Achieving Efficient and Fast Update for Multiple Flows in Software-Defined Networks

Yujie Liu†, Yong Li†, Yue Wang†, Athanasios V. Vasilakos‡, Jian Yuan†
† Tsinghua National Laboratory for Information Science and Technology
Department of Electronic Engineering, Tsinghua University, Beijing 100084, China
‡ Department of Computer and Telecom. Engineering, University of Western Macedonia, Greece
liyong07@tsinghua.edu.cn

ABSTRACT

Aiming to adapt traffic dynamics, deal with network errors, perform planned maintenance, etc., flow update is carried out frequently in Software-Defined Networks (SDN) to change the data plane configuration, and how to update the flows efficiently and successfully is an important and challenging problem. In this work, we address the multi-flow update problem and present a polynomial-time heuristic algorithm, which aims at completing the update in the shortest time considering link bandwidth and flow table size constraints. By extensive simulations under real network settings, we demonstrate the effectiveness and efficiency of our algorithm, which has near-optimal performance and is hundreds of times faster than the optimal solution.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations—*Network management*

Keywords

Software-defined networks; flow update; heuristic algorithm

1. INTRODUCTION

With the increasing demand of simplifying network control, Software-Defined Networking (SDN) [1] is an emerging network architecture to provide efficient control and management of the network. Different from the Internet, SDN separates the data plane from the control plane, and the data plane forwards the packets using the forwarding table configured by the controller in the control plane through southbound interfaces, such as OpenFlow. However, the forwarding rules of flows need to change frequently in many scenarios to adapt to various traffic dynamics, deal with network errors, perform planned maintenance, etc. Therefore, the controller needs to reconfigure the forwarding plane. The flow update should be carried out successfully, since if the update fails, the network will not be configured properly to meet the new demand, which may cause network error and service degradation. Thus, we desire to finish the update process as fast as possible to adapt to the new circumstances rapidly.

Formally, flow update in SDN can be described as follows: multiple active flows are transmitted in the network, and the controller installs a set of new packet-forwarding rules, says B, into the switches to replace the initial rules A. Normally a two-phase method is used to update the flow table [4] [5], which first stamps every packet with a version number, *e.g.*, VLAN tag, then performs the update by the following three stages: (I) install B in the middle nodes of the network; (II) install B at the perimeter of the network; (III) when all packets processed by A have left the network, remove A in all nodes. Note that in stage (II) the switches contain both the initial and final forwarding rules, which requires double flow table space. Since flow table is a limited resource, there may not be enough space when multiple flows are updated at the same time. Therefore, it is a viable solution to split the update process into several steps [2, 3, 4].

In the multi-flow update process, there are many possible situations in the intermediate steps when a part of the flow paths have been updated. Even though the network is not congested under the initial and expected states, it may still suffer from congestions and packet losses during the update process. From the perspective of links, bandwidth is a major constraint factor, since the physical links shared by several flows may be very busy. To avoid congestions, we need to make sure the link utilization during update is not beyond the capacity limits. From the perspective of switches, flow table space is another important constraint factor. Note that in SDN flow table size is very limited, because the commonly used flow table, especially TCAM (Ternary Content Addressable Memory), is expensive and power hungry [3]. For example, the analysis in [3] demonstrates that in order to use 15-shortest path, up to 20K flow entries are needed, which is beyond the flow table size of even next-generation SDN switches. Once the flow table is fully occupied, the switch will refuse to install other flow entries, which causes network forwarding error. Moreover, flow table usage is closely correlated with bandwidth utilization, since forwarding the packets of a flow through the path requires using bandwidth of the links as well as installing flow entries into the switches. Thus, we need to jointly consider the above two key factors instead of dealing with them independently in the multi-flow update problem.

In this paper, we study the problem of multi-flow update by taking both link bandwidth and flow table size con-

DCC'14, August 18, 2014, Chicago, Illinois, USA.

http://dx.doi.org/10.1145/2627566.2627572.

straints into account. Our contributions can be summarized as below.

