Fast Network Configuration in Software Defined Networking

Stefan Achleitner, Novella Bartolini, *Senior Member IEEE,* Ting He, *Senior Member IEEE*, Thomas La Porta, *Fellow IEEE*, Diman Zad Tootaghaj

***Abstract*—*Software Defined Networking (SDN)* provides a framework to dynamically adjust and re-program the data plane with the use of flow rules. The realization of highly adaptive SDNs with the ability to respond to changing demands or recover after a network failure in a short period of time, hinges on efficient updates of flow rules. We model the time to deploy a set of flow rules by the update time at the bottleneck switch, and formulate the problem of selecting paths to minimize the deployment time under feasibility constraints as a *mixed integer linear program (MILP)*. To reduce the computation time of determining flow rules, we propose efficient heuristics designed to approximate the minimum-deployment-time solution by relaxing the MILP or selecting the paths sequentially. Through extensive simulations we show that our algorithms outperform current, shortest path based solutions by reducing the total network configuration time up to 55% while having similar packet loss, in the considered scenarios. We also demonstrate that in a networked environment with a certain fraction of failed links, our algorithms are able to reduce the average time to reestablish disrupted flows by 40%.**

***Index Terms*—Software-defined networks, Configuration management, Mathematical optimization**

I. Introduction

It is widely recognized that flow configuration, i.e., the configuration of flow forwarding rules at the SDN switches, is critical for SDN operation. Configuration updates due to changed flows must be performed consistently and quickly to avoid congestion [1], [2], [3], delays [4], loops and policy violations [3], [5], [6]. In particular, the need to reestablish disrupted flows caused by failing links as the result of a failure, congestion or attack on the network infrastructure, motivates the requirement to perform fast network reconfiguration.

The time required for deploying a given flow configuration is dominated by the time to insert/update flow rules in the *ternary content addressable memory (TCAM)* of each involved switch. Previous works such as [4], [7], [8], [9] report different per-rule update times ranging from $10ms$ up to $400ms$ depending on different models. The deployment time of network-wide flow rules on a data-center sized topology is not negligible since rule updates in a single switch must be performed sequentially to ensure consistency of rules [4], [5], [6], while multiple switches can be updated in parallel [5], [6].

Moreover, practical networks can require a huge number of rule updates to support new flows. In [10] the authors show an example of a 1500-server cluster with a medial flow arrival rate of $100K$ per second. A flow configuration algorithm heedless of the update time can generate thousands of updates per second at some switches, which causes large flow setup delays and severe performance degradation for the majority of flows (which are short-lived). Meanwhile, it can be computationally challenging to compute a flow configuration that can satisfy traffic engineering requirements (e.g., bounded delay, balanced load), while balancing the rule updates across switches.

If the focus of network providers is on in-process flows, shortest path routing can be chosen. However, alleviating congestion or accommodating new flows requires a potentially disruptive amount of time to install new flow rules. Our solution provides a trade-off for routing optimization and reduces the time to reconfigure the network $45-55\%$ on average in typical scenarios, while experiencing negligible packet loss. In comparison to previous works [9], [11] focusing on efficient network updates in SDN, we also demonstrate that our framework is able to reestablish disrupted flows in SDN more than 40% faster compared to the baseline.

In this paper, we develop an optimization framework and associate flow configuration algorithms to support fast-changing flow demands in SDNs, by taking into account both the time to compute a new flow configuration at the controller and the time to deploy this configuration at the switches. The focus of our work are networks with high flow dynamics, such as data-centers, corporate networks or IoT deployments. As a trade-off for faster configuration time, our framework generates slightly longer paths on average, compared to an algorithm which aims at satisfying flows through paths of minimum length. In a recent study, He et al. [7] observe a per-hop delay on SDN-enabled switches of 0.15ms on average, demonstrating a limited impact of longer paths in data-center networks. In such scenarios our algorithms achieve an improvement in the network configuration time in the order of 10^2ms.

Our framework aims at minimizing the configuration time while satisfying flow demands under link capacity and path length constraints. Additional constraints can be formulated to represent other traffic engineering objectives, such as routing cost. Specifically, we make the following contributions:

- ***Formulation of the optimization problem***: We formulate a *mixed integer linear program (MILP)* to compute the flow configuration with the minimum deployment time under feasibility (e.g., link capacity, path length) constraints. We show that this problem is NP-hard.
- ***Development of efficient heuristics***: We develop a set of heuristics designed to approximate the minimum time to deploy a network configuration.
- ***Evaluation of proposed heuristics***: We show via extensive simulations on sample network topologies that our algorithms reduce the average network configuration time up to 55%, with respect to traditional shortest path approaches.

S. Achleitner is with Palo Alto Networks. The work was performed when he was at the Pennsylvania State University.
T. He, T. La Porta and D. Z. Tootaghaj are with the Pennsylvania State University
N. Bartolini is with the Sapienza University of Rome

II. Related Work

The procedure of deploying flow rules on SDN switches has been discussed abundantly in the literature. The works in [2], [5], [6], [9], [12] focus on policy-preserving update procedures with different techniques.

The work of Vissicchio et al. [5], [6] aims at finding a schedule of updates that prevents inconsistencies and policy violations. Their algorithm works very well for a single flow update, but makes the assumption that each flow is independent of other flows. As the authors state, the algorithm might result in high packet loss rates during the update transition due to congestion since link capacity is not considered. Brandt et al. [2] address the problem of scheduling updates so as to avoid congestion with splittable and unsplittable flows. However, even though OpenFlow 1.3 supports group tables with unequal splitting of flows, most implementations of OpenFlow in current switches do not support splittable flows. The work of Mizrahi et al. [3] proposes an update of OpenFlow (included in OpenFlow 1.5) based on the support of the Time Precision Protocol (TPP) at each switch to avoid inconsistencies in global updates. While our work focuses on optimizing the time to compute and deploy a configuration in an SDN-based network, we discuss in Section VII-A how our approach can be combined with existing policy-preserving update procedures.

The authors of [13], [14] propose a set of abstract operations to guarantee consistency between updated configurations in SDN. Ludwig et al. [15] discuss the problem of consistently updating a network in order to ensure waypoint enforcement and loop freedom. Updating routes in a loop-free manner in SDN is further discussed by Foerster et al. [16].

Inconsistencies in SDN-based network configurations may occur when rule updates, which are communicated from the controller to the data-plane via an asynchronous network, arrive out-of-order. To address such inconsistencies Zheng et al. [17], [18] present a study of algorithms to configure SDNs in a way that individual node updates can be scheduled at specific times. Further, Mizrahi et al. [19] propose a method to perform accurate time-based rule updates in SDN by using TimeFlips, which are implemented with the usage of time-stamp fields in the TCAM memory of a network switch.

The authors of [4], [7], [8] aim to improve the efficiency of flow rule updates by optimizing the deployment procedure at a switch's TCAM. One factor that affects efficient configuration of SDNs, as pointed out in these papers, is the per-rule update time which significantly varies by different SDN hardware switch vendors as pointed out by the authors of [7], [20]. Wen et al. [4] state a per-rule update time of 12-15ms, while [9] states a per-rule update time reaching 18ms. In [8], 40-50 rules per second is reported, and [7], [20] discuss the high dependency of rule updates on vendor-specific TCAMs.

The existing literature points out the negative effects of increasing the number of rules on a switch, including increased deployment time of rules [9], [20] or increased packet classification time [8]. Meanwhile, reducing rules by combining them, e.g. with the use of address prefixes, can have severe security implications as discussed in [21], [22].

The work from Paris et al. [1] considers the variability of flow demand as a limitation to the applicability of solutions even as simple as a Linear Program (LP) formulation of the problem of flow reconfiguration. The authors consider an iterative execution of a global flow optimization problem, and propose the adoption of sub-optimal solution with gradually lower optimality gap at given time instants.

While the above works focus on reducing the update time at a single switch [4], [7], [8] or minimizing the link leasing cost [1], we focus on reducing the total network configuration time across all switches by optimizing the updates of rules.

III. Problem Formulation

In this section we formulate the problem of minimizing the network update time as an optimization problem. The nomenclature used in this paper is shown in Table I.

TABLE I: Nomenclature and notation

Notations	Descriptions
Input parameters of the network	
V	Set of nodes in a network
E	Set of links in a network
P	Set of possible paths in a network
χ	Set of pre-computed candidate paths
$c_{ij}, \widetilde{c}_{ij}$	Total/residual capacity of link (i,j)
H, H_n	Set of all/new flows
s_h, t_h	Source/destination of flow h
d_h	Rate of flow h generated at the source node
F_i	Set of flow rules to be updated/inserted on a switch i
Q_i	Current flow table size on switch i
w_i	Per-rule update/insertion time on switch i
Optimization problem parameters	
φ_h	Upper bound for the number of hops on a path
a_{ij}^h	Current routing of flow h through link (i,j) and switch i
k_{ij}^h	Current routing of flow h through link (i,j)
u_i^h	Current routing of flow h at switch i
Optimization problem decision variables	
v_i^h	Path sequence variable of flow h on switch i
z_p^h	Path selection variable of flow h on path p
x_{ij}^h	Link selection variable of flow h on link (i,j)
R	Maximum update time minimized over all switches
τ_i	Deployment time on switch i (used in evaluations)

A. Network Model

We model a network as an undirected graph $\mathcal{G} = (V,\ E)$, where V is the set of switches and E the set of links. Each link $(i,j) \in E$ is associated with a capacity c_{ij}. Let H denote the current set of flows on this network. Each flow $h \in H$ is associated with a source s_h, a destination t_h, and a demand d_h specifying the flow rate. Depending on the construction of SDN rules, which can be specified by the network operator, a flow h can be a single flow identified by a pair of IP addresses, as well as a batch of flows identified by an IP-prefix. Our framework will compute an assignment of flows on a network topology, where the detailed definition of a flow h can be set by the network operator. In our framework, b_i^h denotes the volume of flow h generated by switch i, given by $b_i^h = d_h$ if $i = s_h, b_i^h = -d_h$ if $i = t_h,$ and $b_i^h = 0$ otherwise.

We assume single-path routing (i.e., unsplittable flows) in this work, since single-path routing was the technique typically used in the SDN controllers (POX, Floodlight, OpenDayLight) and switches (Brocade ICX 6610) at the time we performed our experiments. We expect that SDNs will be used for multi-path routing and are planning to extend our models to this

technique in the future. In our optimization models this can be achieved by directly using fractional variables instead of integer variables by rounding a multi-path solution to a single-path solution as we will further discuss in Sections IV and V.

B. Model of Network Update Time

The delay between the flow arrival and the time the network is reconfigured to support it includes four components: (a) queueing delay, where the SDN controller waits for a specific amount of time or a certain number of flow demands to be processed in a batch, (b)computation delay, when the controller calculates the new configuration, (c) communication delay, which is the time to communicate the new configuration (i.e., flow rules) to the involved switches, and (d) deployment delay, which is the time to deploy the rules at the switches. The dominating factors are (b) and (d) as explained below.

