Comparison of Routing Algorithms with Static and Dynamic Link Cost in SDN - Extended Version

Erdal Akin, Turgay Korkmaz
University of Texas at San Antonio, Texas, USA
Email: erdal.akin@utsa.edu, korkmaz@cs.utsa.edu

***Abstract*—In this paper, we compared the existing routing algorithms in the context of Software Defined Networking (*SDN*), where a logically centralized controller obtains the global view of the network, makes the routing decisions, and installs them to the switches in the selected paths. We considered three categories of routing algorithms with static link cost (*RA-SLC*), dynamic link cost (*RA-DLC*) and dynamic link cost with minimum interference (*RA-DLCMI*). We then implemented the routing algorithms from each category using *RYU SDN* controller and tested them on *Mininet* emulator. In our evaluations, we considered both the idealistic case assuming the availability of accurate network state information (NSI) and the practical case where the NSI is collected periodically and has some inaccuracy. Our experimental results showed that both *RA-DLC* and *RA-DLCMI* outperformed *RA-SLC* in terms of throughput and the number of accepted flows while there was no significant difference between *RA-DLC* and *RA-DLCMI*. We also observed that periodic collections caused inaccuracies in NSI negatively affected the performance of all algorithms. Further research is needed to effectively obtain accurate NSI.**

***Index Terms*—Routing Algorithms; Minimum interference routing; SDN; Software Defined Networking; Traffic Engineering.**

I. Introduction

Traditional IP Network was originally designed as a simple but highly fault-tolerant distributed system that can provide best-effort communications services in a trusted environment. However, these key assumptions behind the original design did not match up with the real-world situations as the Internet became the part of daily life. Accordingly, many new features had to be added. Unfortunately, since it was almost impossible to replace or redesign the underlying architecture [26], [36], [17], many new features were simply patched on top of the original architecture, resulting in one of the largest and complex distributed system to manage.

To simplify network management while creating a flexible architecture that can seamlessly integrate new features into the network, researchers have introduced the Software-Defined Networking *(SDN)* as a new paradigm [29], [40]. In essence, SDN separates the *control plane* from *data plane*, and builds up a logically centralized controller, which obtains the global view of the underlying network and makes all control plane decisions (e.g., which path to use) [13], [30], [19]. The controller then simply conveys these decisions to the underlying routers and switches via a well-defined *API* such as *OpenFlow* [21]. This centralized control plane greatly simplifies network configuration and new policy installation, particularly when compared to the traditional distributed control plane where each router/switch has to figure out what to do through exchanging massages with each other.

Realizing the potential benefits of SDN, the industry has also shown significant interest in developing and deploying SDN-based technologies. For example, many commercial switches/routers have now support for *OpenFlow API* [13]. An *OpenFlow switch* is simply a forwarding element (such as a router, switch, firewall, etc.) that forwards incoming packets based on the routing decisions that are determined and installed by SDN controllers [30], [26], [13].

One of the key challenges in SDN is how to efficiently *route* the given flows through the underlying network. To achieve this, the centralized controller has to (i) obtain the global view of the underlying network, (ii) compute the feasible and/or optimal paths for the given flows by taking into account the flow demands and the current network state information, and (iii) install the new routing rules to the forwarding elements (*SDN switches*) in data plane. The third step is simply achieved by using an open communication protocol (e.g., OpenFlow [30]). Therefore, we mainly focus on the first two steps. More specifically, we present an OpenFlow-based implementation of various existing routing algorithms and compare their performances on two different network topologies with two methods for obtaining network state information. For the implementation and testing, we used *RYU* controller [38] and *Mininet* emulator [27].

The existing routing algorithms use either *static link*

cost (e.g., hop count, distance, link capacity) or *dynamic link cost* (e.g., available link capacity, link utilization). Researchers have further improved the routing algorithms using dynamic link cost with other metrics (e.g., number of flows on a link) to *minimize the interference* among the flows. Accordingly, we divided the routing algorithms into three categories and specifically implemented the followings to compare:

- RA-SLC: Routing Algorithms with *Static Link Cost* include Minimum Hop Algorithm (*MHA*), Shortest Path Algorithm (*SP*) and Widest Shortest Path Algorithm (*WSP*) [15], [34], [22].
- RA-DLC: Routing Algorithms with *Dynamic Link Cost* include Constraint Shortest Path First (i.e. Dynamic Shortest Path (*DSP*)), Dynamic Widest-Shortest Path Algorithm (*DWSP*).
- RA-DLCMI: Routing Algorithms with *Dynamic Link Cost and Minimum Interference* include Minimum Interference Routing Algorithm (*MIRA*), the Least Interference Optimization Algorithm (*LIOA*) and the Improved Least Interference Routing Algorithm (*ILIOA*) [12], [22], [8], [33].

These algorithms have been proposed with the assumption that the controller has the accurate network state information. To achieve this, we used a shadow topology, which can represent accurate state information in the case of a single controller. However, in practice, because of multiple controllers and background traffic, the controllers need to obtain the state information from the SDN switches by querying them. In that case, as we discuss later on, it is very challenging to obtain the accurate state information. We show that this negatively impacts the performance. Therefore, the future algorithms need to take into account the inaccuracies of network state information (NSI).

In our tests, we observed that *RA-DLC* and *RA-DLCMI* outperform *RA-SLC* as expected. We also observed that there is no significant difference between the existing algorithms using dynamic link cost. These general trends were the same under both accurate NSI and periodic NSI. However, under periodically collected NSI, since the controller used the inaccurate NSI, all algorithms have accepted more flows than that the network can carry. Unfortunately, accepting more requests has caused packet loss for every flow. Clearly, this is not desirable for the sake of Quality of Service (*QoS*), even though total transfered bandwidth is high. We will discuss more results in Section V.

The rest of the paper is organized as follows. In Section II, we give background information about *SDN*. We present existing routing algorithms in Section III. In Section IV, we explain how to mimic the availability of accurate NSI while also discussing how to periodically collect NSI in practice. In Section V, we present our experimentation set up and discuss our results. Finally, we conclude the paper by stating challenges and future works in Section VI.

II. Background of Software Defined Networking

A. What is Software Defined Networking?

Software-Defined Networking is a new network architecture that decouples the network's control logic (the Control Plane) and forwarding logic (the Data Plane) [26]. After this separation, routers and switches become simple forwarding devices while the central controller becomes the brain of the network. As illustrated in Figure 1, this is in sharp contrast to the traditional IP network where the control logic and forwarding logic are coupled and distributed.

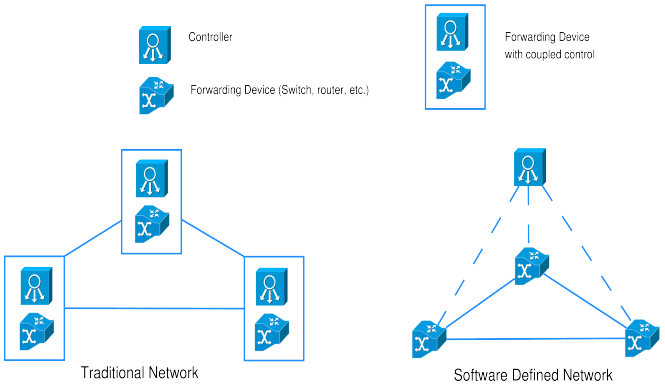


Fig. 1: Comparing Traditional Network and SDN.

The main components and the general architecture of SDN are shown in Figure 2. From this Figure, we can see that SDN has the following four key innovations over the traditional networks:

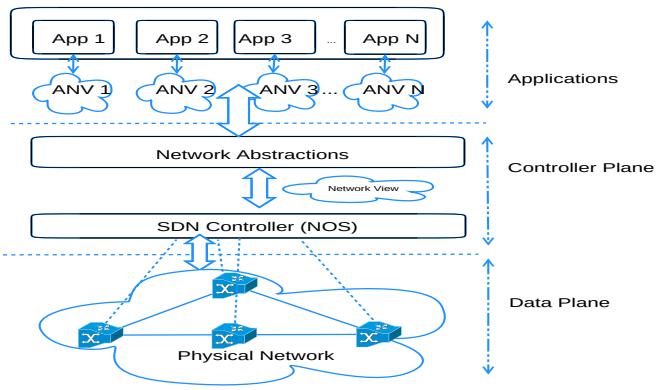


Fig. 2: Software Defined Network Architecture.

