Simplifying Software-Defined Network Optimization Using SOL

Victor Heorhiadi and Michael K. Reiter, ***University of North Carolina at Chapel Hill;***
Vyas Sekar, ***Carnegie Mellon University***

https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/heorhiadi

This paper is included in the Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16).

March 16–18, 2016 • Santa Clara, CA, USA

ISBN 978-1-931971-29-4

Open access to the Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16) is sponsored by USENIX.

Simplifying Software-Defined Network Optimization Using SOL

Victor Heorhiadi, Michael K. Reiter, Vyas Sekar
UNC Chapel Hill, UNC Chapel Hill, Carnegie Mellon University

Abstract

Realizing the benefits of SDN for many network management applications (e.g., traffic engineering, service chaining, topology reconfiguration) involves addressing complex optimizations that are central to these problems. Unfortunately, such optimization problems require (a) significant manual effort and expertise to express and (b) non-trivial computation and/or carefully crafted heuristics to solve. Our goal is to simplify the deployment of SDN applications using *general* high-level abstractions for capturing optimization requirements from which we can *efficiently* generate optimal solutions. To this end, we present SOL, a framework that demonstrates that it is possible to simultaneously achieve generality and efficiency. The insight underlying SOL is that many SDN applications can be recast within a unifying *path-based* optimization abstraction. Using this, SOL can efficiently generate near-optimal solutions and device configurations to implement them. We show that SOL provides comparable or better scalability than custom optimization solutions for diverse applications, allows a balancing of optimality and route churn per reconfiguration, and interfaces with modern SDN controllers.

1 Introduction

Software-defined networking (SDN) is an enabler for network management applications that may otherwise be difficult to realize using existing control-plane mechanisms. Recent work has used SDN-based mechanisms to implement network configuration for a range of management tasks: traffic engineering (e.g., [40]), service chaining (e.g., [39]), energy efficiency (e.g., [19]), network functions virtualization (NFV) (e.g., [14]), and cloud offloading (e.g., [44]), among others.

While this body of work has been instrumental in demonstrating the potential benefits of SDN, realizing these benefits requires significant effort. In particular, at the core of many SDN applications are custom optimization problems to tackle various constraints and requirements that arise in practice (§2). For instance, an SDN application might need to account for limited TCAM, link capacities, or middlebox capacities, among other considerations. Developing such formulations involves a non-trivial learning curve, a careful understanding of theoretical and practical issues, and considerable manual effort. Furthermore, when the resulting optimization problems are intractable to solve with state-of-the-art solvers (e.g., `CPLEX` or `Gurobi`), heuristic algorithms must be crafted to ensure that new configurations can be generated on timescales demanded by the application as relevant inputs (e.g., traffic matrix entries) change (e.g., [19, 29]). Furthermore, without a common framework for representing network optimization tasks, it is difficult to reuse key ideas across applications or to combine useful features into a custom new application.

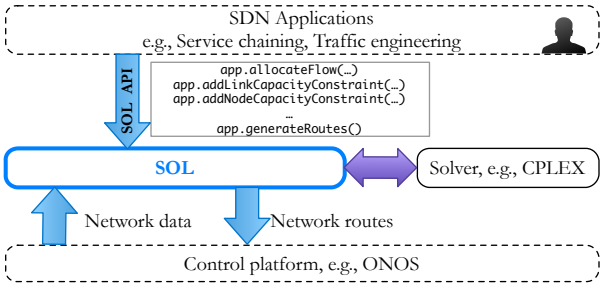


Figure 1: ***Developers use the SOL high-level APIs to specify optimization goals and constraints. SOL generates near-optimal solutions and produces device configurations that are input to the SDN control platform.***

Our goal in this work is to raise the level of abstraction for writing such SDN-based network optimization applications. To this end, we introduce SOL, a framework that enables SDN application developers to express high-level application goals and constraints. Conceptually, SOL is an intermediate layer that sits between the SDN optimization applications and the actual control platform (see Fig. 1). Application developers who want to develop new network optimization capabilities express their requirements using the SOL API. SOL then generates configurations that meet these goals, which can be deployed to SDN control platforms.