The queueing delay (a) T_B defines the time an SDN controller waits until a batch of flows is processed and the network configuration is updated. In highly dynamic environments such as data-centers or IoT deployments where the network is based on SDN, it is beneficial to set the queueing delay to a specified time threshold such as the lower bound of TCP retransmission timeout, e.g. $1s$ as suggested in RFC 793 [23]. As a second option, the queueing delay can also be defined by setting a threshold to the number of *packet-in* messages (i.e. arriving flows) in the SDN controller. In a real network environment such an approach must be used with caution since the time between two newly arriving flows is undefined. Therefore it is recommended to use a combination of both approaches and trigger a new network configuration whenever the first threshold condition, waiting time or number of *packet-in* messages, is satisfied. Similar models are used for queueing in Cisco switches [24], considering both delay and number of arrived packets. As we discuss in Section VI, we use a queue size threshold of 40 flows in our evaluations.

The (southbound) communication delay (c) $T_{S,i}$ between the controller and the switch i consists of the *transmission delay* plus the *propagation delay*:

$$T_{S,i} = \frac{\text{OpenFlow packet size}}{\text{Control network bandwidth}} + \frac{D_i}{0.75 \cdot c}, \qquad (1)$$

where c is the speed of light[1], and D_i is the distance from the controller to the switch i. The second term in Equation 1 depends on the distance between the SDN controller and switches. Within a range of $10^3 - 10^4$ meters this delay is in the range of $10^{-6}s - 10^{-5}$s, which can be assumed for most data-center, IoT and campus network deployments where SDN is typically applied. In such scenarios this term is orders of magnitude smaller compared to the first term, and thus $T_{S,i}$ is roughly the same for all the switches in the same SDN.

The typical bandwidth between the controller and the switches ranges from *17Mbps* [25] to *276Mbps* [26] for modern SDNs. Assuming a bandwidth of *276Mbps* and a maximum OpenFlow packet size of $64KB$ [27], $T_S \approx 1.8ms$ for all switches.

The computation (b) and the deployment (d) delays are in the order of 10–$1000ms$, dominating the total delay. The computation delay depends on the controller hardware and the flow configuration algorithm, while the deployment delay depends on the switch hardware and the flow configuration itself. Our focus is to minimize these delays by designing efficient flow configuration algorithms, with fast deployment.

To this end, we analyze the impact of flow configuration on the deployment time. In SDN, deploying a new configuration requires updating the corresponding rules in the flow tables of the switches. As observed in [9], the update time of a flow table tends to grow with the number of updated rules, as updates on a given switch must be performed sequentially to ensure consistency [4], [5], [6]. This observation implies that given a set F_i of rules to be updated at switch i, the total update time at i, denoted by τ_i, satisfies

$$\tau_i(F_i) = \sum_{r \in F_i} w_i(r), \qquad (2)$$

where $w_i(r)$ denotes the time to update a single rule $r \in F_i$. As observed in [4], [9], the rule update time $w_i(r)$ is generally an increasing function of the current flow table size $Q_i(r)$. In production networks, the flow table size is typically much larger than the number of updated rules. Since each rule update changes the flow table size by at most one (if adding/deleting a rule), this means that $Q_i(r)$ can be approximated to a constant for all $r \in F_i$, which implies that $w_i(r) \approx w_i$ for all $r \in F_i$. Under this assumption, (2) is simplified to $\tau_i(F_i) = |F_i| w_i$.

While updates on a single switch are performed sequentially, updates on different switches can usually be done concurrently. This implies that the total time $\tau(F)$ to deploy a set of updated rules $F = \bigcup_{i \in V} F_i$ across all the switches will be determined by the switch with the maximum update time, i.e.,

$$\tau(F) = \max_{i \in V} w_i \cdot |F_i|. \qquad (3)$$

We will discuss the extension of our approach to policy-preserving updates [5], [6] in Section VII-A.

C. Flow rule deployment times

As discussed in the literature (e.g. [4], [7], [8], [20]) the time to insert/update a flow rule in the TCAM of SDN-enabled switches varies significantly depending on different hardware vendors and rule priorities. Inserting a rule with higher priority than most other rules may require an expensive re-ordering of the rules in the hardware TCAM memory, which increases the update time as discussed by *He at al.* [7].

He et al. [7] perform measurements on three switches (IBM, Intel, Broadcom) and report times ranging from as low as *3ms* on an empty switch, to *∼100ms* on a switch that already has a few hundred rules deployed. Katta et al. [8] perform measurements on a Pica8 switch and report rule deployment times between *12ms* and *80ms*. Kuzniar et al. [20] evaluate three different hardware platforms (HP, Pica8, Dell) and report per-rule installation/modification times between *33-400ms*. Wen et al. [4] discuss a hardware extension to reduce deployment times to *12-15ms* per flow rule. Measurements of the per-rule deployment times we conducted on a Brocade ICX 6610 switch range from *350-450ms*.

[1] 0.75 in the denominator is a reduction factor for c on copper wire.

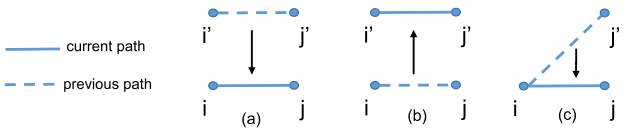


Fig. 1: Updates at i: rule addition (a), removal (b), change (c).

D. The MinUpdateTime Problem

Our goal is to minimize the network update time by selecting a path for each flow $h \in H$ which minimizes the maximum number of updated rules per switch, weighted by its per-rule update time. We represent the path selection by a decision variable $x_{ij}^h \in \{0, 1\}$, which indicates if flow h traverses link $(i, j) \in E$. Note that although the links are undirected, the traversals of links are directed. The decision variables must satisfy the *flow conservation constraint*

$$\sum_{j \in V} x_{ij}^h = \sum_{k \in V} x_{ki}^h + \text{sgn}(b_i^h), \quad \forall h \in H, i \in V, \tag{4}$$

where $\text{sgn}(y)$ is the sign function that is 1 if $y > 0$, -1 if $y < 0$, and 0 if $y = 0$. This constraint ensures that the path formed by links with $x_{ij}^h = 1$ can route the flow from s_h to t_h. These variables must also satisfy the *link capacity constraint*

$$\sum_{h \in H} (x_{ij}^h + x_{ji}^h) \cdot d_h \leq c_{ij}, \quad \forall (i, j) \in E, \tag{5}$$

where c_{ij} is the capacity of link $(i, j) \in E$.

While (4) and (5) are typical constraints in flow configuration, we have a novel objective, i.e., minimizing the network update time. As discussed for equation (3), this objective depends on the number of updated rules at each switch, which depends not only on the current configuration, but also on the previous configuration.

To capture the previous path selection, we introduce a parameter $k_{ij}^h \in \{0, 1\}$, which indicates whether flow $h \in H$ was using link (i, j) before the update. Note that for a newly arrived flow h, $k_{ij}^h = 0$ for all $(i, j) \in E$. Then we have the following observations: (1) if $k_{ij}^h = 1$ but flow h no longer traverses switch i, then the rule for forwarding h needs to be removed from the flow table of switch i; (2) if $k_{ij}^h = 0$ but flow h traverses link (i, j), then either flow h used to traverse a different link (i, j') $(j' \neq j)$ at switch i or it did not traverse switch i at all; in both cases a rule at switch i needs to be updated (modified or inserted). The above covers all possible cases of rule updates.

To write the number of updated rules at a given switch as a function of x_{ij}^h's, we define another parameter a_{ij}^h:

$$a_{ij}^h \triangleq 1 - k_{ij}^h - u_i^h, \tag{6}$$

where $u_i^h \triangleq 1 - \prod_{j \in V}(1 - k_{ij}^h)$. From this definition, $u_i^h = 1$ if flow h used to traverse switch i and $u_i^h = 0$ otherwise. Parameter a_{ij}^h can take 1, 0, or -1, where $a_{ij}^h = 1$ if flow h did not traverse switch i; $a_{ij}^h = 0$ if flow h did not traverse link (i, j) but traversed switch i, and $a_{ij}^h = -1$ if flow h traversed link (i, j). For a new flow k_{ij}^h will be 0 and a_{ij}^h will be 1, for all $(i, j) \in E$. Note that k_{ij}^h and a_{ij}^h are determined by the previous path selection and are thus constants.

Lemma 1. *If all the paths carrying flows are cycle-free, then the number of updated rules at switch $i \in V$ is given by* $\sum_{h \in H} \sum_{j:(i,j) \in E} (a_{ij}^h \cdot x_{ij}^h + k_{ij}^h)$.

Proof. It suffices to show that for a given flow h, $\sum_{j:(i,j) \in E} (a_{ij}^h \cdot x_{ij}^h + k_{ij}^h) = 1$ if there is a rule update for flow h at switch i and $\sum_{j:(i,j) \in E} (a_{ij}^h \cdot x_{ij}^h + k_{ij}^h) = 0$ otherwise.

To see this, consider the three cases of rule update as illustrated in Figure 1. In case (a), i.e., flow h was not traversing switch i but now traverses it via link (i, j), we have $x_{ij}^h = 1$, $x_{ik}^h = 0$ for all $k \neq j$, $a_{ik}^h = 1$ and $k_{ik}^h = 0$ for all links $(i, k) \in E$, implying $\sum_{k:(i,k) \in E} (a_{ik}^h \cdot x_{ik}^h + k_{ik}^h) = \sum_{k:(i,k) \in E} x_{ik}^h = 1$. In case (b), i.e., flow h was traversing link (i, j) but now no longer traverses switch i, we have $x_{ik}^h = 0$ for all $(i, k) \in E$, $k_{ij}^h = 1$, and $k_{ik}^h = 0$ for all $k \neq j$, implying $\sum_{k:(i,k) \in E} (a_{ik}^h \cdot x_{ik}^h + k_{ik}^h) = \sum_{k:(i,k) \in E} k_{ik}^h = 1$. In case (c), i.e., flow h was traversing switch i via link (i, j') but now traverses it via link (i, j), we have $x_{ij'}^h = 0$, $k_{ij'}^h = 1$, $a_{ij'}^h = -1$, $a_{ij}^h = 0$, $k_{ij}^h = 0$, and $x_{ik}^h = k_{ik}^h = 0$ for the other links $(i, k) \in E$ with $k \notin \{j, j'\}$, implying $\sum_{k:(i,k) \in E} (a_{ik}^h \cdot x_{ik}^h + k_{ik}^h) = (a_{ij}^h \cdot x_{ij}^h + k_{ij}^h) + (a_{ij'}^h \cdot x_{ij'}^h + k_{ij'}^h) = 1$.

Moreover, if flow h traverses switch i via the same link (i, j) at both times, then $a_{ij}^h = -1$, $x_{ij}^h = 1$, and $k_{ij}^h = 1$, while $x_{ik}^h = k_{ik}^h = 0$ for the other $(i, k) \in E$ $(k \neq j)$, yielding $\sum_{k:(i,k) \in E} (a_{ik}^h \cdot x_{ik}^h + k_{ik}^h) = 0$. Finally, if flow h does not traverse switch i at both times, then $x_{ij}^h = k_{ij}^h = 0$ for all $(i, j) \in E$, and hence $\sum_{j:(i,j) \in E} (a_{ij}^h \cdot x_{ij}^h + k_{ij}^h) = 0$. □

Note that Lemma 1 only considers flows currently existing in the network.