- *SDN* removes the control logics from network devices (e.g., routers and switches). So the network devices have become simple forwarding elements.
- *SDN* puts control logic in an external and central unit called *SDN* controller or Network Operating System *(NOS)*, which is a software that runs on a server. It obtains the global view of the network and make necessary routing decisions for the simple forwarding devices.
- *SDN* makes routing decisions *per* flow[1] while traditional IP routing is destination based. SDN then places identical policies on the forwarding devices in the determined path so that each packet of the flow receives consistent services [20]. This flow-based programming capability of SDN provides extraordinary flexibility in managing the network [30].
- *SDN* makes it easy to develop several new services by running new applications on top of *SDN* controller. Such applications simply get the abstract network view (*ANV*) from the *SDN* controller. Upon making the necessary policy decisions, they interact with the underlying data plane through *SDN* controller that maps higher-level policy decisions to lower-level forwarding decisions. This is considered as the most important innovation of *SDN*.

B. Routing framework in SDN

The SDN controller has to perform three common routing tasks: (i) discovering network topology and obtaining accurate link-state information, (ii) computing feasible and/or optimal paths for the given flows, and (iii) installing the necessary forwarding rules on each SDN switches over the determined path using *OpenFlow*.

We will now briefly discuss these three tasks. We will then mainly focus on the first two tasks in the rest of the paper and experimentally evaluate the performance of various path selection algorithms with two different methods of obtaining the link-state information.

1) Acquiring the global view of the network: To make routing decisions and compute new paths, the SDN controller first needs to obtain the *accurate* global view of the underlying network. Since management capability is removed from forwarding elements, they do not have any information about the whole network while this was the case in traditional routing protocols (e.g., OSPF). Therefore, the SDN controller is responsible for discovering network topology and obtaining the accurate link-state information from the underlying switches.

[1]A flow can be defined as a sequence of packets from a source to a destination.

After getting the list of switches and links, the SDN controller can query the underlying switches and obtain the link-state information about each link. Since the *static* link-state metrics do not change, the SDN controllers can simply obtain these metrics once when using topology discovery protocols [19], [28]. However, since the *dynamic* link-state metrics (e.g., available bandwidth, utilization, delay, jitter etc.) do often change, the SDN controller needs to query the switches and gets the sate of every link at very short intervals to maintain an accurate view. Unfortunately, this will create a significant overhead, particularly when we need high accuracy in the link-state information, which is necessary to better utilize the underlying network resources while meeting the flow demands and avoiding congestions.

So one of the challenging issues here is *how to achieve the accurate acquisition of the dynamic link-state information while minimizing the overhead on the controller and in the network.* To address this challenge, more research is needed. However, when comparing the existing routing algorithms that assume the availability of accurate state information, we will mimic the accuracy by maintaining a shadow of the network at the controller and update it after each path selection. But this cannot be implemented in practice because the involvement of multiple controllers, elastic background traffic, and excessive delays etc. Therefore, as done in practice, we will also consider periodic link-state collection approach with three different intervals and show their impacts on the routing algorithms. We present the details of our link-state representation and collection mechanisms in Section IV.

2) Computing feasible and/or optimal paths: After discovering topology and obtaining the link-state information of each link, the SDN controller computes a feasible and/or optimal path for a given flow based on the routing algorithms that we discuss in Section III.

All the existing algorithms assume that the accurate network state information is available; thus, represented as a directed graph $G = (V, E)$, where V is the set of nodes/switches and E is the set of links. Let $n = |V|$ be the number of nodes and $m = |E|$ be the number of edges in the network. Each link $(u, v) \in E$ is associated with a link cost $C(u, v)$ based on the obtained static or dynamic link-state information. As a matter of fact, most of the existing algorithms that we compare differ in how they determine $C(u, v)$, as we further discuss in Section III. After determining $C(u, v)$ and eliminating the links that do not satisfy the requested demand, most of the existing algorithms mainly use Dijkstra's shortest path algorithm or its modified versions to find a path.

3) Installing the forwarding rules: Upon computing a path, the SDN controller installs the necessary rules on the forwarding table of each switch using *OpenFlow*. One key question here is how and who starts the path computation and the rule installment process. These can be done in a proactive or reactive manner. It is also possible to use a hybrid approach. We can explain these three modes of operations as follows:

Reactive Mode:
When a switch receives a request that does not match with the existing rules in its forwarding table, it considers this as a new flow and creates an Open Flow Protocol (*OFP*) packet-in message for the first packet of the new flow. It then sends this message to the controller for a decision. Accordingly, the controller creates a rule and sends it to the switch via OpenFlow. The switch uses this rule to forward future packets. Even though this approach uses existing flow table memory more efficiently, unfortunately, it causes performance delay because of continuous communication between forwarding elements and the controller for the first packet of each new flow. This delay can be ignored in small scale networks; but, it is a significant burden for geographically remote controllers, short-lived flows, general purpose CPU and/or vSwitches [9], [32], [3], [4].

Proactive Mode:
The SDN controller determines forwarding rules for possible traffic in advance and installs the necessary rules to the switches ahead of time. So when a new flow arrives, the switches can forward its packets without consulting with the controller. Accordingly, the switches forward all the packets at line rate [9], [32], [3], [4], [42]. Moreover, since all flows find a match in the switches and avoid consultation with the controller, this mode significantly reduces the burden on the controller. The disadvantage of this approach is that the flow tables should be coarse-grained because of scalability issues. To meet fine-grained control needs, we can use the hybrid approach discussed next.

Hybrid Mode:
The controller and switches can use the combination of reactive and proactive modes. Accordingly, we can get the flexibility of reactive mode in providing fine-grained control while benefiting from proactive mode by eliminating delay and avoiding significant burden on the controller. Based on the network performance, the proactive mode can periodically modify routes and flow tables to improve performance [9], [3], [4].

In the simulations, since we mainly evaluate the routing algorithms based on their performance in finding the paths for given flows, we will use the reactive mode of operation. When a new flow is seen, the source switch will inform the controller. Accordingly the controller will compute an appropriate path and install it through the network.

III. Existing Routing Algorithms

We divide the existing routing algorithms into three groups: *RA-SLC* (Routing Algorithms with *Static* Link Cost), *RA-DLC* (Routing Algorithms with *Dynamic* Link Cost), and *RA-DLCMI* (Routing Algorithms with Dynamic Link Cost and Minimum Interference). Before describing the details of specific algorithms, we would like to first present Algorithm 1 as a general framework to better show the key steps that are performed by almost all *RA-SLC*, *RA-DLC*, and *RA-DLCMI* with some exceptions. For example, *Dynamic Widest Shortest Path (DWSP)* does not perform Step 1. Instead, it directly uses the residual bandwidth of each link along with a modified version of Dijkstra's algorithm to find a path with the largest residual bandwidth in Step 3.

Algorithm 1 General Framework for All Routing Algorithms

Input: Graph $G(V, E)$, vector BW which is the initial bandwidth capacity of the links, vector RBW which is the residual bandwidth of the links, vector I which is the number of flows carried on the links, and a flow request between the pair of nodes (s, d) with the required demand of $r_{(s,d)}$.

Output: p, a path between the pair of nodes (s, d) that satisfies the requested demand of $r_{(s,d)}$. That is $\min\{RBW_{(u,v)} | (u, v) \in p\} \geq r_{(s,d)}$.

1: Based on the proposed heuristic, compute link cost metric $c_{(u,v)}$ for all edges $(u, v) \in E$.
2: Eliminate all links that have residual bandwidth less than requested demand $r_{(s,d)}$.
3: Use *Dijkstra's algorithm* to compute the shortest path p between the pair of nodes (s, d).
4: SDN Controller installs the forwarding rules on the SDN switches in p.
5: SDN switches route the packets of the requested demand $r_{(s,d)}$ through path p.
6: SDN Controller periodically obtains the updated link-state information.