- We formulate the multi-flow update problem as an optimization problem that minimizes the number of steps to find the optimal flow update scheme, under both link bandwidth and flow table size constraints.
- We propose a heuristic algorithm to solve the multi-flow update problem, which fully utilizes the network resources and finds the solution in polynomial time.
- We demonstrate the effectiveness and robustness of our algorithm by extensive simulations under realistic settings. The results demonstrate our algorithm has near optimal performance and is hundreds of times faster than the optimal solution.

The rest of the paper is organized as follows. In Section 2, we describe and formulate the multi-flow update problem. We depict the optimal solution and our heuristic algorithm in Section 3. After presenting performance evaluation in Section 4, we conclude the paper in Section 5.

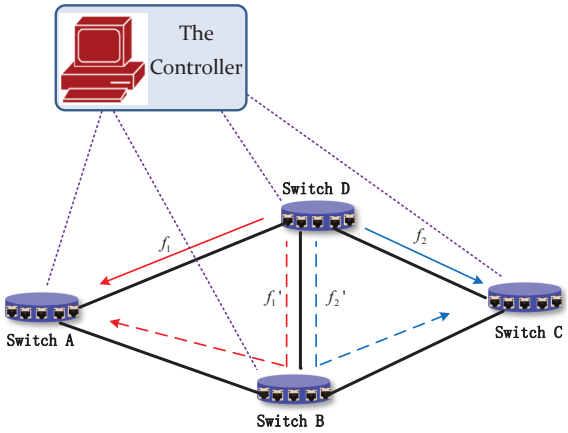


Figure 1: System Overview. For each flow the traffic size is 2 units and 4 rules are required. The link capacity is 10 units, and the flow table size of the switches is 15 rules. We cannot update f_1 and f_2 in one step, and a feasible solution is to update them one by one.

2. PROBLEM FORMULATION

2.1 Problem Statement

We now explain the details of the multi-flow update problem by an example shown in Fig. 1. In this scenario, there are several switches and a SDN controller. The controller has full knowledge of the network forwarding state and configures the flow table of all switches via OpenFlow protocol. There are multiple active flows in the network, and their forwarding paths are determined by relative flow entries in the switches. When changes occur in traffic demand or network topology, the controller detects the event and carries out flow update to refresh the forwarding rules.

During the update, transient congestion may happen which leads to packet losses and service degradation. Thus, it is complicated to carry out multi-flow update, and a feasible solution is to update the flows step-by-step. As shown in Fig. 1, two flows, f_1 and f_2, need to change their paths to f_1' and f_2'. The link capacity is 10 units, and the flow table size of a switch is 15 rules. The traffic size of each flow is 2 units and 4 rules are required in the switches on its path. To carry out the update, we should first add the new forwarding rules into relative switches. Therefore, if we update f_1 and f_2 in one step, Switch D will have to hold 16 rules, which is beyond its flow table size. However, if we separate the update into two steps, it is completed successfully. Thus we investigate the multi-flow update problem with the multi-step strategy.

2.2 System Model

We now formally describe the multi-step flow update problem. The network consists of D switches denoted by set U which are connected by L directed links denoted by set E. The traffic of K active flows need to be updated. We use b_j and $s_j, j \in \{1, \cdots, K\}$ to represent their traffic load and the number of flow entries that need to be installed to route flow j through the switches respectively. The old routing paths are denoted by $\{P_1, \cdots, P_i, \cdots, P_K\}$, where P_i is the path from the source node of flow i to its destination node, and the new paths are denoted by $\{P_1', \cdots, P_i', \cdots, P_K'\}$. A path from a source node u_s to a destination node u_d is defined as the list of nodes $\{u_s = u_0, u_1, u_2, \cdots, u_k = u_d\}$, and the links connecting them, where u_i is the next hop node of u_{i-1} and $(u_{j-1}, u_j) \in E$. In every path the next hop of a switch is only one, and the paths are loop-free. We use $c(e)$ to represent the capacity of link e and $q(u)$ to represent the flow table size of switch u.