Remark: We ignore terminated flows as the rules of terminated flows will expire automatically according to the OpenFlow protocol [27] and do not require explicit updates. We now formulate our optimization problem, referred to as the *MinUpdateTime problem* as shown in (7).

$$\min R \tag{7a}$$

$$\text{s.t. } w_i \sum_{h \in H} \sum_{j:(i,j) \in E} (a_{ij}^h \cdot x_{ij}^h + k_{ij}^h) \leq R, \quad \forall i \in V, \tag{7b}$$

$$\sum_{j \in V} x_{ij}^h = \sum_{k \in V} x_{ki}^h + \text{sgn}(b_i^h), \quad \forall h \in H, i \in V, \tag{7c}$$

$$\sum_{h \in H} (x_{ij}^h + x_{ji}^h) \cdot d_h \leq c_{ij}, \quad \forall (i, j) \in E, \tag{7d}$$

$$\sum_{i,j \in E} x_{ij}^h \leq \varphi_h, \quad \forall h \in H, \tag{7e}$$

$$v_i^h - v_j^h + 1 \leq |V|(1 - x_{ij}^h), \forall h \in H, (i, j) \in E \tag{7f}$$

$$v_i^h \in \{1, \ldots, |V|\} \quad \forall h \in H, i \in V \tag{7g}$$

$$x_{ij}^h \in \{0, 1\}, \quad \forall h \in H, (i, j) \in E. \tag{7h}$$

Constraints (7c, 7d) ensure that selected paths route flows within link capacities. Constraint (7e) ensures that a flow h is routed on a path not exceeding a length of φ_h hops which allows to control the end-to-end delay of a flow. Notice that a path cannot traverse the same switch multiple times (i.e., cannot contain cycles), as this will cause conflicts in the forwarding rules. Hence, constraints (7f, 7g) ensure cycle elimination according to the Miller–Tucker–Zemlin technique [28] with the introduction of path sequence variables v_i^h.

Constraint (7h) ensures that each flow is routed along a single path, and constraint (7b) ensures that the objective value R is no smaller than the maximum update time of any switch. Problem (7) minimizes the network update time due to flow table updates. We will design efficient algorithms for this optimization and evaluate the computation time separately (Section VI). The above formulation can be extended to model additional traffic engineering objectives, e.g., by adding constraints on routing cost or load balancing metrics. We leave the study of these additional constraints to future work.

E. The Non-disruptive MinUpdateTime Problem

The solution to the MinUpdateTime problem (7) can reroute existing flows, which can cause temporary disruption to applications relying on these flows. For disruption-sensitive applications (e.g. real-time streaming or high performance computing jobs in data center networks), it is desirable that the configuration is calculated for the new flows only, leaving the existing flows undisrupted. We therefore define Non-disruptive MinUpdateTime problem that does not re-route existing flows. Let $H_n \subseteq H$ denote the set of newly arrived flows. We consider a variation of (7) that only selects paths for flows in H_n. Accordingly, constraints (7c, 7h) are only imposed on x_{ij}^h where $h \in H_n$. The link capacity constraint (7d) is revised by replacing the original link capacity c_{ij} ($(i,j) \in E$) by the *residual link capacity* $\widetilde{c}_{ij}$, defined as

$$\widetilde{c}_{ij} \triangleq c_{ij} - \sum_{h \in H \setminus H_n} (k_{ij}^h + k_{ji}^h) d_h. \tag{8}$$

The only substantial change is in how we count the number of rule updates in constraint (7b). Since the updates are only for the new flows (to add their rules), and $a_{ij}^h \equiv 1$ and $k_{ij}^h \equiv 0$ for a new flow $h \in H_n$, $\forall (i,j) \in E$, we revise constraint (7b) as follows,

$$w_i \sum_{h \in H_n} \sum_{j:(i,j)\in E} x_{ij}^h \leq R, \quad \forall i \in V. \tag{9}$$

Putting the above changes together gives the following optimization for the *Non-disruptive MinUpdateTime problem*:

$$\min R \tag{10a}$$

$$\text{s.t. } w_i \sum_{h \in H_n} \sum_{j:(i,j)\in E} x_{ij}^h \leq R, \qquad \forall i \in V, \tag{10b}$$

$$\sum_{j \in V} x_{ij}^h = \sum_{k \in V} x_{ki}^h + \text{sgn}(b_i^h), \qquad \forall h \in H_n, i \in V, \tag{10c}$$

$$\sum_{h \in H_n} (x_{ij}^h + x_{ji}^h) \cdot d_h \leq \widetilde{c}_{ij}, \qquad \forall (i,j) \in E, \tag{10d}$$

$$\sum_{i,j \in E} x_{ij}^h \leq \varphi_h, \qquad \forall h \in H, \tag{10e}$$

$$v_i^h - v_j^h + 1 \leq |V|(1 - x_{ij}^h), \forall h \in H_n, (i,j) \in E \tag{10f}$$

$$v_i^h \in \{1, \ldots, |V|\} \qquad \forall h \in H_n, i \in V \tag{10g}$$

$$x_{ij}^h \in \{0,\ 1\}, \qquad \forall h \in H_n, (i,j) \in E. \tag{10h}$$

F. Properties of the Problems

Checking feasibility of our MinUpdateTime problem (7) or Non-disruptive MinUpdateTime problem (10) is known as the *specified demand integral multi-commodity flow problem* and is known to be NP-complete [29]. However, in the following we prove that even if feasibility is given, problems (7) and (10) are still NP-hard.

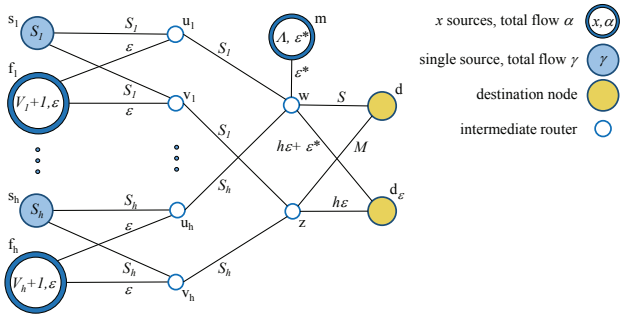


Fig. 2: NP-hardness of MinUpdateTime

Theorem 2. *Both the MinUpdateTime problem (7) and the Non-disruptive MinUpdateTime problem (10) are NP-hard, even if they are known to be feasible.*

Proof. We show the NP-hardness of the Non-Disruptive MinUpdateTime problem (10) by reducing the classic formulation of the Knapsack problem to a specific instance of the Non-Disruptive MinUpdateTime problem in polynomial time. We select an instance of Non-Disruptive MinUpdateTime which starts with an offloaded network, so that the same proof will hold for both the general version (7) and the non-disruptive (10) version. We recall that the Knapsack problem considers a knapsack of size S, and a set of items I, where each item $i \in I$ has a size S_i and a value $V_i > 0$. The problem is to find a subset $I' \subseteq I$ such that $S(I') \leq S$ and $V(I')$ is *maximized*, where $S(I') = \sum_{i \in I'} S_i$ and $V(I') = \sum_{i \in I'} V_i$. Given an instance of the Knapsack problem, we construct an instance of the Non-Disruptive MinUpdateTime problem as illustrated in Figure 2. For each Knapsack item $i \in I$, with $h = |I|$, we construct: a single source node s_i, requiring to transmit a flow of S_i units to a destination node d; a node f_i, sending $V_i + 1$ flows to a destination node d_ε, for a total amount of ε units of flow, where ε is an arbitrarily small value, with $\varepsilon < \min_{i \in I} S_i$; the intermediate nodes u_i and v_i connected to the source nodes through the links (s_i, u_i) and (s_i, v_i) of capacity S_i and links (f_i, u_i) and (f_i, v_i) of capacity ε. We then have two intermediate nodes w and z such that each node u_i, $i \in I$, is connected to w via a link of capacity S_i, and each node v_i is connected to z via a link of the same capacity. Node w is also connected to a node m, with link (m, w) of capacity ε^*. The node m generates Λ equal flows directed to d_ε, with $\Lambda \triangleq \lceil \sum_{i \in I} (V_i + 1) \rceil$, for a total flow equal to ε^*. Finally, nodes w and z are connected to the destination nodes d and d_ε. The capacity of the link (w, d) is S, the same as the Knapsack.

Notice that this construction ensures that any feasible routing will allow either a single flow of size S_i through node u_i, and $V_i + 1$ flows of total size ε through v_i, or vice versa, i.e. a flow of size S_i through v_i and $V_i + 1$ flows of size ε through u_i. The capacity of link (w, d) allows only a limited number of sources among $\{s_1, \ldots, s_h\}$ to route their flow through node w to d. The capacity limitation on link (w, d) is the key to the correspondence of our construction to the original Knapsack

problem. Link (w, d) can only accommodate a number of single source flows S_i up to capacity S. All the other single source flows, exceeding the capacity S of link (w, d) will be routed to d via node z. In fact, the capacity of the link (z, d) is an arbitrarily large value M, where $M \geq \sum_{i \in I} S_i$, having room for all the unique source flows to be routed to node d.

The capacity of links (w, d_ε) and (z, d_ε) is equal to $h\varepsilon + \varepsilon^*$ and to $h\varepsilon$, respectively. In order to force the routing of as many single source flows as possible through link (w, d) we set ε and ε^* such that $h\varepsilon + \varepsilon^* < \min_{i \in I} S_i$. Furthermore, we set $h\varepsilon < \varepsilon^*/\Lambda$ so that the only route available to the flows generated by m is across link (w, d_ε). Notice that this construction, together with the capacity limitations of the considered links, imposes that the router with maximum number of rule updates is always w, regardless of the flow routing decision. MinUpdateTime should therefore minimize the number of rules of node w. We introduce a binary decision variable z_i, to reflect the decision ($z_i = 1$) to route the single flow S_i from node s_i to node u_i and the multiple flows coming from node f_i will be routed through node v_i. If instead flow S_i is routed along link (s_i, v_i) ($z_i = 0$) the multiple flows from f_i will traverse link (f_i, u_i).

A solution is feasible for the constructed Non-Disruptive MinUpdateTime if and only if the selected flows S_i routed through link (w, d) for a capacity $\sum_{i \in I} z_i \cdot S_i$ do not exceed S. This is equivalent to selecting a subset of items $I' \subseteq I$ of the Knapsack problem, for which $i \in I'$ if $z_i = 1$, such that the size $S(I')$ does not exceed the Knapsack capacity S.

With this construction, the optimal number of rules on node w is $n(w) = \min_{z_i : i \in I}\{\Lambda + \sum_{i \in I}[z_i \cdot 1 + (1 - z_i) \cdot (V_i + 1)]\} = \min_{z_i : i \in I}[\Lambda + |I| + \sum_{i \in I} V_i \cdot (1 - z_i)]$ [2]. Solving the MinUpdateTime optimally also requires finding the values of the binary variables z_i that minimize $n(w)$ which is equivalent to maximizing $\sum_{i \in I} z_i \cdot V_i$, the value of the Knapsack, which concludes the proof. □

Discussion: The problems (7) and (10) implicitly assume that proper admission control has been implemented to guarantee the feasibility of the flow demand H. For unsplittable flows, the admission control problem itself is NP-hard, but can be approximated to a ratio of $O(\Delta\alpha^{-1} \log^2 |V|)$, where Δ is the maximum node degree and α the expansion of the network topology [30]. Here we assume that the demands are feasible to focus on the problem of minimizing the network update time among the feasible flow configurations.