The first two steps are not needed in *RA-SLC* because these algorithms do not take into account the changes in link-state information. Accordingly, *RA-SLC* start with Step 3 and always find the same path for all the flows between a given pair of nodes (s, d).

In contrast, the first two steps (particularly, Step 1) are very important for the success of *RA-DLC* and *RA-DLCMI*. In Step 1, these algorithms compute the dynamic link cost metric based on the proposed heuristics. Therefore, this step is the most crucial one and it is the key difference between the proposed algorithms, as we discuss later in detail. In Step 2, all the algorithms simply eliminate the links whose residual bandwidth is less than the requested demand.

The first two steps constitute the pre-computation phases of *RA-DLC* and *RA-DLCMI*. Since the link-state information changes after installing the forwarding rules for each flow, these pre-computation phase need to be performed for each new flow. However, since it will be very costly to obtain all the changes in link-state information, the SDN controller obtains link-state information in a periodic manner. As we discuss later, this introduces some inaccuracy in the link-state information and may negatively affect the network performance.

The remaining steps are the same for all routing algorithms including *RA-SLC*. In Step 3, all algorithms simply use *Dijkstra's algorithm* to find the least cost (shortest) path based on the dynamic link costs determined in Step 1 (or static link cost obtained once during topology discovery). After finding the shortest path p, the controller installs the necessary forwarding rules on the SDN switches in Step 4. Then, the SDN switches route the packets of the given flow through p in Step 5. Finally, in Step 6, the SDN controller periodically queries the SDN switches and obtain the dynamic link-state information (e.g., residual bandwidth) to use when making routing decisions for the new flows.

In the following three subsections, we describe the key assumptions and the ideas behind the existing routing algorithms that we classified into three groups, as shown in Table I. In addition to summarizing their key characteristics, we examine the advantages and disadvantages of each group of routing algorithms.

A. Routing Algorithms with Static *Link Cost* (RA-SLC)

The SDN controller using *RA-SLC* needs to discover the network topology and determine the static link cost (e.g., hop count, distance, 1/bandwidth) for each link. Since RA-SLC do not take into account link-state information that changes over the time (e.g., link utilization), the controller can simply calculate the static link cost *once* during the topology discovery phase and use it when computing a path for any given flow.

Clearly, using static link cost significantly reduces the message overhead and the pre-computation overhead because the controller does not need to periodically query each switch to obtain the up-to-date link-state information or re-compute the link cost based on the new link-state information. This approach might be effective when the network utilization is low. However, as the network utilization increases, this approach will cause significant unbalance and congestion in the network because it always calculates the same paths for the flows with the same source and destination pairs.

To overcome these problems, researchers have considered *RA-DLC* and *RA-DLCMI* that we discuss next. To better understand the performance improvements of *RA-DLC* and *RA-DLCMI*, we included the following *RA-SLC* in our evaluations.

Minimum Hop Algorithm (MHA) chooses the shortest path that has the minimum number of links between source and destination nodes [22]. Such paths can simply be computed using BSF (Breadth First Search) or *Dijkstra's Algorithm* by setting the cost of each link to 1.

Shortest Path Algorithm (SP) uses *Dijkstra's Algorithm* to choose the shortest path when the cost metric of each link is set to inversely proportional to the bandwidth capacity of that link [15]. Actually, *Open Shortest Path First (OSPF)* also use this same approach [31].

Widest-Shortest Path Algorithm (WSP) tries to find the path with largest bandwidth capacity while minimizing hop-count and can be implemented by using modified version of Dijkstra's Algorithm [34], [23]. Another version known as Shortest-Widest Path algorithm first eliminate all the links that does not support the given demand request and then executes the Dijkstra's shortest path algorithm which selects the links with largest bandwidth capacity in case of cost equality [43].

B. Routing Algorithms with Dynamic *Link Cost* (RA-DLC)

Since *RA-SLC* compute the paths using static link cost, they always find the same paths for all the flows with the same source and destination nodes. This causes congestion on some links while there are alternative links to use. To deal with this problem, the controller needs to periodically obtain information about available network resources and takes that information into account when computing paths. With this in mind, researchers have proposed various routing algorithms with *dynamic* link

Type of Routing Algorithm	Algorithm Name	Cost metric calculation	Computation Complexity	
			Pre-computation per flow	Path Selection Algorithm per flow
RA-SLC	*MHA*	$C_{(u,v)} = 1$	-	Dijkstra's Algorithm
	SP	$C_{(u,v)} = \frac{1}{BW_{(u,v)}}$		
	WSP	$C_{(u,v)} = BW_{(u,v)}$		*Modified Dijkstra's Algorithm*
RA-DLC	*DSP*	$C_{(u,v)} = \frac{1}{RBW_{(u,v)}}$	$\mathcal{O}(m)$ for computing cost metric for each link	*Dijkstra's Algorithm*
	DWSP	$C_{(u,v)} = RBW_{(u,v)}$		*Modified Dijkstra's Algorithm*
RA-DLCMI	*LIOA*	$C_{(u,v)} = \frac{I^{\alpha}_{(u,v)}}{RBW^{\alpha}_{(u,v)}}$		Dijkstra's Algorithm
	ILIOA	$C_{(u,v)} = (1-U_{(u,v)}) * \frac{I^{\beta}}{BW^{\beta}_{(u,v)}} + U_{(u,v)} * \frac{I^{\alpha}}{RBW^{\alpha}_{(u,v)}}$		
	MIRA	Initialize $C_{(u,v)}$ to 0. Using max-flow min-cut algorithm, find min-cut sets (critical links) between the pairs. For each appearance of a critical link (u, v) in the min-cut sets, increase $C_{(u,v)}$ by 1.	$\mathcal{O}(m)$ for computing cost metric for each link + $\mathcal{O}(max - flow \ \ min - cut)$ for finding critical links (e.g., $\mathcal{O}(n^2\sqrt{m})$ with Goldberg-Tarjan highest label pre-flow push algorithm [18])	

In this table, $C_{(u,v)}$ represents cost metric, $BW_{(u,v)}$ is the initial bandwidth capacity, $RBW_{(u,v)}$ is residual bandwidth capacity, $I_{(u,v)}$ is number of flow carried on the link (u, v). $U_{(u,v)}$ is equal to $1 - RBW_{(u,v)}/BW_{(u,v)}$. m is the number of the links and n is the number of nodes in the topology. We choose $\alpha = 0.5$ and $\beta = 0.3$ in our tests because they are shown to give the best results [8], [33]

TABLE I: Classification of existing routing algorithms.

cost such that the underlying network resources can be effectively utilized to increase the throughput while avoiding congestion. To achieve these goals, the routing algorithms have to compute paths for the given demands such that the rejection of the future demands will be minimized.

Since future demands are not known, it is impossible to achieve these goals when the flows arrive one at a time (on-line setting). Moreover, even if they are known (off-line setting), the problem turns out to be *NP-Complete* [16], [10], [14], [6], [5]. Therefore, heuristic algorithms have been proposed. In this section, we present the basic form of these heuristic algorithms as *RA-DLC* and review their advanced form as *RA-DLCMI* in the next section.

In essence, *RA-DLC* simply use the above mentioned *SP* and *WSP* by taking into account the *residual* bandwidth of each link rather than the static link bandwidth.

Accordingly, we call these algorithms as **Dynamic Shortest Path *(DSP)*** and **Dynamic Widest-Shortest Path *(DWSP)*** [12], [34].

The main advantages of RA-DLC are that they provide alternative paths for the flows between the same source and destination nodes. Accordingly, this allows the operators to effectively utilize all their resources and maximize their revenues. However, these algorithms require significant protocol overhead to be able to accurately obtain the dynamic link-state information. Moreover, the link cost needs to be re-computed based on new link-state information, involving an extra pre-computation cost before the actual path computation starts.

