# The Top-K QoS-aware Paths Discovery for Source Routing in SDN

**Xi Chen[1], Junlei Wu[1,2] and Tao Wu[3]**
[1] School of Computer Science and Technology, Southwest Minzu University
Chengdu, Sichuan 610041 - China
[e-mail: cx@swun.edu.cn]
[2] School of Computer Science and Technology, Hangzhou Dianzi University
Hangzhou, Zhejiang 310018 - China
[e-mail: 1271990125@qq.com]
[3] School of Computer Science, Chengdu University of Information Technology
Chengdu, Sichuan 610225 - China
[e-mail: wut@cuit.edu.cn]
*Corresponding author: Xi Chen

*Received May 29, 2017; revised August 29, 2017; accepted February 13, 2018; published June 30, 2018*

## Abstract

Source routing is the routing scheme that arranges the whole path from source to target at the origin node that may suit the requirements from the upper layer applications' perspective. The centralized control in SDN (Software-Defined Networking) networks enables the awareness of the global topology at the controller. Therefore, augmented source routing schemes can be designed to achieve various purposes. This paper proposes a source routing scheme that conducts the top-K QoS-aware paths discovery in SDN. First, the novel non-invasive QoS over LLDP scheme is designed to collect QoS information based on LLDP in a piggyback fashion. Then, variations of the KSP (K Shortest Paths) algorithm are derived to find the unconstrained/constrained top-K ranked paths with regard to individual/overall path costs, reflecting the Quality of Service. The experiment results show that the proposed scheme can efficiently collect the QoS information and find the top-K paths. Also, the performance of our scheme is applicable in QoS-sensitive application scenarios compared with previous works.

***Keywords:*** Software-Defined Networking (SDN); Source Routing; K Shortest Paths (KSP); Quality of Service (QoS); Path Discovery

## 1. Introduction

Source routing is the routing scheme that arranges the whole path from source to target at the origin node. In traditional IP networks, routes are determined by the intermediate routers in a hop-by-hop fashion. Source routing differs from traditional routing in that paths are designated by the origin node. The origin node specifies the whole path in the Source Routing and Record Route option of the IPv4 header. Source routing works in two flavors, namely Strict Source and Record Route (SSRR) and Loose Source and Record Route (LSRR). SSRR requires paths to be exactly concatenated by routers specified in the IP header options, no more or no less. On the other hand, LSRR loosely requires the routers that must be traversed, but more routers are allowed to appear between any two consecutive routers specified in the IP header options.

One of the well-known usages of source routing is its application in the traceroute utility to detect the round-trip path between source and target. Source routing also has its application in wireless networks. In the MANET and Ad Hoc Network arena, Dynamic Source Routing (DSR) [1], an on-demand routing protocol, is a typical source routing scheme. DSR discovers the whole path from source to target in a peer-to-peer fashion. DSR allows the origin node to carry path information in the packet header so that intermediate nodes do not have to maintain the routing table. Instead, paths are merely cached to keep minimum space usage.

Despite the above applications, the utilization of source routing is limited in real-world networks due to that the path specified by the origin node might not be consistent with the current topology. The essential reason for this deficiency is the lack of the global topology view for the origin node to determine a consistent path. Recent years have witnessed the rise of SDN (Software-Defined Networking) [2]. SDN is an emerging network paradigm which splits the forwarding plane (e.g., switches) and the control plane (i.e., controllers). Usually, the LLDP (Link Layer Discovery Protocol) works in the southbound of SDN. It discovers and aggregates topology of the underlying network so that the controller is aware of the global topology. With the global topology view at hand, it is possible for the SDN controller to achieve globally optimized resource composition and utilization, including optimized end-to-end paths. Therefore, it is quite likely to find the consistent path for source routing in the context of SDN. Usages of source routing in SDN are seen in various aspects, for example, in data center networks [3]–[5], WAN [6], [7], traffic steering/middlebox chaining [8]–[10], fault tolerance/recovery [3], [11]–[13], SFC (Service Function Chaining) [4], [14]–[17], etc.

Source routing can be divided into two stages: path discovery and path deployment (i.e., applying the found path for data forwarding). This paper focuses on the path discovery in the SDN environment. The authors have identified several issues in this topic in the current literature.

- Alternative paths discovery is seldom studied in the current literature. Although reference [3] finds alternative paths, the solution is far from optimized. Besides, QoS is not considered as a path discovery criterion, thus, it is not suitable in QoS-critical applications.
- Intermediate nodes constraint is seldom considered. In applications such as middlebox chaining [8]–[10], paths are required to traverse various intermediate nodes to, for example, deliver specific functions (firewall, deep packet inspection, etc.) or load balancing.

This paper proposes the top-K QoS-aware paths discovery in SDN environment. It finds the top-K paths using the K Shortest Paths (KSP) algorithm and its variations. The contributions of the work are twofold.

- A non-invasive QoS information collecting scheme. This scheme uses LLDP as the "ferry" to load QoS information collected from underlying switches in a piggyback fashion so that no fundamental modification of the current OpenFlow-based southbound interface needs to be made.
- QoS-aware multi-path discovery. The QoS information collected can be utilized to support QoS-aware path discovery. This work conducts a multi-path discovery so that top-K paths are returned for backup or load balancing purposes while satisfying the intermediate nodes constraints.

This paper is organized as follows. Section 2 summarizes previous works. Section 3 discusses the non-invasive and piggyback style QoS information collecting scheme based on LLDP. Section 4 derives the variations of KSP algorithms to find constrained/unconstrained top-K paths based on individual/overall path costs. Experiments based on simulation are conducted on Section 5. And finally, this paper is concluded in Section 6.

## 2. Related Works

In this section, we summarize related works in SDN source routing. Compared with traditional source routing path discovery such as DSR which discovers paths in a distributed fashion, SDN-based source routing path discovery works in a centralized fashion. Distributed path discovery involves traffic overhead (the on-demand RREQ and RREP, or the table-driven routes updates) to deliver topology information or path information. SDN-based source routing, on the contrary, maintains the current topology and paths in the controller thus at large reducing traffic overhead for routing.

Source routing is actively used in SDN researches. Abujoda et al proposed the SDN-based source routing for scalable service chaining in datacenters [4]. The work puts the path information in the packet header. Intermediate switch ports are encoded as plain numbers to reduce the need for extra bytes for path information so that minimum modification of packets can be made. However, it focuses on path deployment and does not specify how paths are discovered in the SDN-based datacenters. Besides, it lacks the QoS support.

Ref [7] focuses on controller scalability and performance issues in SDNs, and discusses a new routing scheme that leverages a variation of source routing for use in OpenFlow-based networks. The research aims to reduce the state needed to be distributed to the network devices by the controller(s) in SDNs, and in turn improve the scale, convergence time, fault tolerance and cost of such network architectures. This work derives the number-encoded source routing path expression. The path expression is pushed from the controller only to the ingress node, which at large reduces the forwarding states that must be propagated to data plane switches.

StEERING [8] tries to arrange a path traversing specific middleboxes by extending the OpenFlow and NOX controller. The key of the extension is the split of a monolithic flow table into several micro tables to constrain the "rule explosion". StEERING solves the path planning problem using Graph Theory. The main deficiency of StEERING is the lack of QoS support. It conducts the path discovery mainly based on middleboxes functions.

Reference [5] proposes a flow table update method based load balancing to avoid the controller from getting over-loaded with regard to interaction traffic between the control plane and data plane. The load balancing measurement differs from traditional load balancing in that

it considers both the loads in the control plane and data plane. By comparing the load of the control plane and data plane in each domain, flow table entry is dynamically installed based on a combination of source routing and direct installation. The controller's load could be reduced and the load balance of the control plane and data plane could be dynamically adjusted. The controller computes the appropriate path which is then sent to the ingress node. The ingress node takes it as the source routing information and passes it to the successive nodes. Nodes along the path install path information as flow entries. The flow table entries for the first H hops on the path are pushed using a modified source routing method whereas the remains hops install flow entries directly instructed by the controller.