IV. Unrestricted Path Selection Algorithms

To reconfigure the network under changes in flow demands, we consider the following algorithms: (i) a *shortest path algorithm* that routes the flows sequentially such that each flow uses the path with the smallest hop count with sufficient residual capacity to carry this flow, (ii) a *randomized rounding algorithm* that solves the optimization (7) (or (10)) via linear programming (LP) relaxation followed by randomized rounding, and (iii) a *minimax path algorithm* that routes the flows sequentially on the feasible path that minimizes the maximum per-rule update time among the traversed switches. The non-disruptive constraint can be applied to all algorithms.

These algorithms are considered for different reasons: (i) is a baseline that is widely used in practice, (ii) is a standard technique to approximate the solution of MILPs, and (iii) solves a sequential variation of our optimization (7, 10) by considering one flow at a time. All these algorithms are *unrestricted* in the sense that they allow flows to be routed along any path in the network, which is in contrast to *restricted* routing as presented later (Section V).

A. Shortest Path Algorithm

As a baseline, we try to route each flow on the shortest available path. Algorithm 1 considers the flows in an arbitrary order (line 4). For each flow h under consideration, it computes the subgraph formed by links with sufficient residual capacity to carry the flow (line 5), finds the shortest (i.e., minimum hop count) path in this subgraph between the source and the destination of flow h (line 6), and selects this path for the flow (line 8). After selecting a path, it updates the residual link capacities for the traversed links (line 9) before processing the next flow. At the end, the algorithm returns $\mathbf{x} = (x_{ij}^h)_{h \in H, (i,j) \in E}$ indicating the selected links (which form a path) for each flow. If the flow configuration has to be non-disruptive, then the initial residual capacity (line 3) will be the remaining link capacities left by existing flows, and the loop (lines 4–9) will only be performed for new flows. Since the primary objective of a shortest path algorithm is to follow the traffic engineering goal of minimizing path lengths, we do not have to add the constraint to limit the number of hops φ_h as discussed in Section III-D.

Algorithm 1 ShortestPath($\mathcal{G}$, H)

```
1: x = 0
2: for all (i, j) ∈ E do
3:     c̃_ij = c_ij
4: for all h ∈ H do
5:     G_h = G − {(i, j) ∈ E : c̃_ij < d_h}
6:     p_h = shortest_path(G_h, s_h, t_h)
7:     for all (i, j) ∈ p_h do
8:         x^h_ij = 1
9:         c̃_ij = c̃_ij − d_h
10: return x
```

Complexity: Finding the minimum hop count path between a given pair of nodes (line 6) can be done in $O(|E|)$ time via a breadth first search, and thus the computation for each flow (lines 5–9) has complexity $O(|E|)$. Since the algorithm routes the flows one by one, its overall complexity is $O(|H| \cdot |E|)$.

Remark: Although not designed to optimize the network update time, the shortest path algorithm strives to optimize a related metric: minimizing the hop count minimizes the number of rules installed across all the switches.

B. Randomized Rounding Algorithm

The MILP formulation of the MinUpdateTime problem (7) and its non-disruptive version (10) allows us to leverage standard techniques to the approximation of MILPs. In particular, *randomized rounding (RR)* is a widely used approach

[2]Notice that routing some multiple source flows from z through d and w to reach d_ε would be sub-optimal for the MinUpdateTime problem, as an equivalent solution with fewer rules installed on w is possible by routing these flows directly from z to d_ε through link (z, d_ε).

to approximate *integer linear programs (ILPs)* and MILPs. Below, we will discuss the application of randomized rounding to solve (7), and similar steps can be used to solve (10).

As shown in Algorithm 2, randomized rounding first (line 2) solves an LP relaxation of (7). Such a relaxation is obtained by replacing the integer variable x_{ij}^h by a fractional variable f_{ij}^h, and replacing the integer constraint (7h) by a linear constraint $f_{ij}^h \in [0, 1]$. Moreover, we remove the cycle avoidance constraints (7f). After solving the LP for a link-level fractional solution f_{ij}^h, we convert it into a path-level fractional solution, where f_p^h is the fraction of flow h routed on the cycle-free path p (line 3). This is done by the subroutine *fractional_path*, based on a maximum flow algorithm as explained below. Viewing the fraction f_p^h as the probability of routing flow h on path p, we round the fractional solution to an integral solution by randomly selecting a single path for each flow $h \in H$ according to the distribution $(f_p^h)_{p \in P}$ (line 7). Since after rounding, some links may exceed their capacities, we further check for sufficient residual capacity on the selected path before assigning a flow (line 8). If the path does not have sufficient residual capacity, the flow will be dropped.

Algorithm 2 RandomizedRounding($\mathcal{G}$, H)

1: $\mathbf{x} = \mathbf{0}$
2: $(f_{ij}^h)_{h \in H,(i,j) \in E}$ = MinUpdateTime_LP($\mathcal{G}$, H)
3: $(f_p^h)_{h \in H, p \in P}$ = fractional_path($\mathcal{G}$, $(f_{ij}^h)_{h \in H,(i,j) \in E}$)
4: **for all** $(i, j) \in E$ **do**
5: $\widetilde{c}_{ij} = c_{ij}$
6: **for all** $h \in H$ **do**
7: randomly sample p_h according to distribution $(f_p^h)_{p \in P}$
8: **if** $\widetilde{c}_{ij} \geq d_h$ for all $(i, j) \in p_h$ **then**
9: **for all** $(i, j) \in p_h$ **do**
10: $x_{ij}^h = 1$
11: $\widetilde{c}_{ij} = \widetilde{c}_{ij} - d_h$
12: **return** $\mathbf{x}$

As discussed, Algorithm 2 relies on a subroutine *fractional_path* (line 3) to convert the link-level fractional solution given by the LP relaxation to a path-level fractional solution used for rounding, with cycle-free paths. This subroutine can be implemented by *the Edmonds-Karp algorithm* [31] as follows. Given a network $\mathcal{G} = (V, E)$ with "link capacities" f_{ij}^h for each $(i, j) \in E$, we route a flow of demand d_h from node s_h to node t_h by repeating the following steps:

1) find a path p from s_h to t_h with bottleneck residual capacity f ($f > 0$) using breadth first search;
2) route f units of flow h on path p (i.e., $f_p^h = f$) and subtract f from the residual capacities of links on p.

We repeat the above steps until there is no path from s_h to t_h with positive residual capacity. Let P_h be the set of paths used for flow h and f_p^h ($p \in P_h$) the fraction of flow on each path. The flow conservation constraint (7c) guarantees that $\sum_{p \in P_h} f_p^h = 1$. Then applying this algorithm to all $h \in H$ provides the path-level fractional solution $(f_p^h)_{h \in H, p \in P}$ for all the flows, where $P = \bigcup_{h \in H} P_h$ ($f_p^h \equiv 0$ for all $p \in P \setminus P_h$).

Complexity: The bottleneck of the randomized rounding algorithm is solving the LP relaxation (line 2), which can be done in polynomial time using interior point methods. Specifically, the LP relaxation of (7) has $O(|H| \cdot |E|)$ variables and $O(|H| \cdot |E|)$ constraints, assuming $|V| = O(|E|)$. Using *Karmarkar's algorithm* [32], this LP can be solved in $O(|H|^{7.5} \cdot |E|^{7.5})$ time. We note that the link-level to path-level conversion (line 3) can be done in $O(|H| \cdot |E|^2)$, where each run of Edmonds-Karp takes $O(|E|^2)$, as finding a path by breadth first search takes $O(|E|)$ and each path saturates at least one link. The overall complexity of the randomized rounding algorithm is therefore $O(|H|^{7.5} \cdot |E|^{7.5})$.

C. Minimax Path Algorithm

As is shown later (Section VI), solving the LP relaxation of (7) described in Section IV-B still requires a computation time that is significantly larger than the time to deploy the rules for reasonably large networks. For a significant reduction of the complexity, we propose a *minimax path algorithm* as follows.

The basic idea is that instead of jointly routing all the flows, we consider one flow at a time as in the baseline solution (Section IV-A). The difference from the baseline is that for each flow h, we select the path with the minimum update time by solving the optimization (7) with $H = \{h\}$ and c_{ij} being the current residual capacity on link $(i, j) \in E$. We have two key observations:

1) if we consider flows in an ordered sequence where existing flows come before new ones, the optimal solution to (7) is not to reroute any existing flow. This occurs when the optimization problem (7) is used to solve iterations of a greedy approach, such as minimax path, which processes flows sequentially, without a global view.

2) for a new flow, the objective value of (7) equals $\max_{i \in p_h} w_i$ for the selected path p_h, i.e., the maximum per-rule update time among the switches traversed by this flow.

Therefore, the optimal solution to (7) when considering one flow at a time (with existing flows considered first) is to keep each existing flow as is, and route each new flow on the path that minimizes the maximum per-rule update time among all the paths with sufficient residual capacity. Note that this solution is always non-disruptive.

The above observations lead to a path selection algorithm similar to Algorithm 1, except that line 3 initializes the residual capacity to be the capacity left by the existing flows, line 4 only iterates among the new flows, and line 6 is replaced by

$$p_h = \textit{minimax\_path}(\mathcal{G}_h,\ s_h,\ t_h,\ \varphi_h), \tag{11}$$

which selects the path from s_h to t_h in $\mathcal{G}_h$ that minimizes the maximum per-rule update time among the traversed nodes and is limited to a number of φ_h hops on a path. Similar to the subroutine *shortest_path* in Algorithm 1, subroutine (11) may not be able to find any path for a flow, in which case the flow will be dropped.

The minimax path problem can be solved by modifying any shortest path algorithm to change the calculation of path length from the sum of link/node weights to the maximum of link/node weights [33].

Complexity: Using the minimax version of the Dijkstra's algorithm, *minimax_path*($\mathcal{G}_h,\ s_h,\ t_h,\ \varphi_h$) can be computed in $O(|E| + |V| \log |V|)$ time. Thus, the overall minimax path algorithm has a complexity of $O(|H_n| \cdot (|E| + |V| \log |V|))$.

V. Restricted Path Selection Algorithms

In large networks, computing paths from scratch is time-consuming and can become the bottleneck of network updates (see Section VI). To address this challenge, we propose to reduce the solution space by restricting each flow to a set of *candidate paths*, inspiring a two-step solution: (1) step one computes a set of candidate paths offline for each switch pair, and (2) step two selects one of the candidate paths online for each flow.