In the multi-step flow update scheduling problem, considering the network resource constraints, we want to finish the update as quickly as possible. We use the binary variable $I_k, k \in \{1, \cdots, K\}$ to indicate whether the update is still in progress in step k: $I_k = 1$ if some flow is updated in step k, and $I_k = 0$ if the update has been completed before the kth step. To further describe how the update is carried out in detail, we use x_{ij} to denote the part of flow j that is updated from P_j to P_j' in step i. Let $f_i(e), i \in \{0, 1, \cdots, K\}$ represent the traffic load on link $e, e \in E$ when the ith update step is finished. Note that, $f_0(e)$ and $f_K(e)$ indicate the initial and final traffic distribution in the network respectively. We use $n_i(u), i \in \{0, 1, \cdots, K\}$ to denote the maximum number of flow entries on switch u during step i.

2.3 Formulation of the Flow Update Problem

Since we aim to complete the update using the least number of steps, the objective of the optimization problem can be formulated as follows,

$$\textbf{min} \quad \sum_{k=1}^{K} I_k. \tag{1}$$

From the system model, we have $\sum_{i=1}^{K} x_{ij} = 1, j \in \{1, \cdots, K\}$ which means the entire traffic of every flow has been routed to the new path, since every flow should have finished the update in K steps. The variable I_k is restricted by the linear combination of x_{ij} as $I_k \geq \sum_{j=1}^{K} \frac{1}{K} x_{kj}, k \in \{1, \cdots, K\}$ and $I_k \geq I_{k+1}$, $k \in \{1, \cdots, K-1\}$. Since I_k is a binary variable, we can observe that $I_k = 1$ if any value of $x_{kj}, k \in \{1, \cdots, K\}$ is above zero which means that some

flow is updated in step k, and $I_k = 0$ if the values of x_{kj} are all zeros meaning that the update has been completed before the kth step. Note that the constraint of $I_k \geq I_{k+1}$ ensures that as long as the update is finished in the kth step, there will be no more update operations in the subsequent steps.

Then we formalize the link and switch constraints of the problem. Concerning the link bandwidth constraint, $f_0(e)$ represents the initial traffic load on each link. It can be calculated by adding all the traffic load of the flows routed through link e as $f_0(e) = \sum_{j=1}^{K} In(e \in P_j)b_j$, where $In(\cdot)$ is a function indicating the value of the input logic expression. Then we assume in the ith step flow j moves x_{ij} part of its traffic to the new path. If link e belongs to the new path P'_j, the value of $f_i(e)$ should add $x_{ij}b_j$; if link e belongs to the old path P_j, the value of $f_i(e)$ should subtract $x_{ij}b_j$; if link e belongs to both of P'_j and P_j or neither of them, the value of $f_i(e)$ will not change. In this way, we can calculate $f_i(e)$ iteratively for every step,

$$\begin{aligned} f_i(e) =& f_{i-1}(e) + \sum_{j=1}^{K} In(e \in P'_j)b_j x_{ij} \\ & - \sum_{j=1}^{K} In(e \in P_j)b_j x_{ij}. \end{aligned} \tag{2}$$

Based on the iterative formula of $f_i(e)$, we have

$$\begin{aligned} f_i(e) =& \sum_{j=1}^{K} In(e \in P_j)b_j + \sum_{a=1}^{i}\sum_{j=1}^{K} In(e \in P'_j)b_j x_{aj} \\ & - \sum_{a=1}^{i}\sum_{j=1}^{K} In(e \in P_j)b_j x_{aj}. \end{aligned} \tag{3}$$