SlickFlow [3] focuses on the source routing based fault recovery. The fault recovery application on the controller pushes the path information as special headers on the ingress switches. The path consists of several segments where the next hop and the alternative path from that hop are contained in each segment. Upon network failure at a certain node, alternative path is adopted proactively by the current node to reduce controller intervention. The source routing in SlickFlow is SSRR. The path discovery is an "all routes" based solution which is not optimized in large-scale networks.

Our work, as we will discuss in later sections, differs from these works in that: (1) QoS information is considered as a path discovery criterion to support QoS provisioning; (2) multiple paths can be found; (3) node constraints are also considered to find restricted paths.

## 3. QoS Information Collecting and Evaluation

### 3.1. QoS over LLDP Scheme

In standard SDN networks, controllers are aware of switches directly connected to them through bidirectional Hello messages in the standard OpenFlow protocol. However, the underlying link states between switches (i.e., how switches are mutually connected) are not visible to controllers in the first place. Therefore, in the initial stage of an SDN network, controllers do not have the topological knowledge of the whole SDN network. In order to perform centralized control over an SDN network, controllers must carry out topology discovery. Controllers usually use LLDP to fulfill such a task. LLDP is an IEEE proposed protocol widely used in network arena for topology discovery.

The controller instructs a switch to multicast the LLDP packet to all of its ports through a PacketOut (instructive packets from controllers to switches). In this PacketOut, topological information of the switch such as chassis information, port information, etc., is all contained. All other switches connected to this sender switch receive the LLDP packet, and then match this packet against the flow table entries of their own, only to find no matches for LLDP packets. Thus, switches will send a PacketIn (packets from switches to controllers) containing this LLDP to the controller asking how to process this packet. Since the PacketIn contains topology information about both the sender switch and the receiver switch, the controller can now assert that there exists a link between the two switches based on the received PacketIn. By means of this iterative PacketIn/Out interaction, topology of the whole SDN network can be discovered by the central controller. This centralized topology discovery in SDN is quite different from how LLDP works in traditional networks where topology discovery is done by individual switches independently, although LLDP is used in both cases.

Standard LLDP packets usually contain basic information such as the MAC address, chassis information, port information, etc. We can see from the above topology discovery phase, no QoS information is contained in LLDP packets. Should QoS information be

incorporated, the QoS-aware topology discovery can be done to enable further QoS-aware decisions and policies, thus QoS provisioning becomes possible.

LLDP is a TLV (Type/Length/Value, i.e., key-value pair with length information) based protocol where TLVs are used for property description. We can include QoS information as custom TLVs in LLDP packets. In this way, LLDP can be seen as the "ferry" containing QoS information (i.e., QoS over LLDP) and other useful properties as its payload. We define QoS TLV as follows in **Fig. 1**.

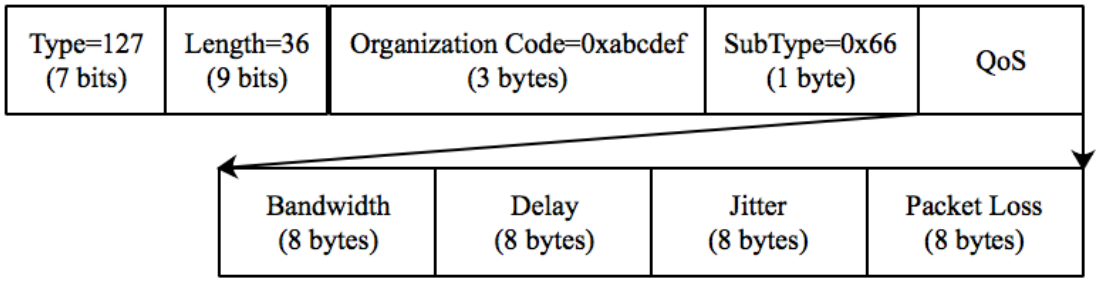



**Fig. 1.** QoS over LLDP packet format

In the TLV Type field, it must be designated as 127 to indicate this is a custom TLV. The Length field specifies the variable-length value contained in the TLV. The Organization Code field indicates the designer of this customized TLV. We use the Organization Code as 0xabcdef for the time being. The Subtype field specifies the detailed type of the contained value. Value String field (i.e., the QoS field in **Fig. 1**) gives the real value. We contain various QoS properties in the Value String. In order for the receiver to conveniently parse the different QoS properties, we use the predefined property order and length. We can see from **Fig. 1** several properties are included in fixed length in our current settings, namely delay, bandwidth, packet loss and jitter, 8 bytes for each property. Therefore, a QoS over LLDP packet is 38 bytes longer than a pure LLDP packet in length. Note that more properties such as availability can be included in the future work. Upon receiving the QoS over LLDP packet, the switch fills QoS properties in corresponding TLV fields. We have implemented this mechanism in OVS (Open vSwitch) [18]. Section 5.1 will demonstrate the QoS collecting capabilities and performance of QoS over LLDP.

### 3.2. QoS Property Conversion Algorithm

After QoS over LLDP collects QoS information at the southbound interface, QoS information is preprocessed to enable the QoS-aware finding of the top-K paths (see for detail in Section 4). Commonly seen QoS properties are used for path evaluation. The top-K paths discovery algorithms used in this paper are the top-K paths variations of the Dijkstra [19] algorithm, i.e., the K Shortest Paths (KSP) [20]. Dijkstra uses ***cost*** to evaluate a path, therefore, QoS properties must be transformed as additive and negative (i.e., bigger values means worse) properties so that cost for each intermediate node and the whole path can be calculated. Paths are thus ranked by costs which in turn reflect the QoS. In this subsection, we introduce the transformation algorithms that convert various QoS properties to costs.

#### 3.2.1. Calculating Individual Path Costs

**Table 1** exhibits how commonly seen QoS properties can be converted as additive and negative properties to calculate costs (the 3rd column). $v_{i,j}$ is the actual QoS property value of

the intermediate node $n_i$ where $j$ indicates the $j$-th QoS property, i.e., $j \in \{dl, bw, pl, av, jt\}$, namely, ***delay***, ***bandwidth***, ***packet loss***, ***availability*** and ***jitter***. Notice that more QoS properties can be used in real applications. For delay and jitter which are natively additive and negative, their corresponding costs are themselves. Bandwidth is neither a positive nor an additive property along the path. Thus, we follow the OSPF metric calculation convention: the outgoing interface bandwidth of every node is used to divide a benchmark bandwidth. In traditional settings, the benchmark is 100,000,000 bps. We can also set the benchmark to be the bandwidth of the ingress node. For availability, it can be converted as negative by getting its reciprocal; it then can be converted as additive from multiplicative by applying logarithm. Similar method can be used for packet loss.

In addition, **Table 1** also gives how path properties can be calculated using intermediate node properties (the 2nd column).

Although different algorithms apply for different node properties when they are aggregated to calculate corresponding path properties, they share the same path property cost algorithm (the 4th column) once they are converted as additive and negative. The costs, $c_{path,j}$, in the 4th column are referred to as ***individual path costs*** since they correspond to a given individual property $j, j \in \{dl, bw, pl, av, jt\}$, of the path.