A. Offline Path Computation

The first step aims at precomputing a set of paths between each pair of nodes with minimum overlap, such that they will provide enough diversity to route the flows to avoid excessive updates at any single switch. We precompute χ paths per switch pair, where χ is a design parameter that controls the trade-off between complexity (i.e., time to compute the rules) and optimality (i.e., time to deploy the rules). Specifically, given a parameter χ and a switch source-destination pair (s, t), we want to find χ paths from s to t such that the maximum number of paths traversing the same switch is minimized.

This problem is structurally similar to the Non-Disruptive MinUpdateTime problem in (10): if $w_i \equiv 1$ for all $i \in V$, the objective of (10) is precisely to minimize the maximum number of flows in H_n that traverse the same switch. The difference is that the link capacity constraint (10d) no longer applies since we are precomputing paths instead of routing flows. Cycle avoidance constraints (10f) are also removed to reduce the complexity without affecting the optimal solution, as we can remove cycles in a given solution without increasing the objective value. Thus, the problem of computing candidate paths, referred to as the *CandidatePath problem*, is a special case of (10) for $H_n = \{1, \ldots, \chi\}$, hereby denoted with $[\chi]$, $(s_h,\ t_h) \equiv (s,\ t)$ for all $h \in [\chi]$, $w_i \equiv 1$ for all $i \in V$, and $\widetilde{c}_{ij} = \infty$ for all $(i, j) \in E$, i.e.,

$$\min R \tag{12a}$$

$$\text{s.t.} \sum_{h\in[\chi]} \sum_{j:(i,j)\in E} x^h_{ij} \le R, \qquad \forall i \in V \setminus \{s, t\}, \tag{12b}$$

$$\sum_{j\in V} x^h_{ij} = \sum_{k\in V} x^h_{ki} + b_i, \qquad \forall h \in [\chi], i \in V, \tag{12c}$$

$$\sum_{i,j\in E} x^h_{ij} \le \varphi_h, \qquad \forall h \in [\chi], \tag{12d}$$

$$x^h_{ij} \in \{0,\ 1\}, \qquad \forall h \in [\chi], (i, j) \in E, \tag{12e}$$

where $b_i = 1$ if $i = s, b_i = -1$ if $i = t,\ b_i = 0$ otherwise. The maximum length of a candidate path, φ_h, is specified by constraint (12d). Note that we exclude s and t in (12b) to avoid the trivial solution of χ identical paths, as s and t have to be on every path. After solving (12), each set of links $\{(i, j) \in E :\ x^h_{ij} = 1\}$ $(\forall h \in [\chi])$ forms a candidate path for the switch pair $(s,\ t)$. The setting of χ depends on the network topology and on the possibilities to compute paths with minimum overlap. If there are fewer than χ non-overlapping paths, the optimization model (12) returns less than χ paths. This affects the performance of our restricted paths approach since there are fewer options to distribute the load of flow rule updates.

Complexity: In contrast to the hardness of (10), its special case (12) can be solved in polynomial time as all the flows $h \in [\chi]$ have the same source and the same destination. Given R, we can check whether there exist χ paths from s to t that do not traverse a node more than R times by computing the maximum flow between s to t under node capacity constraint R, as there exist such χ paths if and only if the maximum flow is at least χ. Based on the result of the check, we can use binary search to find the minimum R for which the check is positive. As each check can be performed in $O(|E|\chi)$ time by the *Ford-Fulkerson algorithm* [31], and the binary search takes at most $O(\log \chi)$ steps (as the optimal R is bounded by χ), the overall complexity of solving (12) is $O(|E|\chi \log \chi)$.

B. Online Path Selection

The second step aims at selecting one path per flow from the candidate paths to minimize the network update time under link capacity constraints.

Formulating this problem as an optimization requires rewriting the optimization (7) in terms of a new decision variable $z^h_p \in \{0,\ 1\}$, which indicates whether path p is selected to carry flow h. Specifically, let P_h denote the set of candidate paths for flow h (which is precomputed for any switch pair $(s_h,\ t_h)$), and $P^h_{ij} \triangleq \{p \in P_h :\ (i, j) \in p\}$ the subset of candidate paths traversing link $(i, j) \in E$. It is easy to see that flow h traverses link (i, j) if and only if it is routed on one of the paths in P^h_{ij}, i.e., $x^h_{ij} = \sum_{p\in P^h_{ij}} z^h_p$. Moreover, since every candidate path in P_h is guaranteed to connect s_h and t_h, we can satisfy flow conservation by ensuring that $\sum_{p\in P_h} z^h_p = 1$. We can rewrite the MinUpdateTime problem in (7) as follows:

$$\min R,\ \text{s.t.} \tag{13a}$$

$$w_i \sum_{h\in H} \sum_{j:(i,j)\in E} \Big(a^h_{ij}(\sum_{p\in P^h_{ij}} z^h_p) + k^h_{ij}\Big) \le R, \quad \forall i \in V, \tag{13b}$$

$$\sum_{p\in P_h} z^h_p = 1, \qquad \forall h \in H, \tag{13c}$$

$$\sum_{h\in H} d_h \cdot (\sum_{p\in P^h_{ij}} z^h_p) \le c_{ij}, \qquad \forall (i, j) \in E, \tag{13d}$$

$$z^h_p \in \{0,\ 1\}, \qquad \forall h \in H, p \in P_h, \tag{13e}$$

where a^h_{ij} and k^h_{ij} in (13b) are constants as defined in (7). It is easy to see that if P_h contains all possible paths from s_h to t_h, then (13) is equivalent to (7), which is hard to solve. The key difference here is that P_h is limited to a subset of all possible paths (i.e., the candidate paths), thus allowing faster computation. Thus, we refer to (13) as the *Restricted MinUpdateTime problem*.

If the flow configuration has to be non-disruptive (as in (10)), then constraint (13b) will be reduced to

$$w_i \sum_{h\in H} \sum_{p\in P^h_i} z^h_p \le R, \quad \forall i \in V, \tag{14}$$

where $P^h_i \triangleq \{p \in P_h :\ i \in p\}$ is the set of candidate paths for flow h that traverses switch i.

Feasibility: An issue with limiting P_h to the precomputed paths is that a set of flows that are originally feasible may become infeasible under the constraint, i.e., there is no feasible

solution to (13). This issue can be addressed by applying admission control before routing flows by (13), which selects a subset of flows $H' \subseteq H$ to ensure feasibility while maximizing certain measure of profit. For example, to minimize the total packet loss, we can define the profit for each flow h to be its demand d_h, and to minimize the number of dropped flows, we can define a uniform profit for each flow. Both of these problems are special cases of the *unsplittable flow problem (UFP)*, which is NP-hard but approximable [30]. Our focus is not on UFP; instead, we can apply any existing solution to UFP and feed the set of admitted flows to (13).

Hardness: The simplest non-trivial case of (13) is $\chi = 2$. However, even this case is NP-hard as shown below.

Corollary 3. *The Restricted MinUpdateTime problem (13) is NP-hard for any $\chi \geq 2$.*

It suffices to show that (13) is NP-hard for $\chi = 2$ in the special case of $H = H_n$. This is directly implied by the proof of Theorem 2, which reduces the knapsack problem by constructing an instance of (13) where each flow has at most two different paths to choose from. The hardness of finding the optimal solution to the Restricted MinUpdateTime problem motivates the search for efficient alternatives. In theory, any unrestricted path selection algorithm can be modified to select from the candidate paths. Below we modify the algorithms presented in Section IV.

1) Restricted Shortest Path Algorithm: The shortest path algorithm in Algorithm 1 can be applied to the Restricted MinUpdateTime problem by limiting its search of the shortest path to the candidate paths for each flow. Specifically, line 5 is replaced by finding the subset of candidate paths with sufficient residual capacity:

$$\widetilde{P}_h = \{p \in P_h : \min_{(i,j)\in p} \widetilde{c}_{ij} \geq d_h\}, \tag{15}$$

and line 6 is replaced by selecting the shortest path in $\widetilde{P}_h$:

$$p_h = \arg\min_{p\in\widetilde{P}_h} |p|, \tag{16}$$

where $|p|$ is the hop count of path p. If no candidate path has sufficient residual capacity to carry this flow (i.e., $\widetilde{P}_h = \emptyset$), then the flow will be dropped.

Complexity: Both finding valid candidate paths by (15) and finding the shortest path by (16) takes $O(\chi \cdot |E|)$ time, and thus the restricted shortest path algorithm has complexity $O(|H| \cdot \chi \cdot |E|)$. If the design parameter χ is set to a constant (i.e., $\chi = O(1)$), the complexity becomes $O(|H| \cdot |E|)$. This is the same as the unrestricted shortest path algorithm (Section IV-A).

2) Restricted Randomized Rounding Algorithm: We can apply the idea of randomized rounding to the MILP formulation in (13). Given a fractional solution $z_p^h \in [0, 1]$ to the LP relaxation of (13), we can round it to an integral solution by following the same steps as in Algorithm 2.

Complexity: By similar analysis as in Section IV-B, we see that solving the LP relaxation takes $O(\chi^{5.5}|H|^{5.5}(|E| + \chi|H|)^2)$ time as there are $O(\chi|H|)$ decision variables and $O(|E| + \chi|H|)$ constraints, while the rounding takes $O(|H| \cdot |E|)$ time. Thus, using randomized rounding to solve (13) has complexity $O(\chi^{5.5}|H|^{5.5}(|E| + \chi|H|)^2)$. If $\chi = O(1)$, then this complexity becomes $\max(O(|H|^{5.5}|E|^2),\ O(|H|^{7.5}))$, which is reduced from the $O(|H|^{7.5}|E|^{7.5})$ complexity of applying randomized rounding to the unrestricted problem (7).

3) Restricted Minimax Path Algorithm: To adapt the minimax path algorithm (Section IV-C), we again consider one flow at a time with existing flows considered first, except that now our goal is to solve (13) for $H = \{h\}$, where h is the flow under consideration. Similar to the observations made in Section IV-C, we observe that the optimal solution to (13) for an existing flow is not to reroute it, and the optimal solution for a new flow is to route it on the path that minimizes the maximum per-rule update time among the candidate paths for this flow. These observations imply an algorithm similar to the minimax path algorithm, except that the subroutine *minimax_path* in (11), which selects the path that minimizes the maximum per-rule update time among all possible paths, is replaced by selecting from the candidate paths with sufficient residual capacity:

$$p_h = \arg\min_{p\in\widetilde{P}_h} \max_{i\in p} w_i, \tag{17}$$

where $\widetilde{P}_h$ is the set of candidate paths with sufficient residual capacity for flow h as defined in (15). Again, if no candidate path has sufficient residual capacity (i.e., $\widetilde{P}_h = \emptyset$), then the flow will be dropped.

Complexity: For each flow $h \in H_n$, finding candidate paths by (15) takes $O(\chi \cdot |E|)$ time, and selecting the minimax path by (17) takes $O(\chi \cdot |V|)$ time. Thus, the restricted minimax path algorithm has complexity $O(|H_n| \cdot \chi \cdot |E|) = O(|H_n| \cdot |E|)$, assuming $|V| = O(|E|)$ and $\chi = O(1)$. This is essentially the same complexity of the unrestricted minimax path algorithm.