C. *Routing Algorithms with* Dynamic *Link Cost and Minimum Interference* (RA-DLCMI)

Using just the dynamic link metrics (e.g., residual bandwidth) as in *RA-DLC* may not be enough to maximize the chance of accepting more future demands while also providing better performance (e.g., less delay and loss). To accomplish these goals, *RA-DLCMI* try to evenly distribute the traffic load throughout the network and minimize the *interferences* among the flows. Accordingly, when determining the cost of each link, *RA-DLCMI* use the combination of other metrics (e.g., critical links between source-destination pairs and/or *number of flows* carried on the links) and the dynamic metrics (e.g., residual bandwidth).

In the literature, researchers have proposed several heuristic algorithms that mainly differ in how they compute the dynamic cost of each link using different combination of various metrics. We will first discuss the details of the key algorithms that we implemented. We then give a brief description of the other heuristic algorithms proposed in the literature.

Minimum Interference Routing Algorithm *(MIRA)* assumes that all the flow requests come from a predefined set of (ingress-egress) pairs. So when computing the shortest path for a given pair, MIRA tries to avoid the links that are more crucial to the other pairs [22]. To do that, it gives higher costs to such crucial/critical links. More specifically, before computing the shortest path for a given pair, it initially sets the cost metric of each link to 0. Using the residual bandwidth information in *max-flow min-cut* algorithm, MIRA computes the min-cut set between each pair except for the pair for which it tries to find a path. It then increases the cost of each critical link by 1 for each appearance in the min-cut sets. After this pre-processing step of computing link costs, MIRA runs Dijkstra's algorithm. MIRA is able to maximize the minimum available capacity between all other (ingress-egress) pairs. However, due to the excessive computational complexity of the pre-processing step for each flow, MIRA would have a limited use in practice, particularly when the network is large.

Least Interference Optimization Algorithm *(LIOA)* calculates the link cost metric using the inversely proportional ratio of the number of flows on each link to the residual capacity on that link [8]. Specifically, *(LIOA)* computes the link cost using $C_{(u,v)} = \frac{I^{\alpha}_{(u,v)}}{RBW^{\alpha}_{(u,v)}}$, and then calls Dijkstra's algorithm to determine the shortest path. Due the nature of the link cost calculations, *(LIOA)* favors the links with less number of flows and high residual bandwidth. Accordingly, it reduces the interference between source-destination (ingress-egress) pairs by balancing the number and quantity of flows in the network.

Improved Least Interference Optimization Algorithm *(ILIOA)* is an extended version of *LIOA* and considers more metrics in determining the link cost [33]. *(ILIOA)* takes into account the number of flows on the links, initial bandwidth capacity, residual bandwidth capacity, and utilization of the links. Specifically, *(ILIOA)* computes $C_{(u,v)} = (1 - U_{(u,v)}) * \frac{I^{\beta}}{BW^{\beta}_{(u,v)}} + U_{(u,v)} * \frac{I^{\alpha}}{RBW^{\alpha}_{(u,v)}}$. If link utilization is low, $(1-U_{(u,v)}) * \frac{I^{\beta}}{BW^{\beta}_{(u,v)}}$ dominates the cost metric to choose the links with large capacities. Otherwise, $U_{(u,v)} * \frac{I^{\alpha}}{RBW^{\alpha}_{(u,v)}}$ aims to choose the links with large remaining bandwidth.

In addition to the above algorithms, there are several other *RA-DLC* and *RA-DLCMI*. For example, Dynamic Link Weight (*DLW*) tries to choose lower loaded paths in order to avoid congested links [35]. As in *MIRA*, Wisitak's Routing Algorithm (*WSS*) calculates a cost parameter to find possible critical links [39]. However, it does not take into account residual bandwidth or other dynamic link costs. Fabio's Weight Function (*WF*) uses dynamic link costs; but, it does not take into account pair interferences [37]. BU-MIRA computes a cost metric for each link by using the available bandwidth of links and the number of flows carried on it [44]. The algorithm by Wang-Su-Chen (*WSC*) is the diversified version of *MIRA* and determines the cost metric with different method that takes into account the other *dynamic* and *static* link costs such as hop count and residual bandwidth along with critical links [41]. Bandwidth Constrained Routing Algorithm (*BCRA*) uses residual capacity, initial bandwidth of the links and the number of the flows carried on the links to find interference, as in *LIOA* [24]. Maximize Residual Bandwidth and Link Capacity - Minimize Total Flows Routing Algorithm (*MaxRC-MinF*) computes a cost metric as in *LIOA* and *BCRA* [25]. These algorithms have been intensively evaluated in the literature and

shown to under-perform when compared to the three algorithms (MIRA, LIOA, ILIOA) that we implement. Therefore, we will not include them in our evaluations. In addition, our evaluations do not include Light Minimum Interference Routing (*LMIR*) algorithm [14] because we determined some ambiguities in some steps that do not lead to finding the lowest capacity paths.

IV. Obtaining Network State Information

All the algorithms we presented assume that the accurate network state information (NSI) is available at the controller before computing a path. Unfortunately, the accurate NSI cannot be obtained in practice unless the controller can continuously query the distributed switches. Even if it can, this will impose significant protocol overhead. Moreover, the NSI will still be inaccurate due to delays and measurement errors. To balance the trade-offs between the accuracy and protocol overhead, the controller usually implements a periodic monitoring mechanism where it obtains the link-state information from each switch at a predetermined rate (e.g., every $\mathscr{T}$ seconds) [38]. As the value of $\mathscr{T}$ is reduced, the controller is expected to get much more accurate NSI at the cost of significant protocol overhead. Therefore, the value of $\mathscr{T}$ should be selected carefully.

In the rest of this section, we will first show how we can mimic the availability of accurate NSI at the controller. We then discuss the key steps in periodically collecting NSI. In the next section, to better compare the existing algorithms under the ideal and practical settings, we will use both accurate NSI and periodically collected NSI, respectively.

1. **Accurate Network State Information (*NSI*):** To mimic the availability of accurate NSI, we make the controller keep the shadow of the topology and the residual link capacities. Upon computing a path for a given flow demand, the controller subtracts the requested demand from the residual capacity of the links on the computed path. As a result, the controller can always have the up to date residual graph before computing a new path. Clearly, this idealistic approach cannot be implemented unless there is only one physical controller and there is no background traffic. Nevertheless, we can use this approach in our simulations to better judge the performance of various routing algorithms under the original assumption of having accurate NSI.
2. **Periodically collecting *NSI*:**
   In practice, the controller monitors NSI periodically by sending requests in every $\mathscr{T}$ seconds. Specifically, the controller sends the port statistics requests to the all *OpenFlow* switches. As soon as the switches receive the message, they reply with total traffic on the ports. Let $tx^c_{(u,v)}$ be the currently recorded total amount of traffic that is sent until the current time t_c. Let $tx^p_{(u,v)}$ be the previously recorded total amount of traffic that was sent until the previous query time t_p. Using these quantities, the controller can update the residual bandwidth of link $(u.v)$ as follows:

$$\mathcal{D}_{(u,v)} = \frac{tx^c_{(u,v)} - tx^p_{(u,v)}}{t_c - t_p} \tag{1}$$

$$RBW_{(u,v)} = BW_{(u,v)} - \mathcal{D}_{(u,v)} \tag{2}$$

   Using the equation (1), the controller calculates the average current traffic rate denoted by $\mathcal{D}_{(u,v)}$. Using the equation (2), the controller subtracts $\mathcal{D}_{(u,v)}$ from the $BW_{(u,v)}$, which is the initial capacity of the link, and determines the residual bandwidth denoted by $RBW_{(u,v)}$. Finally, the controller uses the following equations to save the current quantities as the previous quantities for the calculations in the next period.

$$t_p = t_c \tag{3}$$

$$tx^p_{(u,v)} = tx^c_{(u,v)} \tag{4}$$

V. Performance Evaluation

Our goal is to compare the performance of several existing routing algorithms in the context of SDN-based networks. Accordingly, we have implemented some of the promising routing algorithms within *RYU SDN* Controller [38], which provides application program components/interfaces (*APIs*) to create new control applications. We then used our controller on *Mininet* emulator, which creates a network of virtual switches, hosts, controllers, and links [2], [27]. The controller used OpenFlow protocol (version 1.3) to interact with the underlying switches on *Mininet*. The traffic generation and performance measurements were performed by the hosts on *Mininet* using *Linux* software and *iperf* [1].