**Table 1.** QoS Property Transformation

| Node Property | Path Property | Node Property Cost | Individual Path Cost |
|---|---|---|---|
| delay (negative) | $\sum_{i=1}^{n} v_{i,j}$ | $c_{i,j} = v_{i,j}$ | $c_{path,j} = \sum_{i=1}^{n} c_{i,j}$ |
| bandwidth (positive) | $\min(v_{i,j})$ | $c_{i,j} = \frac{v_{bench}}{v_{i,j}}$ | $c_{path,j} = \sum_{i=1}^{n} c_{i,j}$ |
| packet loss (negative) | $1 - \prod_{i=1}^{n}(1 - v_{i,j})$ | $c_{i,j} = \log\left(\frac{1}{1 - v_{i,j}}\right)$ | $c_{path,j} = \sum_{i=1}^{n} c_{i,j}$ |
| availability (positive) | $\prod_{i=1}^{n} v_{i,j}$ | $c_{i,j} = \log\left(\frac{1}{v_{i,j}}\right)$ | $c_{path,j} = \sum_{i=1}^{n} c_{i,j}$ |
| jitter (negative) | $\sum_{i=1}^{n} v_{i,j}$ | $c_{i,j} = v_{i,j}$ | $c_{path,j} = \sum_{i=1}^{n} c_{i,j}$ |

**Fig. 2** shows an example of a simple topology where 2 paths exist between source node n1 and target node n6, namely p1 (dashed lines) and p2 (solid lines). The upper half of **Table 2** exhibits the actual values of QoS properties of various nodes (lines from $v_{1,j}$ to $v_{6,j}$) and the lower half shows the corresponding costs (lines from $c_{1,j}$ to $c_{6,j}$) calculated using algorithms introduced in **Table 1**. Lines in the middle (lines $p1_j$ and $p2_j$) show the aggregated path properties and the lines at the bottom show the individual path costs (lines $c_{p1,j}$ and $c_{p2,j}$). We can see from these lines that lower individual path costs always correspond to better paths in terms of various individual QoS properties.

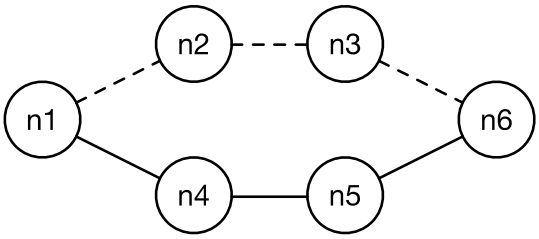



**Fig. 2.** Multi-Path Example

**Table 2.** QoS Property Example

| | DL (ms) | BW (Mbps) | PL | AV | JT (ms) |
|---|---|---|---|---|---|
| $v_{1,j}$ | 1.000 | 4.000 | 0.100 | 0.900 | 0.010 |
| $v_{2,j}$ | 2.000 | 2.000 | 0.050 | 0.990 | 0.010 |
| $v_{3,j}$ | 2.000 | 1.000 | 0.200 | 0.980 | 0.010 |
| $v_{4,j}$ | 3.000 | 6.000 | 0.030 | 0.750 | 0.030 |
| $v_{5,j}$ | 3.000 | 5.000 | 0.030 | 0.800 | 0.100 |
| $v_{6,j}$ | 1.000 | 2.000 | 0.100 | 1.000 | 0.010 |
| $p1_j$ | 6.000 | 1.000 | 0.384 | 0.873 | 0.040 |
| $p2_j$ | 8.000 | 2.000 | 0.238 | 0.540 | 0.150 |
| $c_{1,j}$ | 1.000 | 1.000 | 0.046 | 0.046 | 0.010 |
| $c_{2,j}$ | 2.000 | 2.000 | 0.022 | 0.004 | 0.010 |
| $c_{3,j}$ | 2.000 | 4.000 | 0.097 | 0.009 | 0.010 |
| $c_{4,j}$ | 3.000 | 0.667 | 0.013 | 0.125 | 0.030 |
| $c_{5,j}$ | 3.000 | 0.800 | 0.013 | 0.097 | 0.100 |
| $c_{6,j}$ | 1.000 | 2.000 | 0.046 | 0.000 | 0.010 |
| $c_{p1,j}$ | 6.000 | 9.000 | 0.211 | 0.059 | 0.040 |
| $c_{p2,j}$ | 8.000 | 4.467 | 0.118 | 0.268 | 0.150 |

### 3.2.2. Calculating Overall Path Costs

Paths can be selected based on individual path costs, along with the traffic types (e.g., the transport layer port number, the ToS field in the IP header, etc.). For example, for time-critical traffic or applications, p1 is preferable since it has lower delay cost while for bandwidth-consuming traffic such as video streaming, p2 is preferred since it has lower bandwidth cost. In some other cases, paths are selected based on their overall quality for generic traffic (e.g., best effort), therefore, various path property costs need to be aggregated for the ***overall path cost***. To calculate the overall path cost, 2 steps must be conducted.

- Normalization of node property costs: Normalization eliminates effects of the scale and the unit of different node property costs so that costs are treated in a unified manner for later aggregation. By substituting actual values $v_{i,j}$ in the 3rd column of **Table 1** with normalized values $nv_{i,j}$, normalization can be achieved. $nv_{i,j}$ can be calculated using the following equations, equation (1) for positive properties and (2) for negative, respectively.

$$nv_{i,j} = \begin{cases} \frac{v_{i,j} - v_j^{max}}{v_j^{max} - v_j^{min}}, v_j^{max} \neq v_j^{min} \\ 1, v_j^{max} = v_j^{min} \end{cases} \quad (1)$$

$$nv_{i,j} = \begin{cases} \frac{v_j^{max} - v_{i,j}}{v_j^{max} - v_j^{min}}, v_j^{max} \neq v_j^{min} \\ 1, v_j^{max} = v_j^{min} \end{cases} \quad (2)$$

- Weighted summation of the path property costs: The above step gets normalized property costs for a given path. The weighted summation of these costs is applied to calculate the overall cost of a path.

$$\begin{cases} c_{path} = \sum w_j \times c_{path,j} \\ \sum w_j = 1 \end{cases}, j \in \{dl, bw, pl, av, jt\}, w_j \geq 0 \quad (3)$$

Note again that, $c_{path,j}$ is the criterion to evaluate a path based on the given property $j$ while $c_{path}$ is used to evaluate a path based on the overall status. Both costs can be used to rank paths during path discovery.

## 4. KSP-based Paths Discovery

In the previous section, we can see that paths can be evaluated and ranked by path costs (either by individual path costs or by overall path costs); But simply iterating and ranking every path in a large topology can be time-consuming. Cost-based evaluation can be done using Dijkstra-like algorithms. We solve the QoS-aware source routing paths discovery and ranking using the KSP algorithm [20] and its variations. KSP finds up to K highly ranked paths. We develop two KSP-based solutions, namely:

- The unconstrained source routing paths discovery: In this solution, the paths discovery request does not constrain what nodes to be traversed. The top-K unconstrained paths from source to target are found purely based on path costs (by individual path costs $c_{path,j}$ or overall path costs $c_{path}$).
- The node-constrained source routing paths discovery: In the solution, the paths discovery request specifies nodes that must be traversed along the path. The top-K node-constrained paths found must traverse all the specified nodes and are ranked based on path costs ($c_{path,j}$ or $c_{path}$).