TABLE II: Complexity comparison (assuming $\chi = O(1)$)

Unrestricted Path Selection		
Shortest Path	Minimax Path	Randomized Rounding
$O(\|H\|\|E\|)$	$O(\|H_n\|(\|E\| + \|V\| \log \|V\|))$	$O(\|H\|^{7.5}\|E\|^{7.5})$

Restricted Path Selection		
Shortest Path	Minimax Path	Randomized Rounding
$O(\|H\|\|E\|)$	$O(\|H_n\|\|E\|)$	$O(\|H\|^{5.5}(\|E\| + \|H\|)^2)$

C. Complexity Comparison

We summarize the complexities of the proposed algorithms in Table II, the number of candidate paths per switch source-destination pair is assumed as a constant, i.e., $\chi = O(1)$.

From the comparison, we see that the complexities of the shortest path algorithm and the minimax path algorithm grow linearly with the number of flows, while the complexity of the randomized rounding algorithm grows as a high-order polynomial. This is because the first two algorithms process the flows sequentially, while the randomized rounding algorithm processes the flows jointly.

VI. Evaluation

A. Test Environment

To simulate realistic SDNs, we use the Rocketfuel topologies [34], which are used in many recent publications on SDN such as [5], [6], [35], [36], [37], [38] as well as *fat-tree* topologies [39] that are typically used in data-center networks. We present the results based on the Rocketfuel topology AS1239 with 1944 links and 315 nodes, for a capacity of

TABLE III: Average experimental results

	Non-Disr. Rocketfuel			Disruptive Rocketfuel			Non-Disr. Fat Tree			Disruptive Fat Tree		
	T_{total}	T_{deploy}	$\%\ loss$	T_{total}	T_{deploy}	$\%\ loss$	T_{total}	T_{deploy}	$\%\ loss$	T_{total}	T_{deploy}	$\%\ loss$
Restricted Minimax	1.73s	1.73s	0.08%	1.72s	1.72s	0.13%	3.52s	3.52s	0.17%	3.41s	3.41s	0.93%
Unrestricted Minimax	1.59s	1.47s	0.0%	1.55s	1.46s	0.0%	1.77s	1.64s	0.0%	1.76s	1.66s	0.0%
Restricted RR	1.23s	1.22s	0.02%	2.24s	2.18s	0.06%	3.26s	3.26s	0.01%	4.92s	4.88s	0.0%
Unrestricted RR	16.52s	1.18s	0.64%	908.21s	4.8s	0.06%	2.43s	1.82s	0.21%	59.06s	6.14s	0.0%
Restricted Shortest	1.68s	1.68s	0.45%	1.65s	1.64s	0.54%	3.86s	3.86s	0.45%	3.84s	3.83s	0.44%
***Baseline** (Unrestricted Shortest)*	2.23s	2.06s	0.0%	2.68s	2.06s	0.0%	3.67s	3.48s	0.0%	4.24s	3.46s	0.0%

12 for every link and results from an 8-ary fat-tree with 1280 links and 256 server nodes and a link capacity of 25.

To evaluate dynamic flow demands, 40 flows with a value of 1 arrive in our network every computation round, while 10 previous flows depart. The source and destination nodes of a flow are selected based on a random distribution. A number of $\chi = 25$ candidate paths for every node (switch) pair, is used for the restricted algorithms. To emphasize the evaluation of our algorithms and models on network configuration time, we set a loose bound on the number of maximum hops per path for a flow h of $\varphi_h = 15$.

Considering the measurements of per-rule deployment times discussed in Section III-C, we consider an average duration for insertion/modification of SDN rules of *250ms* per-rule in our experiments. We also evaluate our framework by simulating switches optimized for fast rule deployment, which require hardware extension as discussed by Wen et al. [4] where we assume a per-rule update time of *15ms*.

We run our algorithms in Python and use the Gurobi solver (version 7.0.2) on a computer with Ubuntu 14.04 64bit, an Intel i7 8-core @ 3.60GHz processor, and 16GB memory.

B. Evaluation Metrics

In each round of simulation, we evaluate the flow configuration algorithms in terms of how promptly they update the network and how well they satisfy the flow demands. The promptness is measured by both the time to compute the set of new rules, denoted by $T_{compute}$, and the time to deploy these rules at the switches, denoted by T_{deploy}. Based on these metrics, we evaluate the total network configuration time $T_{total} = T_{compute} + T_{deploy}$. In our simulations, $T_{compute}$ is directly measured, but T_{deploy} is calculated based on the number of updated rules per switch as in Equation (3). We ignore the queueing delay T_B and the southbound communication delay T_S as they are negligible compared to $T_{compute}$ and T_{deploy} in our evaluations. Moreover, since the algorithms also differ in how much flow demand they can satisfy, we measure the packet loss in percentage of the overall demand. To compute the percentage of packet loss we calculate the ratio between the total demand of the dropped flows and the total amount of demand from all flows, in number of packets. Here, the parameter d_h can be seen as the amount of packets transferred by a flow h. We run our algorithms in two different modes: 1) *disruptive*, where rules of existing flows can be updated in addition to accommodating new flows and 2) *non-disruptive*, where only rules for newly arriving flows are added. In the presented evaluations we use the unrestricted shortest path algorithm as a comparison baseline for our proposed algorithms and models.

C. Evaluation results for varying flow demand

In the Rocketfuel topologies, the best performing algorithm in the non-disruptive case shown in Figure 3 is restricted random rounding; it shows the lowest total configuration time and has a negligible packet loss of 0.02%. The packet loss in the random rounding based algorithms is due to the rounding step which may cause congestion.

In Figure 4 we consider the disruptive case. Unlike the minimax algorithm, which is always non-disruptive, the shortest path and the randomized rounding algorithms show worse computation time in the disruptive case since more flows have to be processed in each round. For the trade-off of longer computation time, the introduced algorithms cause lower packet loss by adjusting existing flows. The unrestricted minimax algorithm shows the best performance overall in the disruptive case.

In an 8-ary fat-tree topology, which is typical for a datacenter, we can observe that our unrestricted minimax as well as the unrestricted randomized rounding algorithms show the best performance in the non-disruptive case, shown in Figure 5. We can observe that in the disruptive case, shown in Figure 6, the best performance is shown by the unrestricted minimax algorithm, similar to the Rocketfuel topology.

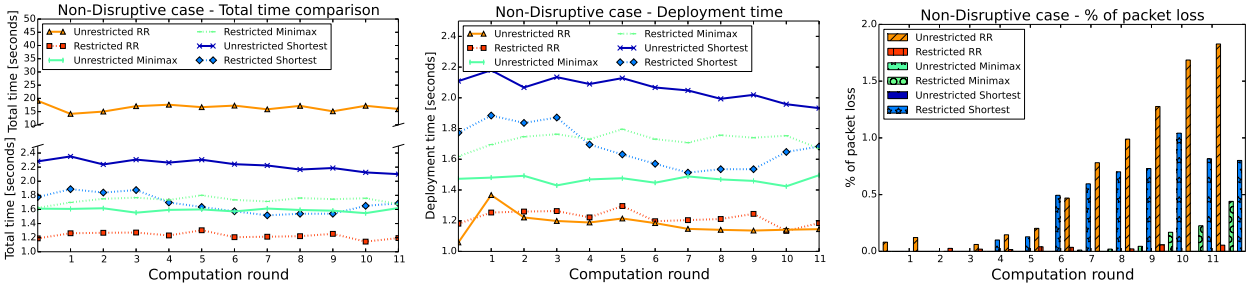
The deployment time for the unrestricted random rounding algorithm is increasing in the disruptive case in both topologies. One reason for this behavior is the random rounding step, as discussed in Section IV-B, which may cause additional flow rule updates on a switch due to its random path selection.

In Table III we summarize the results averaged over all the computation rounds. Our best-performing algorithms outperform the baseline (unrestricted) shortest path algorithm in minimizing the update time, while having similar performance in packet loss. On the Rocketfuel topologies, our algorithms reduce the network update time by $40\% - 45\%$ on average and by $50\% - 55\%$ on fat-tree topologies.

For the same experiment, computing the network configuration times on switches using a hardware extension to reduce the per-rule deployment time to $15ms$, our framework is able to reduce the network configuration time to $80ms$ compared to $300ms$ of the baseline algorithm in the non-disruptive case. In the disruptive case, our framework is able to reduce the network configuration time from $750ms$, required by the baseline algorithm, to $170ms$. This shows that our solution still achieves significant improvement if switches with small per-rule deployment times are used.

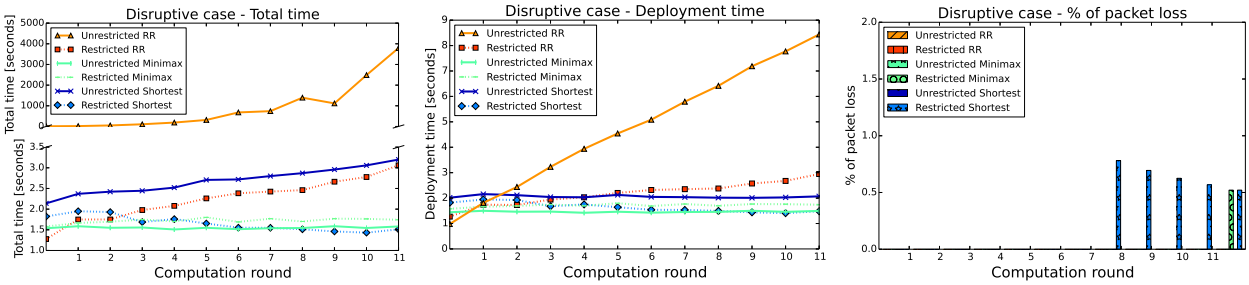
D. Evaluation results for failure recovery

Recovery from failures of links or nodes is critical to maintain connectivity and availability of a network. To demonstrate that our algorithms are able to minimize the delay to reestablish disrupted flows, we consider scenarios where a certain fraction of links have failed. Such scenarios are typical for environments where networked devices are deployed in hostile environments where parts of the network can be compromised (e.g. due to an attack on the network infrastructure).