With regard to the constraint of flow table size, we focus on the maximum number of flow entries on the switches in each step. When traffic load is heavy, our solution can divide the update of a flow into several steps and moves a part of the flow to the new path in each step. To route a part of the flow through its new path, no matter how much traffic is migrated, the new flow entries should be added to the switches. However, only when the whole flow has been updated, the old flow entries can be removed. Thus we introduce two binary variables yo_{ij} and yn_{ij} to represent whether the old forwarding rules of flow j can be removed and whether the new forwarding rules should be added at stage II of step i, which can be described in detail as follows,

$$\begin{aligned} yo_{ij} &= \begin{cases} 0, & i = 1; \\ In(\sum_{a=1}^{i-1} x_{aj} = 1), & 2 \leq i \leq K; \end{cases} \\ yn_{ij} &= sign(\sum_{a=1}^{i} x_{aj}), 1 \leq i \leq K. \end{aligned} \tag{4}$$

Note that, at stage II of the first update step, no old rules will be deleted. Since $n_0(u)$ represents the original number of flow table entries configured in each switch, it can be calculated as $n_0(u) = \sum_{j=1}^{K} In(u \in P_j)s_j$.

Based on the above analysis, we can derive the expression of $n_i(u)$ as follows,

$$\begin{aligned} n_i(u) =& n_0(u) + \sum_{j=1}^{K} In(u \in P'_j)s_j yn_{ij} \\ & - \sum_{j=1}^{K} In(u \in P_j)s_j yo_{ij}. \end{aligned} \tag{5}$$

From the model description, we also have the constraints of the variable x_{ij} as $0 \leq x_{ij} \leq 1$.

As mentioned before, the SDN controller is supposed to complete the update process using the least number of steps during which the link and switch constraints are satisfied. Thus, combining the above objective and the constraints, we formulate the optimization problem as follows,

$$\textbf{min} \quad \sum_{k=1}^{K} I_k$$

$$s.t. \begin{cases} f_i(e) \leq \theta c(e), \forall i \in \{1, \cdots, K\}, \forall e \in E; & \text{(6a)} \\ n_i(u) \leq q(u), \forall i \in \{1, \cdots, K\}, \forall u \in U; & \text{(6b)} \\ 0 \leq x_{ij} \leq 1, \forall i, j \in \{1, \cdots, K\}; & \text{(6c)} \\ \sum_{i=1}^{K} x_{ij} = 1, \forall j \in \{1, \cdots, K\}; & \text{(6d)} \\ I_k \geq \sum_{j=1}^{K} \frac{1}{K} x_{kj}, \forall k \in \{1, \cdots, K\}; & \text{(6e)} \\ I_k \geq I_{k+1}, \forall k \in \{1, \cdots, K-1\}. & \text{(6f)} \end{cases}$$

In the above formulation, θ represents the maximum allowed link utilization which indicates the load of all links should remain below a certain level to guarantee network performance during the update. The decision variables are x_{ij} indicating the flow update solution, and the objective is to find the shortest update process without violation of resource constraints.

3. PROBLEM ANALYSIS AND SOLUTION

3.1 Heuristic Algorithm Design

We propose a heuristic solution for its simplicity and efficiency to solve the multi-step flow update problem. Recalling the constraints shown in (6a) and (6b), we find that flow table constraint is more difficult to meet than the link capacity constraint, since the max overhead of the update process comes from the stage II of each step when the new flow entries have been added to the switches. Thus, flow table space for both the new and the old forwarding rules are required at the same time, which may not be available for the switches with few resources left, such as some key switch belonging to multiple routing paths. This kind of switch is the bottleneck of the update scheduling problem, and a feasible solution is to migrate the old flows passing through the key switch first to make room for the new flows.