### 4.1. The Unconstrained Source Routing Paths Discovery

#### 4.1.1. The Model

We model the SDN data plane as a graph where nodes represent switches and edges represent links. $(N, A)$ represents the directed graph where the finite set $N = \{v_1, v_2, \dots, v_n\}$ is the vertices set and the finite set $A = \{a_1, a_2, \dots, a_m\} \subseteq N \times N$ is the directed edges (arcs) set. An arc is an ordered pair $a_k = \{v_i, v_j\}, i \neq j$ associated with a positive real number which is the cost of this arc, denoted as $c_{i,j}$. $c_{i,j}$ can be assigned using a given node property cost in the 3rd column of **Table 1** or the normalized weighted sum of these costs, in our case. A path is denoted as $p = \langle v_s = v_1', v_2', \dots, v_l' = v_t \rangle$ where $v_s$ and $v_t$ represent the source node and destination node of the path. $cost(p)$ represents the cost of path p, either using individual path cost $c_{path,j}$ or the overall path cost $c_{path}$ which is easily calculated by the algorithm specified in the 4th column of **Table 1**. Expression $sub_p(x, y) = \langle v_x', v_{x+1}', \dots, v_{y-1}', v_y' \rangle$, $x, y \in$

$\{1,2,\dots,l\}$ represents a sub-path of path $p$ from node $v_x'$ to $v_y'$. The unconstrained source routing paths discovery can be modelled as follows. Note that in the further description, we have the following notations: K (in uppercase) is the number of shortest paths to be found; k (in lowercase) represents the k-th shortest path.

**Definition 1** The Unconstrained Source Routing Paths Discovery: The source node $v_s$ and destination node $v_t$ are determined by the user. Given a positive integer K, the unconstrained source routing paths discovery finds set $P^K = \{p^1, p^2, \dots, p^K\} \subseteq P_{st}$ where $P_{st}$ is the set containing all the paths from $v_s$ to $v_t$ such that:

- $\forall k \in \{1,2,\dots,K\}$, $p^k$ is loopless;
- $\forall i,j \in \{1,2,\dots,K\}$, $p^i \neq p^j$, i.e., no duplicated path;
- $\forall k \in \{1,2,\dots,K-1\}$, $cost(p^k) \leq cost(p^{k+1})$, i.e., $p^k$ is found before $p^{k+1}$ (shorter paths are found first).
- $\forall p \in P_{st} - P^K$, $cost(p^K) \leq cost(p)$, i.e., top-K least-costly paths are to be found.

### 4.1.2. The Algorithm

The unconstrained source routing paths discovery can be solved using KSP. KSP tries to establish a tree with the K shortest paths. Suppose that K shortest paths have been found. All these paths can be represented as a tree with the source node $v_s$ as the root and the destination node $v_t$ being duplicated K times as leaves. And all the branches from source to destination constitute the top-K shortest paths. Apparently, if we have already established the K shortest path tree, then the K+1 shortest path consists of two parts:

- A sub-path from the source node $v_s$ to an intermediate node $v_i$ ($v_s$ and $v_i$ can be the same node), i.e., the root. According to Dijkstra algorithm, a sub-path of the shortest path is also the shortest one from $v_s$ to $v_i$;
- The shortest path from the intermediate node $v_i$ to the destination node $v_t$, i.e., the spur. This shortest path is constrained in that none of the edges of the K shortest path tree is used to produce it, to prevent duplicated paths or looped paths.

The intermediate node $v_i$ is referred to as the "deviation node" for the K+1 shortest path [15]. The basic thoughts of the algorithm are as follows. First, we calculate the shortest path from the source node $v_s$ to the destination node $v_t$. Then every node along this path is treated as the deviation node to execute a heuristic exploit. During the exploit, the candidates for the second shortest path can be found to be placed in a set. The shortest path in this set is the second shortest path. The next step is that every node along the second shortest path is treated as the deviation node to find the third shortest path, as do in the search for the second shortest path. Then the algorithm iterates till the first K shortest paths are found. Details of the KSP algorithm are given below.

Algorithm 1 and 2 shows in detail the KSP algorithm, where dsp($v_s$, $v_t$) calculates the shortest path from the source node $v_s$ to the destination node $v_t$ using Dijkstra algorithm. The routine nextPath($v_s$, $v_t$) finds the next shortest path between $v_s$ and $v_t$ using deviation method (see details in Algorithm 2). nextPath($v_s$, $v_t$) is the core of the algorithm, and is used to derive variations of the basic KSP in later sections. restoreGraph() is used to set the graph back to its initial linkage states after temporary deletion of arcs. $P^K$ is the ordered set that stores the K shortest paths. $P^C$ is the set that stores the shortest paths candidates. In real implementation, a priority queue (e.g., a heap) can be used for $P^C$ to guarantee that every time a path is fetched from this set it is the shortest one, to improve efficiency. Detailed explanations are shown in the comments in Algorithm 1 and 2.

**Algorithm 1.** KSP($v_s$, $v_t$, $K$)

***Input***: $v_s$: *the source node*
$v_t$: *the destination node*
$K$: *the number of shortest paths*
***Output***: $P^K$: *the K shortest paths set*

```
1:  P^K = ∅;                          //P^K: shortest paths set.
2:  P^C = ∅;                          //P^C: shortest path candidates set
3:  p^1 = dsp(v_s, v_t);              //find the shortest path using Dijkstra algorithm
4:  if p^1 == NULL then
5:      return P^K;                   //no connectivity from source to target
6:  end if
7:  P^K = P^K ∪ {p^1};
8:  while ||P^K|| < K do
9:      curSPath = nextPath(v_s, v_t); //see Algorithm 2 to find next shortest
10:     if curSPath==NULL then
11:         break;                    //no more shortest path, break
12:     end if
13: end while
14: return P^K;
```

**Algorithm 2.** nextPath($v_s$, $v_t$)

***Input***: $v_s$: *the source node*
$v_t$: *the target node*
***Output***: *curSPath: the current shortest path*

```
1:  prevSPath = max(P^K);             //shortest path found in the last iteration
2:  for i = 1; i ≤ prevSPath.length; i++ do
3:      root = sub_prevSPath(1, i);   //get root for every node
4:      for each path from P^K do
5:          root_path = sub_path(1, i);
6:          if root = root_path then
7:              c_{i,i+1} = ∞;        //determine the furthest node as deviation
8:          end if
9:          spur = dsp(v_i, v_t);     //get spur from deviation node
10:         restoreGraph();           //restore the linkage for deviation nodes
11:         if spur=NULL then
12:             return NULL;          //no more valid spur, return NULL
13:         end if
14:         if root ∩ spur = ∅ then
15:             cPath = root+spur;    //construct a loopless candidate
16              P^C = P^C ∪ {cPath};
17          end if
18      end for
19: end for
20: curSPath=min(P^C);                // shortest from P^C is the current shortest
```

```
21:  if curSPath! = NULL then
22:        P^C = P^C − {curSPath}
23:        P^K = P^K ∪ {curSPath};
24:  end if
25:  return curSPath;
```

### 4.1.3. Proof and Analysis

In this section, we analyze the complexity of the algorithm.

**Theorem 1**: The complexity of Algorithm 1 is: $O(Kn(m+nlogn))$. $n$ is the number of nodes, $m$ is the number of the links in the directed graph and $K$ is the number of paths to be found.

**Proof**: The complexity of the heap-optimized Dijkstra algorithm is $O(m+nlogn)$. Since in KSP, all the nodes in path $p^k$ must be traversed as deviation nodes to execute heuristic exploits for $p^{k+1}$, the complexity is $O(l_k(m+nlogn))$, where $l_k$ is the length of path $p^k$, thus in the worst case, the complexity is $O(n(m+nlogn))$, i.e., the complexity of nextPath($v_s$,$v_t$) routine. Since K paths are to be found, the overall complexity of KSP is $O(Kn(m+nlogn))$ theoretically. End of proof.