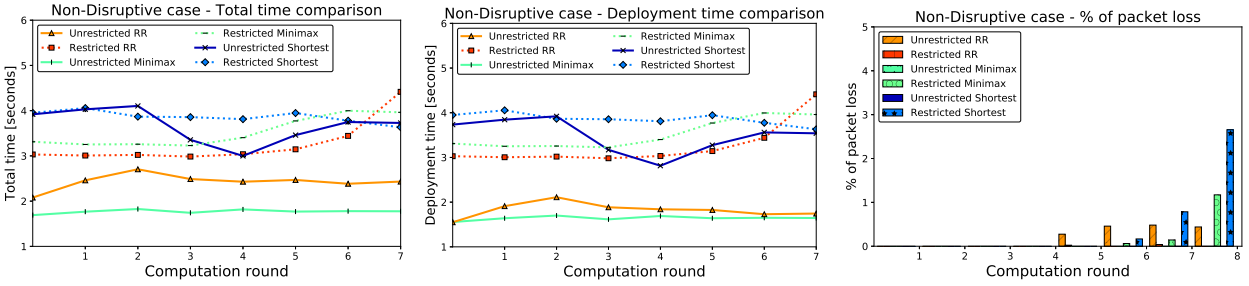
(a) Total time (b) Deployment time (c) Packet loss

Fig. 3: Evaluation results non-disruptive case on AS1239 - 40 flows arrive in each round, 10 previous flows depart



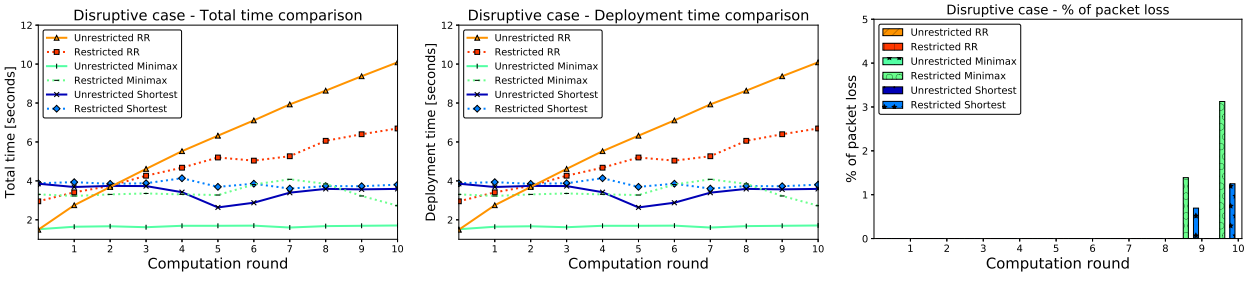
(a) Total time (b) Deployment time (c) Packet loss

Fig. 4: Evaluation results disruptive case on AS1239 - 40 flows arrive in each round, 10 previous flows depart



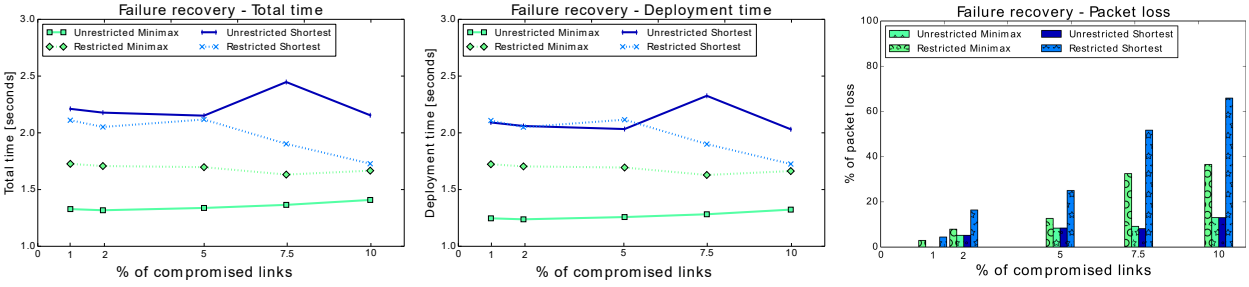
(a) Total time (b) Deployment time (c) Packet loss

Fig. 5: Evaluation results non-disruptive case on 8-ary fat-tree - 40 flows arrive in each round, 10 previous flows depart



(a) Total time (b) Deployment time (c) Packet loss

Fig. 6: Evaluation results disruptive case on 8-ary fat-tree - 40 flows arrive in each round, 10 previous flows depart



(a) Total time (b) Deployment time (c) Packet loss

Fig. 7: Evaluation results failure recovery on AS1239 - 1%-10% of links are failing, 200 flows traversing network

To analyze the performance of our proposed algorithms in such a scenario, we randomly select a certain fraction of links in the Rocketfuel AS1239 topology to be unavailable, and test the performance of our algorithms under such conditions.

In the described scenario, where a certain fraction of links have failed, parts of the network may be disconnected. In case the source and destination nodes of a specific packet flow are located in disconnected parts of the network, it is not possible to determine a path between those endpoints. Since the *Unrestricted Random Rounding* and *Restricted Random Rounding* models are based on solving linear programs, as introduced in (7) and (13), for a batch of flows, solving these models becomes infeasible if no path for a flow between two endpoints exists. Therefore we do not consider these models in the failure recovery evaluation.

In a scenario under the presence of adversaries which cause a certain fraction of links to fail, existing flows that are disrupted due to a compromised link have to be rerouted whenever possible to reestablish the connection. The main goal of our framework to tackle compromised links, is to reroute affected flows whenever possible to reestablish the network services as fast as possible to minimize disruption. To evaluate our algorithms to recover flows, in the described scenario we compute a network configuration for the affected flows in addition to newly arriving flows, which is similar to a non-disruptive case as evaluated in Section VI.

In Figure 7 we show the (a) total network configuration time, (b) deployment time and (c) the amount of packet loss. In the presented results we consider fractions of *1% - 10%* of compromised links (randomly selected), which cause a disruption of $5\% - 15\%$ of the flows in the network. For the maximum path length, we set a loose bound of $\varphi_h = 15$ for this experiment to focus on the network update time. To evaluate the performance of our algorithms to recover flows from failed links, we perform 1500 simulations where we fail a random set of links in each simulation. The reported results in this paper show the average over all the performed simulations.

The deployment time, as shown in Figure 7(b), reflects the objective function of our evaluated algorithms. Our proposed unrestricted minimax algorithm shows the best performance compared to the baseline algorithms for all evaluated fractions of compromised links.

Both baseline algorithms do not perform well in scenarios with failing links. The unrestricted shortest path algorithm shows an increasing configuration time since fewer shortest paths become available with an increasing number of failing links. This causes congestion on a number of paths and results in an increased deployment time. Once the network gets close to its saturation, the deployment time drops since only a small number of new flows can be accommodated and fewer disrupted flows recovered while the packet loss increases as shown in Figures 7(a), 7(b) and 7(c). The restricted shortest path algorithm shows a decreasing network configuration time caused by significant packet losses since only a restricted number of candidate paths is considered which does not make it a good approach for a scenario with flow disruption.

We also evaluate the restricted versions of the minimax and shortest path algorithms, introduced in Section V, as shown in Figure 7. The restricted algorithms perform well in terms of total and deployment time, but do not show a good performance in terms of packet loss, as shown in the third plot of Figure 7.

TABLE IV: Average experimental results

	Failure recovery		
	T_{total}	T_{deploy}	$\%\ loss$
Restricted Minimax	1.66s	1.66s	17.07%
Unrestricted Minimax	1.33s	1.25s	6.94%
Restricted Shortest	1.98s	1.98s	29.18%
***Baseline** (Unrestricted Shortest)*	2.24s	2.12s	6.33%

In Table IV we provide an overview of the average results of the evaluated algorithms. Considering the evaluated metrics total time and deployment time, our unrestricted minimax algorithm shows the best performance in networked environments with failing links. As shown in Table IV the total network reconfiguration time to reroute disrupted flows is less than $2s$, a value which is often used as the initial TCP timeout on RedHat Linux hosts [40].

Our unrestricted minimax algorithm is able to reduce the network configuration time to reestablish disrupted flows by 40% on average, while having a similar performance in terms of packet loss (with a difference within 1%), compared to the unrestricted shortest path baseline algorithm.

TABLE V: Average path lengths in terms of hops

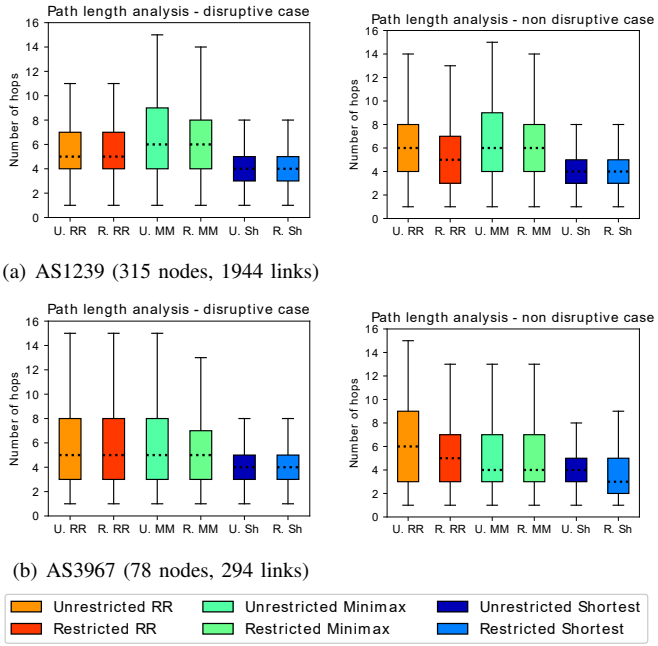
	Rocketfuel		Fat-Tree	
	$Non.D.$	$Disr.$	$Non.D.$	$Disr.$
Restricted Minimax	5	6	5	6
Unrestricted Minimax	5	6	8	6
Restricted RR	5	5	6	6
Unrestricted RR	6	5	7	6
Restricted Shortest	4	4	5	6
***Baseline** (Unrestricted Shortest)*	4	4	6	6

E. Evaluation of path lengths

As a trade-off for shorter network configuration time, our algorithms may compute longer paths compared to the shortest-path baseline algorithm. This effect is smaller on structured topologies where path lengths tend to be uniform, such as in the case of the fat-tree, while it is more evident for the topologies of the Rocketfuel data-set, as we show in Table V. To evaluate the impact of our algorithms on the lengths of computed paths in detail, in Figures 8 we show a study on the Rocketfuel AS1239 (a) topology, which is the largest topology (315 nodes, 1944 links) in this data set and the AS3967 (b) topology which is the smallest Rocketfuel topology (78 nodes, 294 links). As in the experiments above, we set a bound of $\varphi_h = 15$ for the maximum path length.

As expected, the shortest path based algorithms show the smallest path lengths, since their main objective is to minimize the number of hops between a source and a destination node. Our proposed minimax and randomized rounding algorithms follow a different objective and do not minimize the path length between network endpoints. As a trade-off for faster network configuration time they produce longer paths. On a larger topology this effect is slightly stronger on average due to the higher path diversity available between different nodes. Our algorithms compute an average path length of 5.6 hops as can be observed in Figure 8(a). The lower and upper quartiles of path lengths for the shortest path based algorithms

range from three to five hops, while the minimax and random rounding algorithms show a range from three to nine hops considering the lower and upper quartiles. The effect of longer paths is slightly lower on a smaller topology as shown in Figure 8(b) due to the reduced path diversity. Here the average path length computed by our algorithms is 4.6 hops.



(a) AS1239 (315 nodes, 1944 links)

(b) AS3967 (78 nodes, 294 links)

Fig. 8: Path length analysis

The analysis of resulting path lengths shows that longer paths are a trade-off in return for faster network configuration time. In scenarios such as cloud- or data-center networks with a very short propagation delay between network endpoints, a faster network configuration time is beneficial. Considering a *1000Mbps* SDN-switch we compute a per-hop delay of approximately *0.15ms* for a maximum packet size of *64kB*, similar as discussed by He et al. [7]. Further, according to studies presented in [41], [42] over 80% of flows in data-centers are short term flows and therefore benefit from a faster network configuration time.

