The concept involved is similar to the analogy of the Constrained Shortest Path First (CSPF) routing protocol as opposed to the Open Shortest Path First (OSPF) routing protocol. In OSPF, the first open path is immediately selected as the path for routing IP packets whereas in CSPF, the route is determined on the basis of constraints the network is subject to. A similar concept of layer 3 routing protocols is now being extended to the layer 2.5 MPLS protocol, which is extensively used in SDN. [10]

The set of white edges and white vertices will act as constraints for our system. Any constraint applied in terms of hop count, latency, bandwidth limitations or distance vectors, will all be converted in two sets of white vertices and white edges for the network graph. The sets of white vertices and edges must be included in each path found in the algorithm. If a path calculated does not include these two sets of constraints, the path will not be considered as a valid path in the network graph.

The following formulation is the pseudo code of the modified Yens algorithm in order to incorporate the constraints.

```
whiteEdges = set of white edges;
whiteVertices= set of white vertices;


  function KSP (Graph, source, destination, K,
    whiteEdges, whiteVertices):

  A[0] = Dijkstra(Graph, source, destination);
  //Determine the shortest path from the source to the sink.

  flag=true;

  if A[0].edges().donotcontain(whiteEdges): then
      flag=false;
      // Check if the first path contains the white edges
  if flag==true then
      if if A[0].vertices().donotcontain(whiteVertices) then
          flag=false;
          // Check if the first path contains the white
            vertices
```

```
if flag==true then
    C[0] = A[0];
    //Add the path to the final paths only if it satisfies the
      constraints
flag=true;
// Reset the flag condition for the next k iterations

B = [];
// Initialize the set to store the potential kth shortest path.


foreach k in K do
    // The spur node ranges from the first node to the
      next to last node in the previous k-shortest path.

    foreach i in 0 to (A[k  1])  2: do
        // Spur node is retrieved from the previous
          k-shortest path, k  1.

        spurNode = A[k-1].node(i);
        // The spur node ranges from the first node to the
          next to last node in the previous k-shortest path.

        rootPath = A[k-1].nodes(0, i);
        foreach path p in A do
            if rootPath == p.nodes(0, i): then
                remove p.edge(i,i + 1) from Graph;
                // Remove the links that are part of the
                  previous shortest paths which share the
                  same root path.

            else
                continue;
            foreach node rootPathNode in rootPath
              except spurNode: do
                remove rootPathNode from Graph;
                // Calculate the spur path from the spur
                  node to the sink.

            spurPath = Dijkstra(Graph, spurNode,
              destination);
            // Entire path is made up of the root path and
              spur path.

            totalPath = rootPath + spurPath;
            // Add the potential k-shortest path to the
              heap.


            B.append(totalPath);
            // Add back the edges and nodes that were
              removed from the graph.

            restore edges to Graph;
            restore nodes in rootPath to Graph;

        if B is empty:
        // This handles the case of there being no spur
        paths, or no spur paths left.
        // This could happen if the spur paths have
          already been exhausted (added to A),

            break;

        B.sort();
        // Sort the potential k-shortest paths by cost.
        path.


        if B[0].edges().donotcontain(whiteEdges): then
            flag=false;
            // Check if the first path contains the white
              edges
        if flag==true then
            if B[0].vertices().donotcontain(whiteVertices)
              then
                flag=false;
                // Check if the first path contains the white
                  vertices
        if flag==true then
            C[k] = B[0];
            // Add the next path to the final set only if the
              constraints are satisfied

        A[k] = B[0];
        // Add the next path to the iterating list for the
          next turn

        flag=true;
        // Reset the flag condition

        B.pop();
        //Remove the first element from the set B

    return C;
```

VI. TESTING THE MODIFIED ALGORITHM

The framework, utilities and algorithms for ONOS are all implemented in Java 8. Thus, for the purpose of testing the new algorithm, an environment was set up in a Java Standard Environment in order to obtain the required results of the testing phase. The following steps were steps were executed to testing the new, modified algorithm.