## 4.2. The Node-Constrained Paths Discovery

In SDN environment, paths are sometimes constrained by nodes that must be traversed. For example, in network congestion situation, the source routing might designate a different path to traverse other less-busy nodes to avoid congestion, given that the QoS information can be collected in the way we introduced in Section 2. It is also possible that source routing designates another path to traverse other nodes to achieve load balancing. We call these aforementioned nodes the "via nodes" in general. However, basic KSP cannot find top-K paths with via nodes. Instead, KSP only finds unconstrained paths from source to target ordered by path costs. Therefore, the basic KSP must be extended to support paths discovery where via nodes are required.

### 4.2.1. The Model

**Definition 2** The Node-constrained Source Routing Paths Discovery: The source node $v_s$, destination node $v_t$ and intermediate nodes $viaList = \langle v'_1, v'_2, \dots, v'_V \rangle$ are determined by the user. Given a positive integer K, the node-constrained source routing paths discovery finds set $P^K = \{p^1, p^2, \dots, p^K\} \subseteq P_{st}$, where $P_{st}$ is the set containing all the paths from $v_s$ to $v_t$, such that:

- $seg_i = \langle v'_i, u^i_1, u^i_2, \dots, u^i_U, v'_{i+1} \rangle$, $i \in \{1,2,\dots,V-1\}$, $U \in \{0,1,\dots,n\}$, i.e., there exists at least one segment between 2 consecutive via nodes. Note that segment can be the length of 0, i.e., $U = 0$, indicating that via nodes can be directly connected.
- $p^k = \langle v_s, seg_1, seg_2, \dots, seg_{V-1}, v_t \rangle$, $\forall k \in \{1,2,\dots,K\}$, i.e., a path must traverse all the via nodes.
- $\forall k \in \{1,2,\dots,K\}$, $p^k$ is loopless;
- $\forall i,j \in \{1,2,\dots,K\}$, $p^i \neq p^j$, i.e., no duplicated path;
- $\forall k \in \{1,2,\dots,K-1\}$, $cost(p^k) \leq cost(p^{k+1})$, i.e., $p^k$ is found before $p^{k+1}$ (shorter paths are found first).
- $\forall p \in P_{st} - P^K$, $cost(p^K) \leq cost(p)$, i.e., top-K least-costly paths are to be found.

### 4.2.2. The Algorithm

The reason for the lack of node-constraints of KSP is that it conducts the deviation method purely based on nodes. Thus, we propose vKSP (via nodes supported KSP) algorithm which conducts deviation based on "segments" (i.e., sub-paths between consecutive via nodes) to guarantee via nodes to be traversed orderly.

The vKSP algorithm is shown in Algorithm 3. This algorithm takes as the input the nodes that must be traversed as viaList. For easier processing, source $v_s$ and target $v_t$ are concatenated at the head and tail of viaList to form a chain. To get the shortest path that traverses all the via nodes in this chain, the algorithm gets the shortest path for every i-th and (i+1)-th nodes pair along the chain. These are the shortest segments that constitute the shortest path and are put into the set , $P_i^K$ i.e., the shortest segments from $v_i$ to $v_{t+1}$ in the chain. Also, the second shortest segments from $v_i$ to $v_{t+1}$ are calculated for later iterations. The above is the initialization of the algorithm, shown in lines 3–8.

Concatenating all the new segments in $P_i^K$, $P_{i+1}^K$, $\cdots$, generates a new batch of feasible paths from $v_s$ to $v_t$ which are put into $P^C$. The shortest one from $P^C$ is the next shortest path (shown in lines 15 and 18). The next problem is to determine from which node to generate the next segment so that the next batch paths can be found. Since the algorithm invokes nextPath() routine, it is guaranteed the last one in $P_i^K$ is the longest of currently found shortest segments from $v_i$ to $v_{t+1}$. All the $P_i^K$.getLast() are compared to pick the shortest segment, then the corresponding $v_i$ is the node to derive new segments, i.e., the deviation node (shown in lines 16 and 19). In this way, the algorithm does not have to derive new segments from every node in the next iteration; it only derives new segment from deviation node, thus time complexity is minimized. The above is the main loop of the algorithm, shown in lines 13–20.

**Algorithm 3.** vKSP($v_s$, $v_t$, K, viaList)

***Input***: $v_s$: *the source node*
$v_t$: *the destination node*
$K$: *the number of shortest paths*
$viaList$: *intermediate entities must be traversed*
***Output***: $P^K$: *the K shortest paths set*

```
1:  P^K = ∅;                                   //P^K: shortest paths set.
2:  P^C = ∅;                                   //P^C: shortest path candidates set
3:  chain = concatFullChain(v_s, viaList, v_t);
4:  for i = 1; (i + 1) ≤ chain.length; i++ do
5:      sPart = dsp(v_i,v_{i+1}) → P_i^K;
6:      p^1 += sPart;
7:      nextPath(v_i,v_{i+1}) → P_i^K
8:  end for
9:  if p^1 == NULL then
10:     return P^K;                    //no connectivity from source to target
11: end if
12:   P^K = P^K ∪ {p^1};
13:   while ||P^K|| < K do
14:       for i = 1; (i + 1) ≤ chain.length; i++ do
15:           concatNewPaths(P_i^K) → P^C;
```

| | |
|---|---|
| *16:* | *let I be index of: min($P_i^K$.getLast());* |
| *17:* | ***end for*** |
| *18:* | *curPath = min($P^C$) → $P^K$;* |
| *19:* | *nextPath($v_I$,$v_{I+1}$) → $P_I^K$;* |
| *20:* | ***end while*** |
| *21:* | *return $P^K$;* |

### 4.2.3. Proof and Analysis

In this section, we analyze the complexity of the algorithm.

**Theorem 2**: The complexity of vKSP algorithm is: $O((V+K)n(m+nlogn))$. $n$ is the number of nodes, $m$ is the number of the links, $V$ is the number of the via nodes that must be traversed and $K$ is the number of paths to be found.

**Proof**: The complexity of finding the next shortest path is $O(n(m+nlogn))$ (see Theorem 1). In the initialization, the second shortest segments must be found for every via node. Thus, the complexity is $O(Vn(\mathrm{m}+nlogn))$. For every iteration in the main loop, the next shortest search is conducted for the deviation node only, thus the complexity is $O((V+K)n(m+nlogn))$.

## 5. Implementation and Experiments

Our scheme is implemented in the Floodlight [21] controller and Open vSwitch (OVS) [18]. The experiments are conducted in the Mininet [22] emulation environment together with our modified Floodlight controller and OVS switches. The experiment environment is as follows: Intel Core i7-6700 3.4 GHz, 8 GB RAM, Debian 8 64 bit.

### 5.1. Experiments on QoS over LLDP

The QoS over LLDP is tested in a simple topology (see **Fig. 3**) where a video streaming application is deployed. Host h1 is the video server, hosting a video (about 500 MB in size and 15 min in length) and h3 is the client, streaming the video from h1 using Firefox browser. During the streaming, QoS status is constantly changing in bandwidth, delay, etc., due to resource consumption.