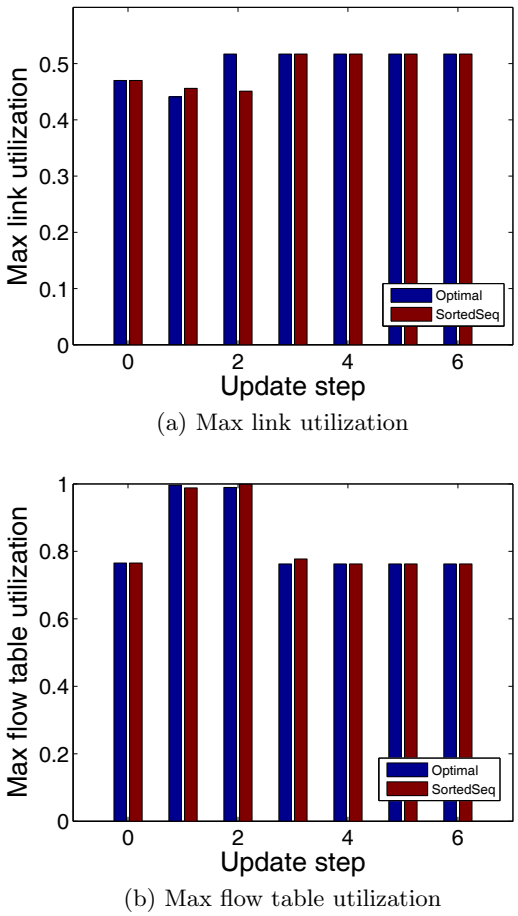
Based on the above analysis, we propose a heuristic algorithm named *SortedSeq* which gives flow table usage a priority consideration. For each flow j, we calculate the maximum percentage of s_j in the free flow table space of each switch on its new path, and denote the value as v_j. On one hand, the greater value of v_j means the update of

Algorithm 1 Heuristic algorithm for the multi-flow update problem.

1: Initialize: set $Up(j) = 0$
2: **if** the flows can be updated in one step **then**
3: $Up(j) = 1, \forall j \in [1, K]$
4: **return** x as the solution, $step = 1$
5: **else**
6: Find u which has the max value of $n_K(u)$, and update flow $j, \tilde{u} \in P_j$ if possible
7: Go over the other flows, and update them if possible
8: **end if**
9: **while** $\min Up(j) = 0$ **do**
10: Sort the flows in the descending order of $v_j = \max \frac{s_u}{q(u)-n_t(u)}$
11: Update the ones for which there is enough resource, refresh $Up(j)$
12: **end while**
13: **return** the solution

flow j is more restricted and should be accomplished earlier. On the other hand, since the initial and final network state is congestion-free, to minimize the amount of update steps, the remaining resources must be fully utilized to update as many flows as possible at each step.

According to the two basic principles, our proposed heuristic algorithm is shown in Algorithm 1. To explain it, let us review the sample scenario in Fig. 1. In step 2-4, *SortedSeq* first checks whether f_1 and f_2 can be updated together. The one-step attempt fails, then the algorithm picks out Switch D which has the most flow entries in the final state, and update f_1 to f_1' in step 5-8. Next in step 9-13, *SortedSeq* updates f_2 to f_2' and outputs the solution $f_1 \rightarrow f_2$. Note that, our algorithm cannot always get the optimal solution, but it has near-optimal performance and achieves a much faster running time as we will show in the next section.



(a) Max link utilization

(b) Max flow table utilization

Figure 2: Performance in each step during the update process: (a) max link utilization, and (b) max flow table utilization.