Since the routing algorithms we compare take into account the given requested demand, we generated *UDP* flows with randomly selected demands rather than using *TCP* flows which try to use all of the available bandwidth to maximize throughput. Upon receiving a randomly generated flow request $(s, d, r_{(s,d)})$, the controller computes a path from s to d based on the network state information at the controller and checks if that path has enough available capacity. If so, it installs the necessary

forwarding rules along the path. Otherwise, it rejects the flow. Each established flow continues to load the network at the given requested demand until the end of the simulation.

As the performance measures, we have considered the followings:

Number of Accepted Flows
: We generate the same set of flow requests under each algorithm and then count the number of flows for which that algorithm finds a path.

Total Transfered Bandwidth (throughput)
: The total amount of transfered data rate of the accepted flows. In the best case, this will the sum of the demands of the accepted flows.

Packet Loss
: The percentage of the packets got lost with respect to packets sent by the accepted flows.

Normalized Path Computation Time
: The computation time of each algorithm over the computation of the algorithm which has the minimum computation time.

In the rest of this section, we first describe the network topologies that we used. Regarding network state information (NSI), we used both methods that we discussed in Section IV. Accordingly, we then present performance evaluation results under *Accurate NSI*, where we keep the shadow of the topology and update it at the controller. Finally, we present performance evaluation results while *Periodically Collecting NSI* with the intervals of $\mathscr{T} = 3, 5, 10$ seconds.

A. Network Topologies

We compared the routing algorithms under two different topologies, namely *MIRANET* [22] and enhanced *ANSNET* [11]. We chose *MIRANET* to fairly compare all the algorithms under the same settings used by the algorithms in [12], [33], [8], [22]. As seen in Figure 3, *MIRANET* topology has 15 nodes and 56 bidirectional links. The thinner links have 5 Mbps and thicker links have 25 Mbps. We kept the same four (s, d) pairs (namely, $(S1, S13), (S5, S9), (S4, S2)$ and $(S5, S15)$) that were originally used in [12], [33], [8], [22]. We randomly generated 100 flows (25 flows between each pair), 200 flows (50 flows between each pair), and 300 flows (75 flows between each pair) with requested demand of $uniform(50, 200)$ kbps.

To see how the algorithms perform on a larger topology, we used the realistic network topology in Figure 4, which is modified from *ANSNET* in [11]. It has 32 nodes and 108 bi-directional links. We randomly assigned bandwidth capacity of the links $uniform(2, 10)$ Mbps.

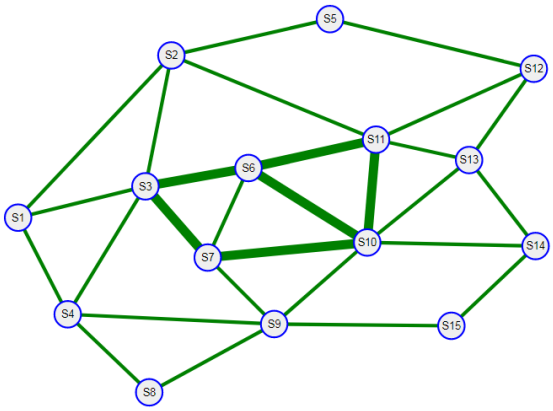


Fig. 3: MIRANET Topology.

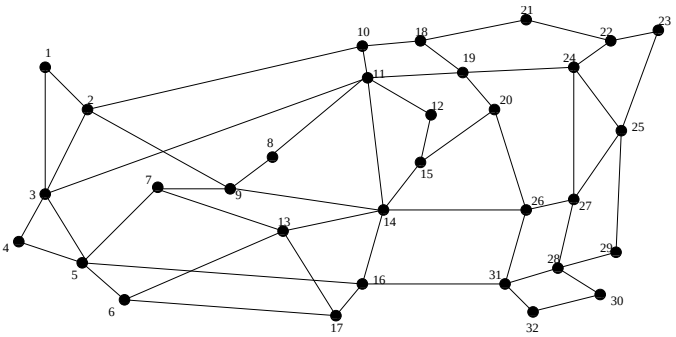


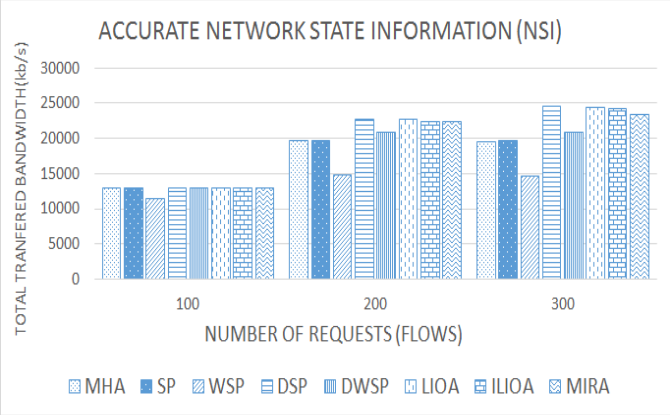
Fig. 4: Modified ANSNET Topology.

We randomly generated $50, 75, 100$ flows with requested demand of $uniform(200, 500)$ kbps while selecting the source from the nodes on the left side $(1, 2, 3, 4, 5)$ and the destination from the nodes on the right side $(23, 25, 29, 30, 31)$. This time pairs could be any match between the chosen source and destination, resulting in 25 different pairs. Under this setting, we did not evaluate *MIRA* algorithm because it gets computationally very expensive as it needs to run *max-flow* algorithm for all pairs except the pair for which it computes a path.

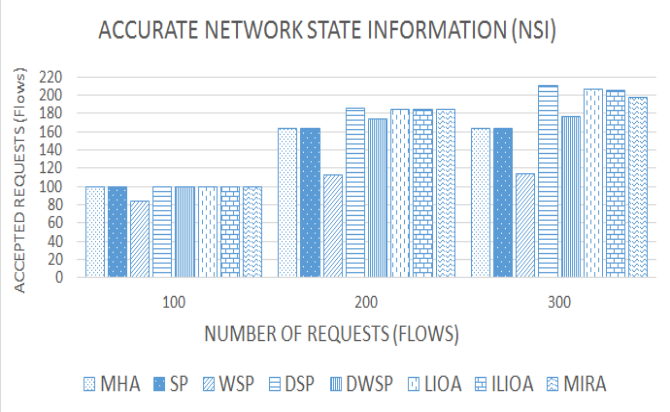
B. Performance Results under Accurate NSI

In general, since *RA-DLC* and *RA-DLCMI* use *dynamic* link costs, they are expected to find alternative paths for the flow requests and thus outperform the *RA-SLC* (e.g., MHA, SP, WSP) in terms of the number of accepted requests and total transfered bandwidth. We clearly see this general trend in Figure 5.

We are actually more interested in how the *RA-DLC* and *RA-DLCMI* perform. As seen in Figure 5, *DSP*, *LIOA* and *ILIOA* give similar results while *DWSP* gives less than these three. It seems that this low performance of *DWSP* can be attributed to the MIRANET topology because *DWSP* gives the similar performance under the



(a) Throughput under MIRANET with Accurate NSI



(b) Accepted Flows under MIRANET with Accurate NSI

Fig. 5: Throughput and Number of Accepted Flows under *MIRANET* Topology with Accurate NSI