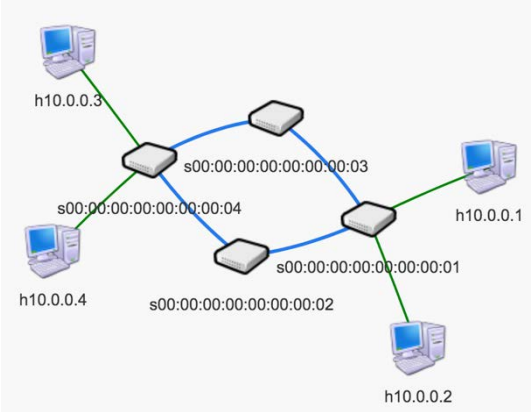



**Fig. 3.** A Simple Topology for Video Streaming Experiment

### 5.1.1. Capabilities Test

We tested the capabilities of QoS over LLDP at first, i.e., whether it is able to successfully and timely collects the QoS information. This experiment is taken in the topology present in **Fig. 3** when video is streamed from the video server h1 to the client h3. The initial QoS properties of each switch port are set as follows: 5 Mbit bandwidth and 2,500 us delay. While the video is being steadily streamed from h1 to h3, QoS properties are changing due to resource consumption, which is typically indicated by curves with fluctuations. **Fig. 4** is a snapshot of the QoS over LLDP frontend GUI. It gives the real-time statistics of various QoS properties of port s1-eth4 (i.e., the fourth Ethernet interface eth4 of switch s1), namely bandwidth (shown in **Fig. 4** (a)), delay (**Fig. 4** (b)), jitter (**Fig. 4** (c)) and packet loss (**Fig. 4** (d)), which proves that QoS over LLDP scheme can effectively collect the changing QoS values according to preset intervals. The default interval for LLDP is 15 seconds. The interval for our QoS over LLDP can be tuned according to precision requirement for QoS monitoring, higher or lower. The x-axis uses real time as the scale, and the precision is 1 second in this experiment to intuitively demonstrate QoS over LLDP's capability of QoS information collecting. We can see from **Fig. 4** that QoS properties can be accurately collected and rendered in the frontend. Also, basic intuitions can be drawn from these QoS property curves. For example, those measured packet losses indicate a high utilization during video streaming; the difference between the preset bandwitch (5 Mbit) and the real-time available bandwidth shown in **Fig. 4** (a) indicates the bandwidth comsumption for the video streaming. QoS over LLDP is also capable of collecting QoS information in more complex topology to conduct QoS-aware decision making such as top-K QoS-aware path discovery (see Section 5.2).

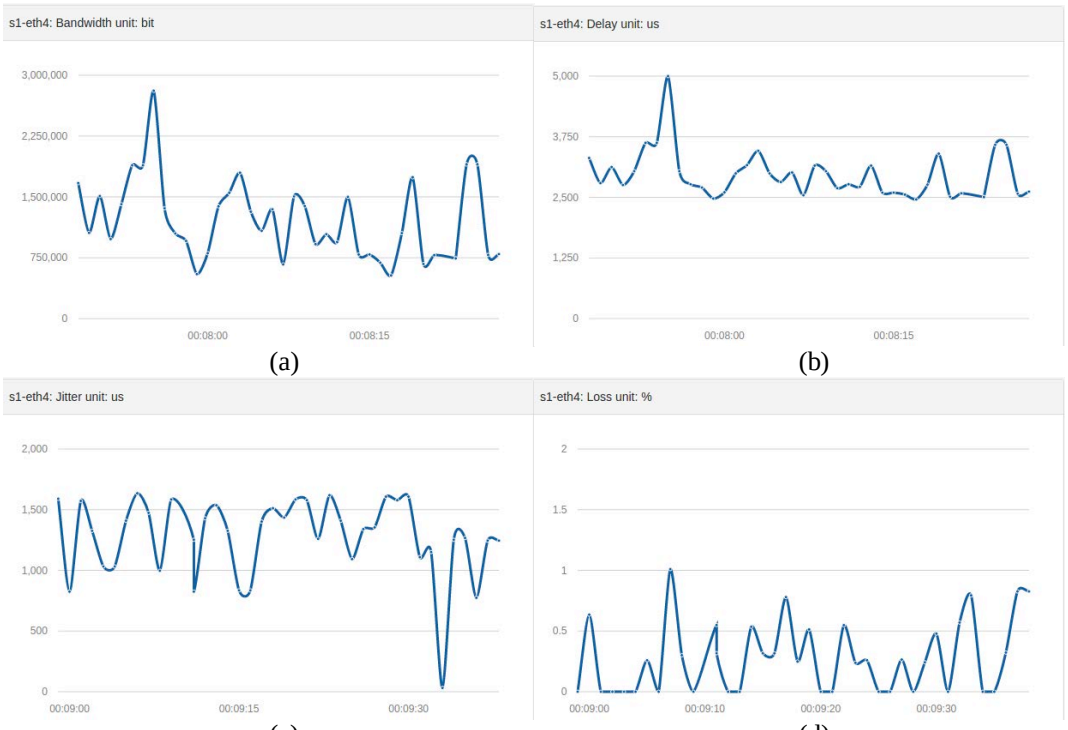



**Fig. 4.** QoS over LLDP Frontend GUI

### 5.1.2. Traffic Analysis

In order to evaluate the impact on network traffic caused by QoS over LLDP, we capture traffic using Wireshark in two scenarios (video streaming vs. no video streaming) with the above simple topology. The evaluation duration is 15 min. The evaluation results are shown in **Table 3**. We first analyze "no video streaming" scenario. As we stated above, QoS over LLDP causes extra network overhead since it contains several QoS TLV bytes. However, the percentages of QoS over LLDP (6.56%) is just slightly greater than pure LLDP (5.27%) by bytes, meaning that QoS over LLDP does not deteriorate the network traffic performance. For video streaming scenario, QoS over LLDP and pure LLDP take 0.021% and 0.015% of the total traffic by bytes, respectively, and 0.59% and 0.58% of the total traffic by packets, respectively. This indicates that QoS over LLDP works in a non-invasive and piggyback fashion with very minor traffic overhead to achieve QoS information collecting. The experiment results indicate that QoS over LLDP is an applicable approach for QoS information collecting in an SDN environment.

**Table 3.** QoS over LLDP vs. LLDP in different scenarios, duration about 15 min

| Scenario | Scheme | Total Packets | LLDP Packets | Total Bytes | LLDP Bytes |
|---|---|---|---|---|---|
| **No Video Streaming** | **Pure LLDP** | 20173 | 2567 (12.72%) | 6526386 | 344222 (5.27%) |
| | **QoS over LLDP** | 21889 | 2563 (11.71%) | 6738255 | 441760 (6.56%) |
| **Video Streaming** | **Pure LLDP** | 440348 | 2550 (0.58%) | 2208747286 | 339866 (0.015%) |
| | **QoS over LLDP** | 437795 | 2580 (0.59%) | 2086069045 | 443204 (0.021%) |

## 5.2. Experiments on Paths Discovery

In this section, we experimented our proposed top-K QoS-aware paths discovery based on KSP/vKSP. Our scheme is compared with the path discovery method adopted in SlickFlow [3]. SlickFlow is a fault recovery scheme for SDN-enabled data center networks. It features in that path discovery is conducted in the controller based on the LLDP-collected topology information, as does our system. SlickFlow enumerates all paths and select top-K paths for the purpose of fault recovery, i.e., the all-routes scheme, which is quite different from the KSP/vKSP. KSP/vKSP does not enumerate all paths; on the contrary, it merely finds the top-K paths using deviation method. This is where performance comparison can be made, given two different algorithms serving the same purpose of paths discovery. Discussion about SlickFlow can also be found in Section 2.

Experiments are carried out under the mesh topology with different switch numbers and switch out degrees (i.e., how many switches are connected from the current switch). Mesh topologies offer plenty of alternative paths, which are suitable to test paths discovery performance in complex networks. QoS over LLDP is adopted to collect underlying QoS information to calculate individual/overall path costs for path ranking (detailed conversion algorithms can be found in Section 3.2). QoS properties of every switch port is uniformly distributed according to the following configurations: $bw \in [80Mbps, 100Mpbs], dl \in [1us, 10us], jt \in [0us, 2us], pl \in [0,0.1]$. **Fig. 5** exhibits a sample mesh topology with 10 switches and 3 out degrees for each switch. The experiment objective is to test the path discovery performance in terms of query time. The experiment procedure is to test how much time is required to conduct the path discovery from host h1 to h9 in different topologies

through node-constrained (via switches s2, s4 and s6) or unconstrained methods. KSP, vKSP and SlickFlow are tested under the same settings.