Other examples in SDN where the path length is a secondary objective are deployments where waypoint enforcement through SDN switches is ensured as discussed by Levin et al. [43]. ISP deployments, where longer paths may impact the user experience or increase the routing cost, optimizing for shorter paths will be the main objective.

The effect of longer paths on the communication delay between network endpoints highly depends on factors such as queuing delay, link bandwidth or current network saturation. To provide a mechanism to control the issue of long paths we add a threshold for the number of hops (φ_h) which can be set to limit the upper bound on path length as we discuss in our problem formulation in Section III. Such a threshold can also be implemented in the heuristic algorithms.

VII. Discussion

A. Policy-preserving Updates

Some networks require the deployment of updated flow rules to preserve network policies as discussed in [2], [5], [6], [9], [12]. Recently, the authors of [5], [6] propose a method to preserve policies during updates by dividing the updated flow rules into a sequence of groups $(G_1, ..., G_n)$, where each group G_q consists of a set of flow rules that can be deployed in an arbitrary order. However, rules in different groups must be deployed sequentially so that all the rules in G_q are deployed before rules in G_{q+1} start to be deployed. To model the network update time using this deployment method, we modify the formula in (3) into

$$\tau(F) = \sum_{q=1}^{n} \max_{i \in V} w_i \cdot |G_{q,i}|, \tag{18}$$

where $F = \bigcup_{q=1}^{n} G_q$ is the overall set of updated rules, and $G_{q,i}$ is the set of rules in G_q for switch i. Accordingly, we can extend the MinUpdateTime problem (7) to accommodate policy-preserving constraints by computing the network configuration time using (18). The challenge is to explicitly express these constraints in the decision variable x_{ij}^h and solve the optimization efficiently, which is left to future work.

B. Future Work

We are planning to evaluate our algorithms on a physical SDN network deployed on a campus or corporate environment. Currently we do not have access to such a network, therefore we have to rely on simulations. In such an environment we expect changes in packet loss since this highly depends on the network load, transmission protocols (e.g. TCP which will retransmit lost packets) and the overall condition of the network. We do expect a significant improvement of the network configuration time with the usage of our algorithms.

VIII. Conclusion

In this work, we study the problem of minimizing the network configuration time in an SDN in response to changing demands, by computing a flow configuration that minimizes the worst case update time across switches. We empirically show that our heuristics can reduce the update time of a shortest-path baseline algorithm up to 55% on average while having little packet loss. Further, we demonstrate that our proposed algorithms are able to reestablish disrupted flows in scenarios with failed links 40% faster on average compared to shortest-path based algorithms.

Acknowledgment

This work was sponsored by the U.S. Army Research Laboratory Cyber Security Collaborative Research Alliance under Cooperative Agreement W911NF-13-2-0045. The views and conclusions contained in this document are those of the authors, and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation hereon.

References

[1] A. Destounis *et al.*, "Minimum cost sdn routing with reconfiguration frequency constraints," *IEEE/ACM Transactions on Networking*, 2018.

[2] S. Brandt *et al.*, "On consistent migration of flows in SDNs," *IEEE INFOCOM*, 2016.

[3] T. Mizrahi *et al.*, "Software Defined Networks: It's about time," *IEEE INFOCOM*, 2016.
[4] X. Wen *et al.*, "RuleTris: Minimizing rule update latency for TCAM-based SDN switches," *IEEE ICDCS*, 2016.
[5] S. Vissicchio *et al.*, "Flip the (flow) table: Fast lightweight policy-preserving sdn updates," in *INFOCOM 2016*.
[6] S. Vissicchio, L. Cittadini, S. Vissicchio, and L. Cittadini, "Safe, efficient, and robust sdn updates by combining rule replacements and additions," *IEEE/ACM Transactions on Networking (TON)*, 2017.
[7] K. He *et al.*, "Measuring control plane latency in sdn-enabled switches," in *ACM Symposium on SDN Research (SOSR)*, 2015.
[8] N. Katta *et al.*, "Cacheflow: Dependency-aware rule-caching for software-defined networks," in *SOSR*, 2016.
[9] X. Jin *et al.*, "Dynamic scheduling of network updates," in *ACM SIGCOMM Computer Communication Review*, 2014.
[10] A. Tootoonchian *et al.*, "On controller performance in software-defined networks," in *USENIX Hot-ICE*, 2012.
[11] H. Xu *et al.*, "Real-time update with joint optimization of route selection and update scheduling for sdns," in *IEEE ICNP 2016*.
[12] N. P. Katta *et al.*, "Incremental consistent updates," in *HotSDN*, 2013.
[13] M. Reitblatt *et al.*, "Consistent updates for software-defined networks: Change you can believe in!" in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, 2011.
[14] M. Reitblatt, N. Foster *et al.*, "Abstractions for network update," *ACM SIGCOMM Computer Communication Review*, 2012.
[15] A. Ludwig *et al.*, "Transiently secure network updates," in *ACM SIGMETRICS Performance Evaluation Review*, 2016.
[16] K.-T. Foerster *et al.*, "Loop-free route updates for software-defined networks," *IEEE/ACM Transactions on Networking (TON)*, 2018.
[17] J. Zheng *et al.*, "Chronus: Consistent data plane updates in timed sdns," in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*.
[18] J. Zheng, B. Li *et al.*, "Scheduling congestion-free updates of multiple flows with chronicle in timed sdns," *IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 2018.
[19] T. Mizrahi *et al.*, "Timeflip: Using timestamp-based tcam ranges to accurately schedule network updates," *IEEE/ACM Transactions on Networking (ToN)*, 2017.
[20] M. Kuźniar *et al.*, "What you need to know about sdn flow tables," in *PAM*. Springer, 2015.
[21] S. Hong *et al.*, "Poisoning network visibility in software-defined networks: New attacks and countermeasures." in *NDSS*, 2015.
[22] S. Achleitner *et al.*, "Adversarial network forensics in software defined networking," in *ACM Symposium on SDN Research (SOSR)*, 2017.
[23] "Rfc 793 - transmission control protocol," https://tools.ietf.org/html/rfc793
[24] "Queuing in cisco switches," https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/qos_conmgt/configuration/xe-3s/qos-conmgt-xe-3s-book/qos-conmgt-qdepth.html
[25] A. R. Curtis *et al.*, "Devoflow: Scaling flow management for high-performance networks," *ACM Computer Communication Review*, 2011.
[26] R. Narayanan *et al.*, "Macroflows and microflows: Enabling rapid network innovation through a split sdn data plane," in *EWSDN, 2012*.
[27] "Openflow," https://www.opennetworking.org/sdn-resources/openflow.
[28] C. Miller *et al.*, "Integer programming formulations and travelling salesman problems," *Journal of the ACM*, vol. 7, 1960.
[29] S. Even *et al.*, "On the complexity of time table and multi-commodity flow problems," in *Foundations of Computer Science, 1975., 16th Annual Symposium on*. IEEE.
[30] A. Chakrabarti *et al.*, "Approximation algorithms for the unsplittable flow problem," in *APPROX Workshop*, 2002.
[31] T. H. Cormen *et al.*, *Introduction to Algorithms*. MIT Press, 2001.
[32] G. Strang, "Karmarkar's algorithm and its place in applied mathematics," *The Mathematical Intelligencer*, 1987.
[33] M. Pollack, "The maximum capacity through a network," *Operations Research*, 1960.
[34] N. Spring *et al.*, "Measuring isp topologies with rocketfuel," *ACM SIGCOMM Computer Communication Review*, 2002.
[35] J. McCauley *et al.*, "Recursive sdn for carrier networks," *ACM SIGCOMM Computer Communication Review*, 2016.
[36] S. Vissicchio *et al.*, "Safe update of hybrid sdn networks," *IEEE/ACM Transactions on Networking (TON)*, 2017.
[37] A. Panda *et al.*, "Scl: Simplifying distributed sdn control planes." in *NSDI*, 2017.
[38] T. Liu *et al.*, "Usa: Faster update for sdn-based internet of things sensory environments," *Computer Communications*, 2018.
[39] M. Al-Fares *et al.*, "A scalable, commodity data center network architecture," in *ACM SIGCOMM Computer Communication Review*, 2008.
[40] "Tcp timeout value in redhat linux," http://www.justsomestuff.co.uk/wiki/doku.php/linux/syn_tcp_timeout
[41] T. Benson *et al.*, "Network traffic characteristics of data centers in the wild," in *ACM SIGCOMM conference on Internet measurement*, 2010.
[42] S. Kandula *et al.*, "The nature of data center traffic: measurements & analysis," in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, 2009.
[43] D. Levin *et al.*, "Panopticon: Reaping the benefits of incremental sdn deployment in enterprise networks," in *USENIX Annual Technical Conference*, 2014.

Stefan Achleitner received his B.S. and M.S. degrees in Computer Science from Vienna University of Technology, Austria. He earned his Ph.D. in Computer Science and Engineering at the Pennsylvania State University, University Park, PA. His research interests include, software defined networking, network security, virtual machines, Internet of Things and machine learning. He is now a security researcher in the threat and application research group at Palo Alto Networks.

Novella Bartolini (SM'16) graduated with honors and received her PhD in computer engineering from Tor Vergata University of Rome, Italy. She is now associate professor at Sapienza University of Rome. She was visiting professor at Penn State University in 2014-2017. She was program chair and committee member of several international conferences. She has served on the editorial board of Elsevier Computer Networks and ACM/Springer Wireless Networks. Her research interests lie in the area of wireless mobile networks and web based systems.

Ting He (SM'13) received the B.S. degree in computer science from Peking University, China, in 2003 and the Ph.D. degree in electrical and computer engineering from Cornell University, Ithaca, NY, in 2007. Ting is an Associate Professor in the School of Electrical Engineering and Computer Science at Pennsylvania State University, University Park, PA. Between 2007 and 2016, she was a Research Staff Member in the Network Analytics Research Group at the IBM T.J. Watson Research Center, Yorktown Heights, NY. Her work is in the broad areas of network modeling and optimization, statistical inference, and information theory.

Thomas La Porta (F'02) is a Distinguished Professor in the department of computer science and engineering at Penn State University, where he is the Director of the School of Electrical Engineering and Computer Science. Prior to joining Penn State in 2002, he was with Bell Laboratories since 1986 as Director of the Mobile Networking Research Department. He is an IEEE Fellow, Bell Labs Fellow, received the Bell Labs Distinguished Technical Staff Award in 1996, and an Eta Kappa Nu Outstanding Young Electrical Engineer Award in 1996. He was the founding Editor-in-Chief of the IEEE Transactions on Mobile Computing, and served as Editor-in-Chief of IEEE Personal Communications Magazine. His research interests include mobility management, signaling and control for wireless networks, mobile data and sensor systems, and network security.

Diman Zad Tootaghaj is a Postdoc Researcher at Hewlett Packard Labs in Palo Alto, California. She received her Ph.D. degree in the department of computer science and engineering at the Pennsylvania State University. She received B.S. and M.S. degrees in Electrical Engineering from Sharif University of Technology, Iran in 2008 and 2011 and an M.S. degree in Computer Science and Engineering from the Pennsylvania State University in 2015. Her current research interests include computer network, recovery approaches and distributed systems.