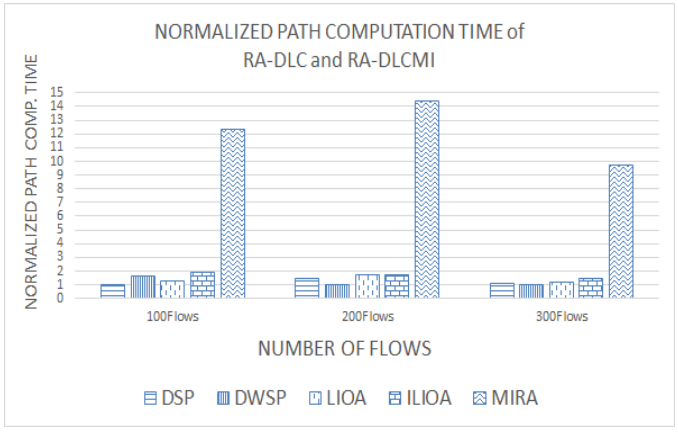
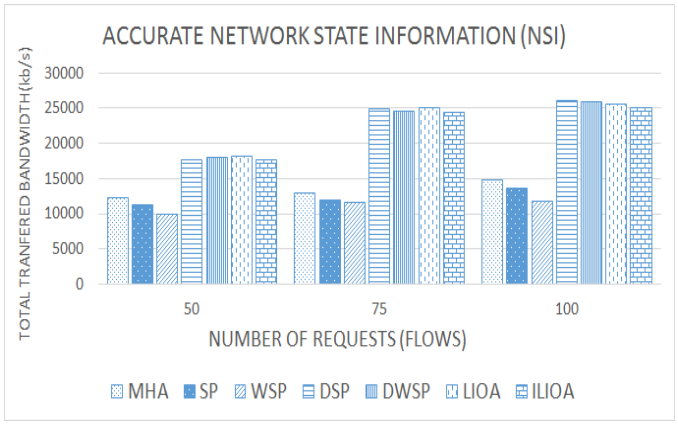
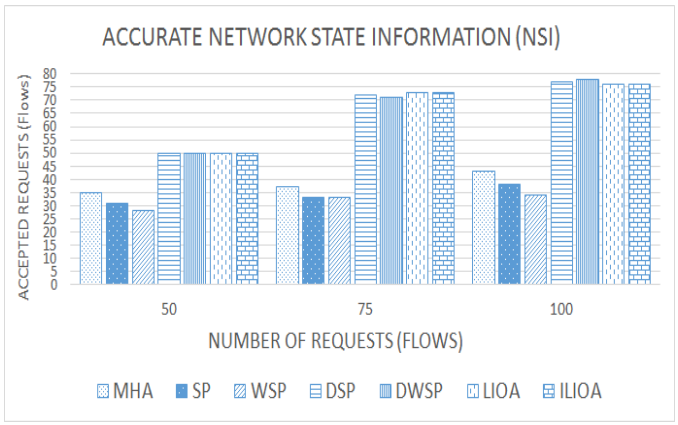


Fig. 6: Normalized Average Computation Time under *MIRANET* Topology with Accurate NSI.



(a) Throughput under ANSNET with Accurate NSI



(b) Accepted Flows under ANSNET with Accurate NSI

Fig. 7: Throughput and the Number of Accepted Flows Results under *ANSNET* Topology with Accurate NSI

larger ANSNET topology, as presented later. We believe that since *DWSP* occupies larger capacity links first and there is not much alternatives in MIRANET topology, *DWSP* cannot later find paths for the flow requests having larger demands.

Among the three best algorithms, *MIRA* is the most computationally expensive solution because it needs to run *max-flow min-cut* algorithm for all pairs. We have measured the actual computation times and normalized them by the fastest algorithm. As seen in Figure 6, the computation time of *MIRA* is $9-15$ times worse than the other algorithms. We did not show the computation time of the *static* algorithms because they are similar to the fastest *dynamic* algorithm as they all just use Dijkstra's algorithm.

Due to the high computation time of *MIRA*, we exclude it when comparing the other algorithms under our larger ANSNET topology with 25 source-destination pairs. As seen in Figure 7, we observe the same trend where *RA-DLC* and *RA-DLCMI* outperform the *RA-SLC* in terms of the number of accepted flows and total transfered bandwidth. We again observe that all the dynamic algorithms including *DWSP* gave similar performance results. We believe that this happens because all the algorithms are able to explore the alternatives that exist in larger topologies like ANSNET.

In theory, there should be no packet loss when we use the accurate NSI because the algorithms do not accept any flow beyond the available link capacities. However, during our experiments, we observed $1-2\%$ packet loss under *MIRANET* and $3-7\%$ packet loss under *ANSNET*. Some of these packet losses are due to the delay in computing paths and installing them on the switches. Moreover, the shadow topology and NSI that we maintain at the controller do not take in to account the protocol overheads imposed by OpenFlow Discovery Protocol (*OFDP*). To discover the topology, the controller sends encapsulated LLDP (Link Layer Discovery Protocol) packets to each switch. The switch sends the packet to one of its adjacent switch via a port. This switch encapsulates received LLDP packet and sends it back to the controller. After this process, the controller learns that there is a unidirectional link between the two switches. The same process is performed for all links in the network that may result in increasing of the controller's usage and load on the links. Thus, if the network size increases, the cost of network discovery increases linearly [7]. Since we could not incorporate this process when keeping the shadow graph for mimicking accurate NSI, we observe the above-mentioned packet losses.

Under periodically collected NSI, We do not have this type of loss because it takes into account total traffic on the ports including these protocol packets. However, periodically collected NSI causes other inaccuracies that also cause packed losses because the algorithms accept more flows than that the network can carry. We explain the reasons of packet loss due to periodically collected NSI in the next subsection.

C. Performance Results under Periodically Collecting NSI

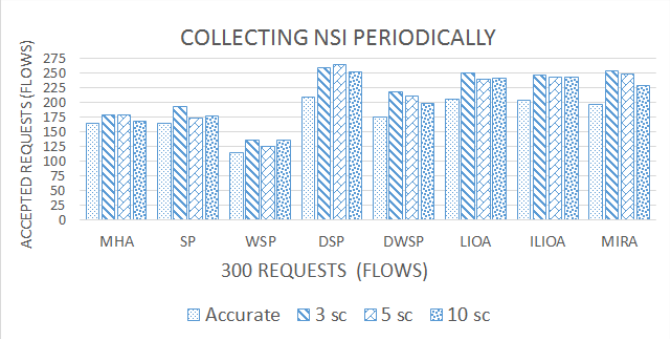
In this section, we conducted more simulation experiments to evaluate the routing algorithms under the same two topologies while *periodically collecting NSI*. Before presenting simulation results, we would like to make some general observations. First, due to the delays in periodically collecting NSI, the controller cannot have accurate information about the links. Accordingly, the controller may wrongly assume that there are enough capacities on the links and accepts more requests than the available bandwidth capacities of the links. This results in losing more packets for all flows.[2] Second, it is also possible that there is enough bandwidth capacity on the link because some flows stop sending traffic. However, until the next query time, the controller will not know that and assume that the link is still overloaded. In this case, the controller does not accept flows, resulting in underutilization of the network resources. In our simulations, we observed the effects of both cases, as explained below.

Simulations in this section use the same two topologies while *periodically collecting NSI* in every $\mathscr{T} = 3, 5, 10$ seconds. In addition, we generated the same set of flows as in Section V-B. Since we observed the similar trends with different number of flows, we will just present the results with 300 flows under *MIRANET* topology and 100 flows under *ANSNET* topology.

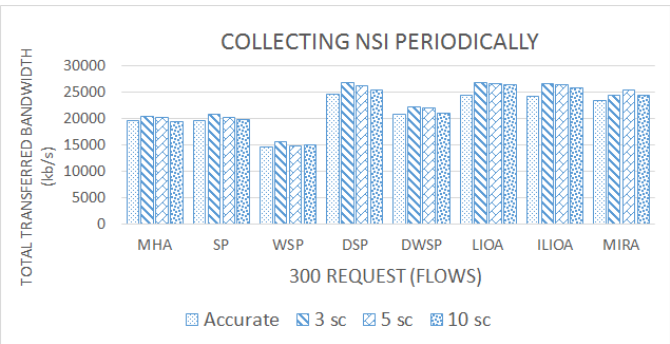
We first report the simulation results under *MIRANET* topology. As seen in Figure 8(a), all the algorithms accept more flow requests under periodically collected NSI than that under accurate NSI. As mentioned above, this often happens because the controller uses the NSI obtained in previous period and thus wrongly accepts more flows that overload some links. Since more flows are accepted, the total throughput will be higher as seen in Figure 8(b). However, this increase in throughput is not desirable because the algorithms wrongly accept more requests than the capacity of the network, causing every flow to lose significant number of packets, as seen in Figure 8(c). It is important for the routing algorithms to maintain the Quality of Service (*QoS*) by rejecting the flows that cannot be supported for the sake of providing *QoS* for the existing flows.