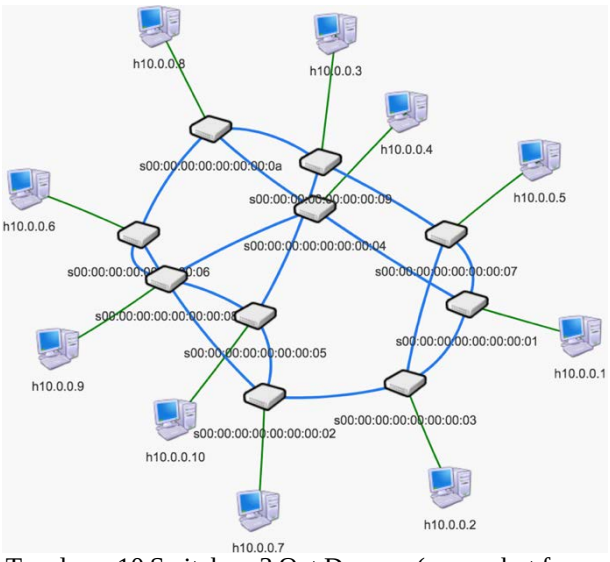



**Fig. 5.** A Sample Mesh Topology, 10 Switches, 3 Out Degrees (a snapshot from our modified Floodlight controller)

**Fig. 6** is the experiment results taken under a mesh topology of 10 switches and varying out degrees. "UC" in the legend indicates the unconstrained path discovery, i.e., finding paths without considering intermediate node constraints, marked as dashed lines. "-3via" in the legend means 3 via nodes are considered during path discovery, i.e., the node-constrained path discovery, marked as solid lines. For SlickFlow, only one path is to be found due to the performance issue, while for KSP and vKSP, several paths are to be found, indicated by "-xpaths" in the legend.

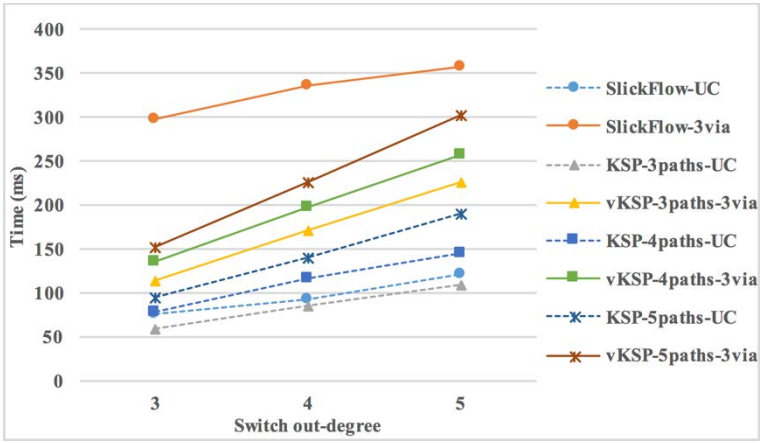



**Fig. 6.** Experiment under 10 Switches

We explain experiment results in later sections with the specific 5 out degree. **Fig. 6** shows that the unconstrained SlickFlow-UC (155 ms) performed well. However, KSP-3paths-UC (116 ms) still outperformed SlickFlow-UC even though 2 more paths were found. The reason

for SlickFlow's deficiency is that it adopts the all routes method where all the possible routes from source to target are enumerated and the best one is then selected. This method is far from optimized. We can see from the figure, KSP's performance scales linearly with path numbers as proven in previous sections. However, as the path number increases, more time is required for path discovery thus large K (i.e., the path number) is not recommended in large topologies. Usually, 3-paths is a desirable tradeoff between performance and the purpose of alternative paths. Besides, SlickFlow does not consider QoS, thus paths are purely ranked by length. On the other hand, KSP proposed in this paper is a top-K QoS-aware path discovery scheme where paths are found and ranked by QoS costs as explained in Section 3 and 4. **Table 4**, as an example, demonstrates the end-to-end QoS properties of the 3 paths found by KSP-3paths-UC. These paths are found based on the bandwidth cost (see **Table 1** for computation details) shown in the last column of **Table 4**. Lower bandwidth costs indicate better performance in bandwidth as shown in **Table 4**, which, however, do not necessarily indicate better performance in other QoS properties. We can see from this table, Path 1 offers widest bandwidth (91.2 Mpbs) while it does not provide best jitter performance (3.2 us vs. 1.7us). For different applications with different QoS preferrences, other costs (e.g., delay cost, jitter cost, packet loss cost) can be used, as explained in Section 3.

vKSP is capable of finding node-constrained paths. We can see from the figure, vKSP's performance scales linearly with the sum of path numbers and intermediate node numbers as proven in previous sections. Therefore, large intermediate node numbers are not recommended, as are not for large path numbers. Still, vKSP-5paths-3via offers good performance (300 ms) compared with SlickFlow-3via (357 ms) even though more paths are found by vKSP.

**Table 4.** End-to-end QoS Properties of Found Paths

| Path | Path Details | BW (Mbps) | DL (us) | JT (us) | PL | BW Cost |
|---|---|---|---|---|---|---|
| Path 1 | h1-s1-s10-s9-h9 | 91.2 | 17.8 | 3.2 | 0.18% | 2.19 |
| Path 2 | h1-s1-s3-s9-h9 | 83.6 | 19.3 | 1.7 | 0.24% | 2.26 |
| Path 3 | h1-s1-s3-s10-s9-h9 | 80.7 | 28.7 | 4.9 | 0.33% | 3.40 |

**Fig. 7** is the experiment results taken under a mesh topology of 20 switches and varying out degrees. Similar analysis applies in this experiment setting as well. We can see that more complex topologies have obvious performance impact. However, KSP and vKSP still offers good performance as proven before. The 5-path node-constrained vKSP-5paths-3via (225 ms) gives better performance than the single-path unconstrained SlickFlow-UC (406 ms). Another difference between KSP/vKSP and SlickFlow is that KSP/vKSP find optimal paths based on path costs which reflect the Quality of Service while SlickFlow does not consider QoS during path discovery. In a word, our scheme offers applicable performance in the top-K QoS-aware paths discovery for source routing in SDN networks.

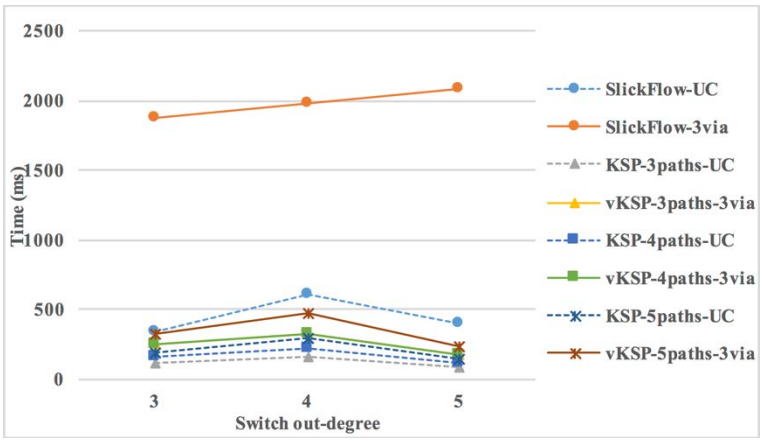



**Fig. 7.** Experiment under 20 Switches

## 6. Conclusion

The SDN paradigm offers more application-centric possibilities for source routing. This paper augments source routing path discovery with QoS-awareness and alternative paths finding in the context of SDN environment. We designed the non-invasive piggyback QoS information collecting scheme where no fundamental modification is required. We also derived the KSP variations to find the top-K ranked alternative paths, in case of load balancing or backup purpose. The experiments exhibit the functionalities and performance of our scheme. The performance of our scheme is applicable in QoS-sensitive application scenarios compared with previous works, as the experiment results showed. In the future work, our scheme is going to be applied in more specific subject of SDN such Service Function Chaining, etc., where source routing is a fundamental aspect.