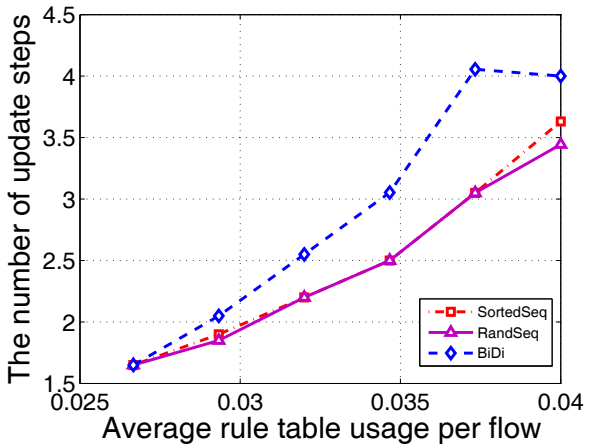
4. PERFORMANCE EVALUATION

We evaluate the performance of our algorithm using extensive simulations over real network settings, and compare it with three algorithms. The first one is the *Optimal* algorithm, which seeks the optimal solution of the flow update problem formulated in (6a) to (6f) using YALMIP toolbox [8]. The other two are *RandSeq* and *BiDi*, each of which changes one important feature of our algorithm respectively. *RandSeq* also iterates over the set of all the remaining flows at every step, but different from *SortedSeq*, it sorts the flows in a random way instead of taking into account the flow table utilization of each flow. In contrast, *BiDi* rearranges the flows by the descending order of v_j before each step, but it searches the flows in a bidirectional way. In each step, *BiDi* runs two searches over the remaining sorted flows: one forward from the flow with the highest value of v_j, and one backward from the flow with the lowest value of v_j. The update is completed when the two searches meet in the middle. It is easy to prove that *RandSeq* and *BiDi* are polynomial-time algorithms.

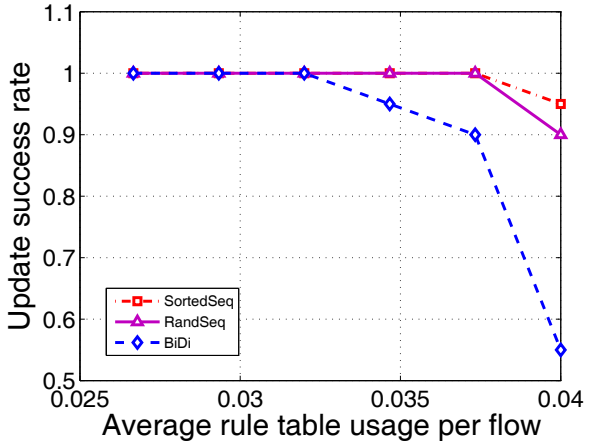
4.1 Experimental Settings

We evaluate our algorithm by Google's inter-datacenter WAN B4 [9], which has 12 nodes and 38 directional links. Since the flow update algorithms are all independent to the routing selection methods, we consider the scenario where the SDN controller uses the shortest path in OSPF to choose the forwarding paths for the flows, and select the source and the destination nodes of each flow in a random way. Based on the weights assigned to each link, the initial routing policy can be easily calculated. Then the weights are rearranged to simulate events causing flow update, such as broken links or dynamic traffic demands. Thus, the final routing policy can be obtained by performing the shortest path routing method again.

The network parameters are configured as follows. The link bandwidth is 1Gbps, and the flow table size of all switches are set to be 750 rules, which is the same as what the testbed switches support in the experiment of [3]. The traffic rate of each flow follows uniform distribution of $[b - \delta_b, b + \delta_b]$, where b is the average flow rate and δ_b is the maximum deviation between the link rate of every flow and the average value. We set b to be 30Mbps and δ_b to be 10Mbps by default. The flow considered in the update scenario actually refers to the aggregation of several real flows. For example, in backbone networks since there are multiple different paths between two switches, in order to



(a) The number of steps



(b) Update success rate

Figure 3: Performance during the update process as function of the flow table size: (a) the number of steps and (b) update success rate.

fully utilize the link bandwidth, several flow entries need to be installed on them. Due to different control granularity of flows, the number of flow entries required by different flows is varied. There is no pubic statistical data regarding this problem to the best of our knowledge, so we assume the number of flow entries required by the flows follows uniform distribution of $[s-\delta_s, s+\delta_s]$, where s is the average number of flow entries needed by the flows, and δ_s is the maximum deviation. We set s to be 30 and δ_s to be 10 by default. We carefully arrange the network configuration, to make sure the network is not congested in the initial and final state, which means that in both cases the traffic carried by every link is less than its capacity and every switch has enough flow entry space to route the flows passing it.