We would like to also note that intuitively the accuracy should be increasing as the collection interval decreases. However, we observed that all the algorithms accept more flows and lose more packets when we periodically collect NSI in every 3 seconds than 5 or 10 seconds. We believe that this happens because of two reasons: (a) collecting NSI with shorter periods significantly increase the load in the network and decrease the responsiveness of the controller, (b) SDN switches do not collect enough statistical information to make accurate estimation. Un-

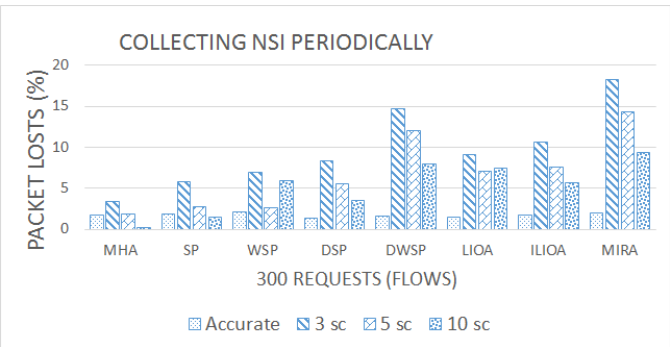
[2]To illustrate this, consider the following example. Assume there is a link with $6Mbps$ capacity and we generated two requests with demands of $5Mbps$. Under accurate NSI, the algorithms accept the first flow and update the available bandwidth capacity of that link as $1Mbps$. So, they do not accept second request which has higher demand ($5Mbps$) than the remaining capacity ($1Mbps$) of the link. But when periodically collecting NSI, the controller can accept both requests since the controller did not get the updated information yet. Clearly, this causes overload on the link and makes both flows suffer from packet loss while total throughput of both flows is higher or similar to the throughput of a single flow under accurate NSI.



(a) Accepted flows under MIRANET.

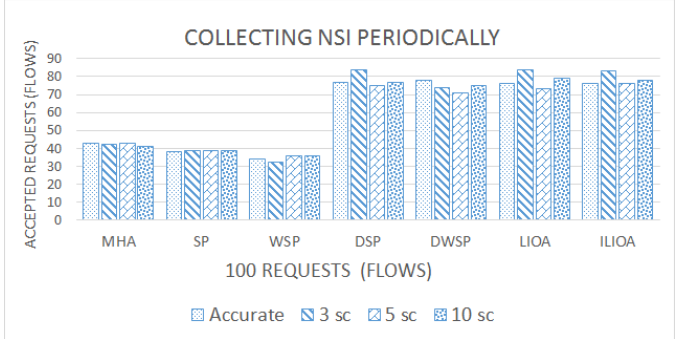


(b) Throughput under MIRANET.

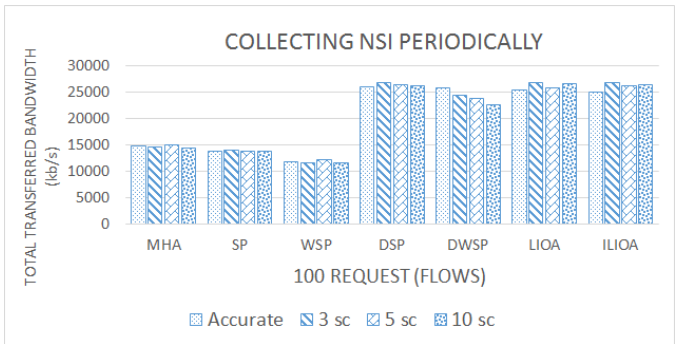


(c) Packet Lost (%) under MIRANET.

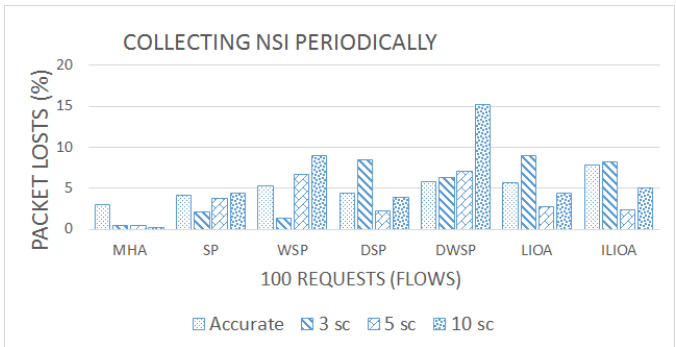
Fig. 8: Accepted Flows, Throughput, and Packet Lost (%) under *MIRANET* topology with Accurate and Periodically Collected NSI in every $3, 5$ and 10 seconds.



(a) Accepted Flows under ANSNET.



(b) Throughput under ANSNET.



(c) Packet Lost (%) under ANSNET.

Fig. 9: Accepted Flows, Throughput, and Packet Lost (%) under *ANSNET* topology with Accurate and Periodically Collected NSI in every $3, 5$ and 10 seconds.

der 5 and 10 seconds collection, the algorithms give more accurate results but still packet lost is high.

We now report the simulation results under *ANSNET* topology. As seen in Figure 9(a), again all the algorithms accept more flow requests when NSI is periodically collected in every 3 seconds, resulting in higher total throughput (as seen in Figure 9(b)) at the cost of more packet loss (as seen in Figure 9(c)). When the periodic collection is in every 5 seconds, all the algorithms accept less requests, resulting in underutilization of the network resources. When we collect NSI in every 10 seconds, the number of accepted flows is not increasing as much as packet loss. We believe that this happens because the controller computes the same paths for different requests due to the inaccurate state information which is stem from longer period of collection.

VI. Conclusion and Future Works

In this paper, we classified existing algorithms into three groups: (i) Routing Algorithms with Static Link Cost (*RA-SLC*) that use static link costs such as hop count, distance and/or link capacity [15], [34], [22]; (ii) Routing Algorithms with Dynamic Link Cost (*RA-DLC*) that use dynamic link cost such as available bandwidth capacity of the links; (iii) Routing Algorithms with

Dynamic Link Cost and Minimum Interference (*RA-DLCMI*) that use other dynamic link costs such as link utilization and number of flows carried on the links along with available bandwidth capacity and static link costs (e.g. hop count, distance, link capacity) [12], [22], [8], [33]. We then compared existing routing algorithms in the context of SDN-based networks. Accordingly, we have implemented many routing algorithms using RYU SDN-controller and tested them on Mininet emulator. As the network topology, we first considered the same *MIRANET* used in the literature. We then considered a larger topology by enhancing *ANSNET*.

One of the key issues was how to obtain network state information (NSI). All routing algorithms have been proposed under the assumption that accurate NSI is available at the controller that makes routing decisions. However, in practice, since the controller periodically collects NSI, it is hard to achieve high accuracy. In our evaluations, we try to compare the existing algorithms under both the idealistic case assuming the availability of accurate NSI and the practical case where the NSI is collected periodically and has some inaccuracy.

In order to mimic accurate *NSI*, we kept the shadow of the network in the controller and we updated links after each path computation to have residual graph. As expected, the results show that *RA-DLC* and *RA-DLCMI* outperform the *RA-SLC* in terms of the throughput and accepted requests under both topologies. We did not see significant difference among *RA-DLC* and *RA-DLCMI* algorithms in therms of throughput and accepted number of flows. However, MIRA was very expensive in terms of computation time. Under accurate *NSI*, we were not expecting any packet loss. However, we have observed $\%1 - 2$ packet loss in *MIRANET* topology and $\%3 - 7$ packet loss in *ANSNET* topology. We attribute these losses to the background traffic generated by OFDP (OpenFlow Discovery Protocol) and the delays until the paths are installed.

Under periodically collected *NSI* in every $3, 5$ and 10 seconds, we observed that the controller does not get consistence results due to inaccuracies in *NSI*. For example, when the algorithms use the available capacity information from previous period, they wrongly accept more flows than the network can carry. Accordingly, even though more flows are accepted and total throughput is higher, this causes significant packet losses and negatively affects the QoS of all flows. It is also possible that the actual available capacity is higher that the controller knows. In that case the algorithms unnecessarily rejects flow request and cause under utilization.