## Acknowledgment

This paper is supported by the Science & Technology Department of Sichuan Province (Grant No. 2016RZ0065), the Education Department of Sichuan Province (Grant No. 15ZA0396), Fundamental Research Funds for the Central Universities, Southwest Minzu University (Grant No. 2018NQN56) and Southwest Minzu University Education and Teaching Reform Project (Grant No. 2017ZC19).

## References

[1] D. A. Maltz and D. C. Johnson, "The Dynamic Source Routing Protocol (DSR) for Mobile Ad Hoc Networks for IPv4," *IETF RFC*, no. 4728, Feb. 2007. Article (CrossRef Link).

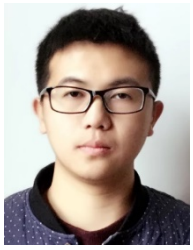
[2] N. Mckeown *et al.*, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008. Article (CrossRef Link).

[3] R. M. Ramos, M. Martinello, and C. E. Rothenberg, "SlickFlow: Resilient source routing in Data Center Networks unlocked by OpenFlow," in *Proc. of IEEE Conference on Local Computer Networks (LCN)*, 2013, pp. 606–613. Article (CrossRef Link).

[4] A. Abujoda, H. R. Kouchaksaraei, and P. Papadimitriou, "SDN-Based Source Routing for Scalable Service Chaining in Datacenters," in *Proc. of 14th IFIP International Conference on Wired/Wireless Internet Communications (WWIC)*, pp. 66–77, 2016. Article (CrossRef Link).

[5] X. Wang, C. Huang, X. Fan, and K. He, "Flow Table Update Method based on Load Balancing for SDN," *J. Huazhong Univ. Sci. Technol. Nat. Sci. Ed.*, vol. 44, no. 11, pp. 75–81, 2016. Article (CrossRef Link).
[6] M. Soliman, B. Nandy, I. Lambadaris, and P. Ashwood-Smith, "Exploring source routed forwarding in SDN-based WANs," in *Proc. of IEEE International Conference on Communications (ICC)*, pp. 3070–3075, 2014. Article (CrossRef Link).
[7] M. Soliman, B. Nandy, I. Lambadaris, and P. Ashwood-Smith, "Source routed forwarding with software defined control, considerations and implications," in *Proc. of ACM Conference on Emerging Networking Experiments and Technology (CONEXT) Student Workshop*, pp. 43–44, 2012. Article (CrossRef Link).
[8] Y. Zhang, N. Beheshti, L. Beliveau, and G. Lefebvre, "StEERING: A software-defined networking for inline service chaining," in *Proc. of IEEE International Conference on Network Protocols (ICNP)*, pp. 1–10, 2013. Article (CrossRef Link).
[9] Z. A. Qazi, C. C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "SIMPLE-fying middlebox policy enforcement using SDN," *Acm Sigcomm Comput. Commun. Rev.*, vol. 43, no. 4, pp. 27–38, 2013. Article (CrossRef Link).
[10] Z. Qazi, C.-C. Tu, R. Miao, L. Chiang, V. Sekar, and M. Yu, "Practical and Incremental Convergence between SDN and Middleboxes," *Open Network Summit*, pp. 1–15, 2013. Article (CrossRef Link).
[11] G. T. K. Nguyen, R. Agarwal, J. Liu, M. Caesar, P. B. Godfrey, and S. Shenker, "Slick packets," *Acm Sigmetrics Perform. Eval. Rev.*, vol. 39, no. 1, pp. 205–216, 2012. Article (CrossRef Link).
[12] C. Guo *et al.*, "SecondNet: a data center network virtualization architecture with bandwidth guarantees," in *Proc. of ACM Conference on Emerging Networking Experiments and Technology (CONEXT)*, p. 15, 2010. Article (CrossRef Link).
[13] C. E. Rothenberg, C. A. B. Macapuna, F. L. Verdi, M. F. Magalhães, and A. Zahemszky, "Data center networking with in-packet Bloom filters," in *Proc. of XXVIII Simp´osio Brasileiro de Redes de Computadores e Sistemas Distribu´ıdos*, pp. 553–566, 2010. Article (CrossRef Link).
[14] M. Arumaithurai, J. Chen, E. Monticelli, X. Fu, and K. K. Ramakrishnan, "Exploiting ICN for Flexible Management of Software-defined Networks," in *Proc. of ACM Conference on Information-Centric Networking (ACM-ICN)*, Paris, France, pp. 107–116, 2014. Article (CrossRef Link).
[15] J. M. Halpern and C. Pignataro, "Service Function Chaining (SFC) Architecture," *IETF RFC*, no. RFC 7665, Oct. 2015. Article (CrossRef Link).
[16] T. Nadeau and P. Quinn, "Problem Statement for Service Function Chaining," *IETF RFC*, no. RFC 7498, Apr. 2015. Article (CrossRef Link).
[17] S. G. Kulkarni *et al.*, "Name enhanced SDN framework for service function chaining of elastic Network functions," in *Proc. of 2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pp. 45–46, 2016. Article (CrossRef Link).
[18] B. Pfaff *et al.*, "The design and implementation of open vSwitch," in *Proc. of 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Oakland, CA, 2015, vol. 40, pp. 117–130. Article (CrossRef Link).
[19] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numer. Math.*, vol. 1, no. 1, pp. 269–271, 1959. Article (CrossRef Link).
[20] E. Q. V. Martins and M. M. B. Pascoal, "A new implementation of Yen's ranking loopless paths algorithm," *4OR*, vol. 1, no. 2, pp. 121–133, 2003. Article (CrossRef Link).
[21] "Home - Project Floodlight - OpenFlow news and projects," *Project Floodlight*. [Online]. Available: http://www.projectfloodlight.org/. [Accessed: 22-May-2017]. Article (CrossRef Link).
[22] "Mininet: An Instant Virtual Network on your Laptop (or other PC) - Mininet." [Online]. Available: http://mininet.org/. [Accessed: 22-May-2017]. Article (CrossRef Link).

**Xi Chen** (1985-) received his B.S. and Ph.D. degrees in Computer Science from Southwest Jiaotong University, China, in 2007 and 2013, respectively. During 2011-2012, he studied as a visiting and joint Ph.D. candidate in School of Computer Science and Engineering, The University of New South Wales (UNSW), Sydney, Australia. He is currently a lecturer in the School of Computer Science and Technology, Southwest Minzu University, China. His research interests include Software-Defined Networking (SDN), Web Services, Computer Networks, etc.



**Junlei Wu** (1994-) received his B.S. degree in the School of Computer Science and Technology, Southwest Minzu University, China, in 2017. He is currently pursuing his M.S. degree in the School of Computer Science and Technology, Hangzhou Dianzi University, China. His research interests include Software-Defined Networking (SDN), Computer Networks, etc.

**Tao Wu** (1984-) received her B.S. and Ph.D. degrees in Computer Science from Southwest Jiaotong University, China, in 2007 and 2014, respectively. She is currently a lecturer in the School of Computer Science, Chengdu University of Information Technology, China. Her research interests include Particle Swarm Optimization (PSO), high performance computing, etc.