A. Compiling the ONOS source code:

The source code of ONOS was taken from the open source platform; a new test file was added to the same, which contained the modified K Shortest Path searching algorithm. The entire code for the same was written in Java 8, by defining appropriate interfaces, classes and methods. The source code was then compiled into the ONOS API using the BUCK build tool. The BUCK build tool is a similar building tool as compared to CMAKE in C++, which is used for Java applications. Once compiled, the ONOS code was successfully converted into a set of runtime APIs. These runtime APIs were exposed to a test environment.

The Apache Maven tool was used as the project management platform in the testing phase. The Maven contains a .m2 repository for managing and running the compiled code on the ONOS controller. Once the program written within the source code was compiled, the results were stored in the form of Java Archive files (.jars). These new files were then pushed into the Maven repository, which could then be used as runtime APIs. The Maven platform provided a coherent and organized method to deploy the application on the ONOS SDN controller and run the simulations for the testing phase.

B. *Emulating the network virtually for testing*

In the real world scenario of SDN, the SDN controller connects to physical switches of the computer network using its southbound protocols like the OpenFlow, in order to pass the control information to the data plane. However, for testing purpose in this scenario, there was no access to physical network switches. In order to overcome this, a software emulator platform of the name Mininet was used. Mininet is the standard network emulator to test applications on ONOS by providing a virtual network. Mininet is a framework written in Python which allows one to define virtual switches, links and hosts using automated scripts. The SDN controller treats these virtual switches as physical ones and passes the control information to the data plane of the switches, using the OpenFlow interface. In this test scenario, several Mininet network topologies were used to simulate a real time physical network. Mininet supports parametrized topologies. With the help of Python code, flexible topologies were created that were used for verifying the algorithm.

C. *Setting up a Testing Class and Framework*

A test class was created to incorporate and map the physical topology of the network into a network graph. The class and the environment were defined in Java. The test class included a call of the modified algorithms method which was defined and compiled along with the source code of the ONOS. The lists of white edges and white vertices were populated over and over again, including different sets of constrained edges and vertices each time. This was done in several phases by populating the constraints data and dynamically filling the datasets by retrieving values from the network topology. ONOS provides an inbuilt support for the Apache Karaf framework which was used to view the data logs and the results of the testing phase of the algorithm. In order to build the test class and environment, the Apache Maven framework was used, to return a built class file in the form of an OAR snapshot. On creating the file in the required form, it was deployed on the ONOS runtime APIs compiled from the source code. This was used with various topologies were set up in the environment with the use of Mininet. [11]

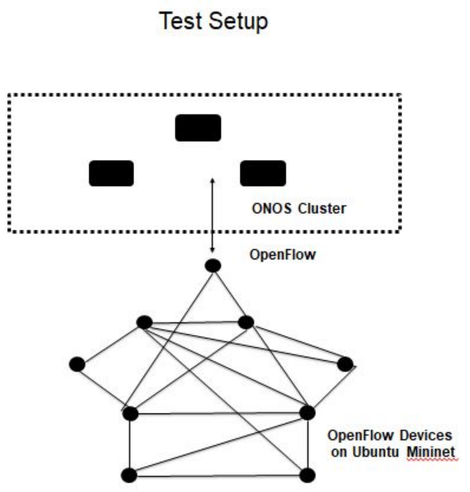

Fig. 5. Figure 5: The Test Setup

D. *Running the algorithm on the network*

The final phase of the testing included calling the modified algorithms method and the original K shortest path algorithm on various network topologies. To test the algorithms, several pairs of sources and destination vertices were defined on the network topologies created using Mininet, which were passed into the algorithms method. The resultant paths were noted at every stage and consistencies in the results was also studied. It was seen that the time taken to execute both the algorithms was roughly the same as the new modifications do not add to the time complexity of the original Yen's algorithm. The constraints on the network changed several times in terms of the sets of white edges and white vertices.

On studying the logged results of the algorithm, it was seen that all the paths returned in the final stage included the required white edges and vertices. The exercise was repeated several times widely varying the constraint based white edges and vertices in order to check the robustness of the algorithm.