As the future work, we plan to specifically focus on how to deal with the inconsistencies caused by periodically collected *NSI*. First, we plan to develop algorithms that can take into account inaccurate state information using some probabilistic link metrics. Second, we plan to investigate how to effectively collect the NSI while minimizing the load on the controller and the message overhead throughout the network.

REFERENCES

[1] iPerf - the TCP, UDP and SCTP network bandwidth measurement tool. https://iperf.fr/. (Accessed on 01/25/2018).

[2] Mininet. http://mininet.org. Accessed: 2018-02-18.

[3] OpenFlow: Proactive vs reactive. http://networkstatic.net/openflow-proactive-vs-reactive-flows/. (Accessed on 01-23-2018).

[4] Reactive, proactive, predictive: SDN models. https://devcentral.f5.com/articles/reactive-proactive-predictive-sdn-models. (Accessed on 01-23-2018).

[5] Erdal Akin and Turgay Korkmaz. An efficient binary-search based heuristic for extended unsplittable flow problem. In *Computing, Networking and Communications (ICNC), 2017 International Conference on*, pages 831–836. IEEE, 2017.

[6] Erdal Akin and Turgay Korkmaz. Routing algorithm for multiple unsplittable flows between two cloud sites with QoS guarantees. In *Computing, Networking and Communications (ICNC), 2017 International Conference on*, pages 917–923. IEEE, 2017.

[7] Abdelhadi Azzouni, Nguyen Thi Mai Trang, Raouf Boutaba, and Guy Pujolle. Limitations of OpenFlow topology discovery protocol. In *Ad Hoc Networking Workshop (Med-Hoc-Net), 2017 16th Annual Mediterranean*, pages 1–3. IEEE, 2017.

[8] Antoine B Bagula, Marlene Botha, and Anthony E Krzesinski. Online traffic engineering: the least interference optimization algorithm. In *Communications, 2004 IEEE International Conference on*, volume 2, pages 1232–1236. IEEE, 2004.

[9] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 1–12. ACM, 2007.

[10] Shigang Chen and Klara Nahrstedt. An overview of quality of service routing for next-generation high-speed networks: problems and solutions. *IEEE network*, 12(6):64–79, 1998.

[11] Douglas E Comer. *Internetworking con TCP/IP*, volume 1. Pearson Italia Spa, 2006.

[12] Eric Crawley, H Sandick, R Nair, and B Rajagopalan. A framework for QoS-based routing in the internet. 1998.

[13] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to SDN: an intellectual history of programmable networks. *ACM SIGCOMM Computer Communication Review*, 44(2):87–98, 2014.

[14] Gustavo B Figueiredo, Nelson LS Da Fonseca, and José AS Monteiro. A minimum interference routing algorithm with reduced computational complexity. *Computer Networks*, 50(11):1710–1732, 2006.

[15] Robert W Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[16] Michael R Garey and David S Johnson. A guide to the theory of NP-Completeness. *WH Freemann, New York*, 70, 1979.

[17] Ali Ghodsi, Scott Shenker, Teemu Koponen, Ankit Singla, Barath Raghavan, and James Wilcox. Intelligent design enables architectural evolution. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, page 3. ACM, 2011.

[18] Andrew Goldberg and Robert Tarjan. Solving minimum-cost flow problems by successive approximation. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 7–18. ACM, 1987.

[19] P Goransson and C. Black. *Software Defined Networks: A Comprehensive Approach*. Morgan Kaufman, 2014.

[20] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, 2008.

[21] Hyojoon Kim and Nick Feamster. Improving network management with software defined networking. *IEEE Communications Magazine*, 51(2):114–119, 2013.

[22] Murali Kodialam and TV Lakshman. Minimum interference routing with applications to MPLS traffic engineering. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 2, pages 884–893. IEEE, 2000.

[23] Turgay Korkmaz and Marwan Krunz. Bandwidth-delay constrained path selection under inaccurate state information. *IEEE/ACM Transactions on Networking (ToN)*, 11(3):384–398, 2003.

[24] Afef Kotti, Rached Hamza, and Kamel Bouleimen. Bandwidth constrained routing algorithm for MPLS traffic engineering. In *Networking and Services, 2007. ICNS. Third International Conference on*, pages 20–20. IEEE, 2007.

[25] Piyaporn Krachodnok. Constraint-based routing with maximize residual bandwidth and link capacity-minimize total flows routing algorithm for MPLS networks. In *Information, Communications and Signal Processing, 2005 Fifth International Conference on*, pages 1509–1514. IEEE, 2005.

[26] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.

[27] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[28] Dan Levin, Andreas Wundsam, Brandon Heller, Nikhil Handigol, and Anja Feldmann. Logically centralized?: State distribution trade-offs in software defined networks. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 1–6, New York, NY, USA, 2012. ACM.

[29] Nick McKeown. How SDN will shape networking. *Open Networking Summit*, 2011.

[30] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[31] John Moy. OSPF version 2. 1997.

[32] Bruno Astuto A Nunes, Marc Mendonca, Xuan-Nam Nguyen, Katia Obraczka, and Thierry Turletti. A survey of software-defined networking: Past, present, and future of programmable networks. *IEEE Communications Surveys & Tutorials*, 16(3):1617–1634, 2014.

[33] Ireneusz Olszewski. The improved least interference routing algorithm. In *Image Processing and Communications Challenges 2*, pages 425–433. Springer, 2010.

[34] Ariel Orda and George Apostolopoulos. QoS routing mechanisms and OSPF extensions. 1999.

[35] Huan Pham and Bill Lavery. New dynamic link weight algorithm for on-line calculation of LSPS in MPLS networks. In *Communication Systems, 2002. ICCS 2002. The 8th International Conference on*, volume 1, pages 117–121. IEEE, 2002.

[36] Barath Raghavan, Martín Casado, Teemu Koponen, Sylvia Ratnasamy, Ali Ghodsi, and Scott Shenker. Software-defined internet architecture: decoupling architecture from infrastructure. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 43–48. ACM, 2012.

[37] Fabio Ricciato and Ugo Monaco. On-line routing of MPLS tunnels with time-varying bandwidth profiles. In *High Performance Switching and Routing, 2004. HPSR. 2004 Workshop on*, pages 321–325. IEEE, 2004.

[38] RYU. Component-based software defined networking framework, 2017. [Online; accessed 6-December-2017].

[39] W Sa-Ngiamsk, S Thipchaksurat, and R Varakutsiripunth. A bandwidth-based constraint routing algorithm for multi-protocol label switching networks. In *Advanced Communication Technology, 2004. The 6th International Conference on*, volume 2, pages 933–937. IEEE, 2004.

[40] Scott Shenker, Martin Casado, Teemu Koponen, and Nick McKeown. The future of networking and the past of protocols, oct. 2011. *Invited talk at Open Networking Summit*.

[41] Bin Wang, Xu Su, and CL Philip Chen. A new bandwidth guaranteed routing algorithm for MPLS traffic engineering. In *Communications, 2002. ICC 2002. IEEE International Conference on*, volume 2, pages 1001–1005. IEEE, 2002.

[42] Minlan Yu, Jennifer Rexford, Michael J Freedman, and Jia Wang. Scalable flow-based networking with DIFANE. *ACM SIGCOMM Computer Communication Review*, 40(4):351–362, 2010.

[43] Man-Ching Yuen, Weijia Jia, and Chi-Chung Cheung. Simple mathematical modeling of efficient path selection for QoS routing in load balancing. In *Multimedia and Expo, 2004. ICME'04. 2004 IEEE International Conference on*, volume 1, pages 217–220. IEEE, 2004.

[44] Mingying Zhu, Wu Ye, and Suili Feng. A new dynamic routing algorithm based on minimum interference in MPLS networks. In *Wireless Communications, Networking and Mobile Computing, 2008. WiCOM'08. 4th International Conference on*, pages 1–4. IEEE, 2008.