We first validate the effectiveness of the algorithms by comparing the performance of the optimal solution and our heuristic algorithm. Then we show how network parameters of flow table size and the number of flows influence the system performance.

4.2 Comparing Our Algorithm with the Optimal Solution

We first observe the performance of *SortedSeq* and *Optimal*, under the scenario where flow table space is heavily utilized. There are 45 flows in the B4 network, and link utilization is not high($<$ 50%). We examine the system performance metrics in each step, in terms of max link utilization and max flow table utilization, and compare the total number of steps. The results are shown in Fig. 2.

From the results, we can observe that for *Optimal* the update is finished in 2 steps. While *SortedSeq* uses 3 steps, and gets similar value of max link utilization compared with *Optimal*. Regarding the max flow table utilization as shown in Fig. 2 (b), we observe that for both algorithms the max usage of flow table becomes close to 100% in the first step, since the switches have to carry two sets of forwarding rules at the beginning. Furthermore, the max flow table utilization of *SortedSeq* is almost the same with *Optimal*, which is consistent with the observations in Fig. 2 (a). The running time of Optimal is about 10 seconds, while running time of our algorithm is only less than 0.02 seconds, which is about 500 times faster than the optimal solution. In general, we can see the two algorithms succeed in finishing the update without violation of resource limits, and our solution has near performance to the optimal one. To verify the robustness of the algorithms, the impact of network parameters on algorithm performance is analyzed in the following sub-sections.

4.3 Impact of Flow Table Size

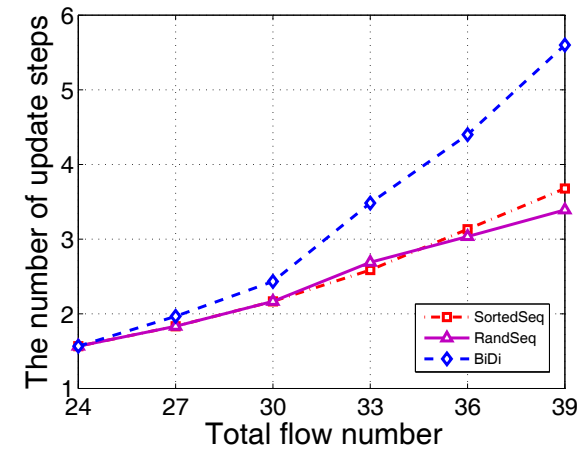
Since flow table is an important resource constraint in SDN, we carry out simulations to evaluate how flow table size can impact the update process. There are 40 flows to update, and we set average link utilization per flow to 2%. Since the link load is relatively low, it is not the bottleneck factor restricting update process. Then we change the average flow table utilization per flow from 2.67% to 4%, and run the simulation 50 times. The network performances are evaluated, and we present the results in Fig. 3.

We observe that the number of update steps rises when flow table size decreases as shown in Fig. 3 (a). *BiDi* needs the most number of steps, since the bidirectional searching method cannot update as many flows as possible in a step. *SortedSeq* and *RandSeq* complete the update with almost the same number of steps, but considering update success rate shown in Fig. 3 (b), *SortedSeq* is better. The results demonstrate that when flow table size is very limited, our algorithm is effective and efficient. Furthermore, if we desire an even more faster solution, the simplified version of our algorithm, *RandSeq*, is a good choice, but at the expense of lower success ratio.