In a nutshell, while keeping the time complexity of the original algorithm the same, the constraints were included into the system in order to only return paths, in which the constraints held true.

VII. CONCLUSION

The objective of this paper was to develop a constraint based K Shortest Path searching algorithm which could be deployed in Software Defined Networks (SDN). The idea was initially drawn from the concept of CSPF routing using in IP packet forwarding in the layer 3 of the traditional computer network. However, most of the SDN environments involved, in this case ONOS, mainly rely on the layer 2.5 protocol of MPLS to forward the data packets, a similar concept has been successfully extended to the core path searching algorithm used in the process. By incorporating the constraints, within the algorithm itself, it propagates ease of use and flexibility of the same. Any modifications further can be made within the source of the framework with relative ease. Also, in the process of modifying the algorithm, no additional time complexity was introduced into the same. In addition to this, the solution set size for the number of paths returned in the result set is significantly smaller. If further searching is now required on the solution set, the time taken will be reduced by a huge amount, leading to greater efficiency in applications where the algorithm is used.

The base algorithm used for further development is the Yens K Shortest Path searching algorithm. The results of the modified algorithms testing phase show that it can calculate the required paths of the constraint based scenario with a similar efficiency of the original Yen's K Shortest Path searching algorithm for graphs. The modified algorithm holds the door open and provided further scope of improvement as and when deemed necessary in the SDN framework. Further research can be undertaken to improve the efficiency and computation of the algorithm to an even greater extent. However, as of now, a real time constraint based K Shortest Path searching algorithm has successfully been created for use and deployment in Software Defined Network platforms and is being deployed in process of being commercially deployed in Service Providers Networks.

VIII. ACKNOWLEDGEMENTS

This work has been supported by the ALTRAN Technologies Ltd, as part of a summer internship program, voluntarily taken up as an addition to the curriculum of the Vellore Institute of Technology. The author would like to thank Mr. Saket Bharadwaj and Mr Suhail Ahmad Khan , Technical Leaders at ALTRAN for their support and guidance.

REFERENCES

[1] SDN: Software Defined Networks- An Authoritative Review of Network Programmability Technologies, Thomas Nadeau and Ken Gray

[2] Software-Defined Networking Using OpenFlow: Protocols, Applications and Architectural Design Choices, Wolfgang Braun and Michael Menth Future Internet 2014, 6, 302-336; doi:10.3390/fi6020302

[3] https://en.wikipedia.org/wiki/ONOS

[4] https://wiki.onosproject.org/display/ONOS/Wiki+Home

[5] RFC A Path Computation Element (PCE)-Based Architecture.

[6] Yens K Shortest Path Algorthim, 1970, Yen, Jin Y. (1970). "An algorithm for finding shortest routes from all source nodes to a given destination in general networks". Quarterly of Applied Mathematics. 27 (4): 526530. MR 0253822

[7] Yen, Jin Y. (1970). "An algorithm for finding shortest routes from all source nodes to a given destination in general networks". Quarterly of Applied Mathematics. 27 (4): 526530. MR 0253822

[8] Eppstein, David (1998). "Finding the k Shortest Paths" (PDF). SIAM J. Comput. 28 (2): 652673. doi:10.1137/S0097539795290477

[9] Fredman, Michael Lawrence; Tarjan, Robert E. (1984). Fibonacci heaps and their uses in improved network optimization algorithms. 25th Annual Symposium on Foundations of Computer Science. IEEE. pp. 338346. doi:10.1109/SFCS.1984.715934

[10] MPLS Fundamentals, By Luc De Ghein Nov 21, 2006 (ISBN 1-58705-197-4)

[11] "ON.Lab Delivers Software for New Open Source SDN Network Operating System - ONOS". PR Newswire. 2014-12-04. Retrieved 2016-06-08.

The author **Siddhant Ray** is currently in his 4th academic year, pursuing his B. Technology in Electronics and Communication Engineering from the Vellore Institute of Technology (VIT), Vellore, India. His areas of research interest include computer networks, machine learning, blockchain and digital communication.