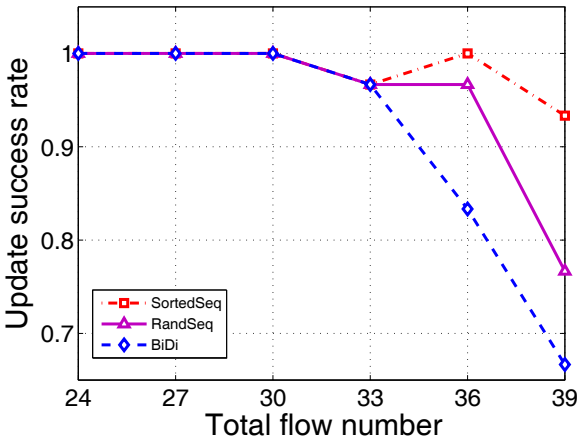
4.4 Impact of Flow Number

We now study how the number of flows impacts on network performance. The simulation scenario is similar to that in the previous sub-section. The link capacity and flow table size are fixed. We increase the number of flows in the network from 24 to 39, while keeping the average link utilization per flow to be 4% and average flow entries required per flow to be 2.67% of the flow table size. The results are shown in Fig. 4.

Concerning the number of steps, as shown in Fig. 4 (a), when the number of flows increases, the algorithms take



(a) The number of steps



(b) Update success rate

Figure 4: Performance during the update process as function of the flow number: (a) the number of steps and (b) update success rate.

more steps to finish the update. *BiDi* uses the largest number of steps when the flow density is large, while *SortedSeq* is much faster. *RandSeq* also performs good in terms of update steps, but with regard to success rate as depicted by Fig. 4 (b), its success ratio is lower than our algorithm when the flow number increases. *BiDi* has the lowest update success rate, because its mechanism does not pay special attention to the busiest switches of the network. When the total number of flows is relatively large (greater than 33), none of the algorithms works well, because the flow table is in heavy use and there is little space left to install new rules. In this case, our algorithm is more robust, which finishes about 15% more update. In conclusion, simulation results indicate that our proposed algorithm is robust and efficient when the total number of flow increases.

5. CONCLUSION

In this paper, we propose a general framework to formulate the multi-flow update problem and present a polynomial-time heuristic algorithm, which aims at completing the update in the shortest time under both link bandwidth and flow table size constraints. Extensive simulations based on real topologies show that the proposed algorithm is efficient and has pretty good performance compared with the optimal solution.

6. ACKNOWLEDGEMENTS

The authors wish to thank the anoymous reviewers for their valuable feedback on improving this paper. This work is supported by the National Basic Research Program of China (973 Program) under grants 2013CB329105, and the National Nature Science Foundation of China (Grant No. 61273214).

7. REFERENCES

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," in *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Apr. 2008.

[2] S. Ghorbani and M. Caesar, "Walk the line: consistent network updates with bandwidth guarantees," in *Proc. ACM HotSDN 2012* (Helsinki, Finland), August 13, 2012, pp. 67–72.

[3] C. Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven WAN," in *Proc. ACM SIGCOMM 2013* (Hong Kong, China), August 12-16, 2013, pp. 15–26.

[4] N. P. Katta, J. Rexford, and D. Walker, "Incremental consistent updates," in *Proc. ACM HotSDN 2013*, August 16, 2013, pp. 49–54.

[5] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proc. ACM SIGCOMM 2012* (Helsinki, Finland), August 13-17, 2012, pp. 323–334.

[6] M. Reitblatt, N. Foster, *et al.*, "Consistent updates for software-defined networks: Change you can believe in!," in *Proc. HotNets 2011* (Cambridge, MA), November 14-15, 2011, pp. 7:1–7:6.

[7] S. Vissicchio, L. Vanbever, L. Cittadini, G. Xie, and O. Bonaventure, "Safe Updates of Hybrid SDN Networks," UCL, 2013, pp. 1–12.

[8] J. Löfberg, "YALMIP: A toolbox for modeling and optimization in MATLAB Computer Aided Control Systems Design," in Proc. IEEE CACSD 2004 (Taiwan, China), September 2-4, 2004, pp. 284–289.

[9] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, and others, "B4: Experience with a globally-deployed software defined WAN," in *Proc. ACM SIGCOMM 2013*, (Hong Kong, China), August 12-16, 2013, pp. 3–14.