There are two natural requirements for such a framework: (1) **generality** to express the requirements for a broad spectrum of SDN applications (e.g., traffic engineering, policy steering, load balancing, and topology management); and (2) **efficiency** to generate (near-) optimal configurations on a timescale that is responsive to application needs. Given the diversity of the application requirements and the trajectory of prior work in developing custom solutions (e.g., [39, 24, 23, 19, 29, 14, 8, 46, 40, 20]), generality and efficiency appear individually difficult, let alone combined. We show that it is indeed possible to achieve both generality and efficiency.

The key insight in SOL to achieve generality is that many network optimization problems can be expressed as *path-based* formulations. Paths are a natural abstrac-

tion for application developers to reason about intended network behaviors and to express policy requirements. For example, we can use paths to specify service chaining requirements (e.g., each path includes a firewall and intrusion-detection system, in that order) or redundancy (e.g., each includes two intrusion-prevention systems, in case one fails open). Finally, it is easy to model device (e.g., TCAM space, middlebox CPU) and link resource consumption based on the volume of traffic flowing through paths that traverse that device or link.

The natural question is whether the generality of path-based formulations precludes efficiency. Indeed, if implemented naively, optimization problems expressed over the paths that traffic might travel will introduce efficiency challenges since the number of paths grows exponentially with the network size. Our key insight is that by combining infrequent, offline preprocessing with simple, online *path-selection algorithms* (e.g., shortest paths or random paths), we can achieve near-optimal solutions in practice for all applications we considered. Moreover, SOL is typically far more efficient than solving the optimization problems originally used to express these applications' requirements.

We have implemented SOL as a Python-based library that interfaces with `ONOS` [5] and `OpenDaylight` [35] (§7). We have also prototyped numerous SDN applications in SOL, including SIMPLE [39], ElasticTree [19], Panopticon [29], and others of our own design (§6 and App. B). SOL is open-source; we have released modules for various applications coded in SOL as well as our `ONOS` extensions [22].

Our evaluations on a range of topologies show that: 1) SOL outperforms several applications' original optimization algorithms by an order of magnitude or more, and is even competitive with their custom heuristics; 2) SOL scales better than other network management tools like Merlin [45]; 3) SOL substantially reduces the effort required (e.g., in terms of lines of code) for implementing new SDN applications by an order of magnitude; and 4) optional SOL extensions can reduce route churn substantially across reconfigurations with modest impact on optimality.

2 Background and Motivation

In this section, we describe representative network applications that could benefit from a framework such as SOL. We highlight the need for careful formulation and algorithm development involved in prior efforts, as well as the diversity of requirements they entail.

2.1 Traffic engineering

Traffic engineering (TE) is a canonical application that was an early driving application for SDN [24, 23]. Fig. 2 shows an example where traffic classes C1 and C2 need to be routed completely while minimizing the load on the most heavily loaded link. A TE application takes as input traffic demands (e.g., the traffic matrix between WAN sites), a specification of the traffic classes and priorities, and the network topology and link capacities. It determines how to route each class to achieve network-wide objectives, e.g., minimizing congestion [11] or weighted max-min fairness [24, 23].

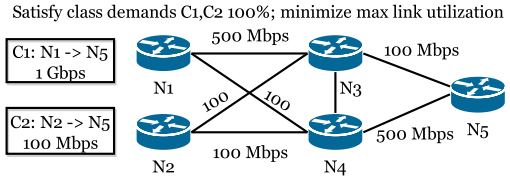


Figure 2: ***Traffic engineering applications***

Challenges: Simple goals like link congestion can be represented and solved via max-flow formulations [1]. However, the expressivity and efficiency quickly breaks down for more complex objectives such as max-min fairness, which multiple research efforts have sought to address [23, 9, 24]. When max-flow like formulations fail, designers invariably revert to "low-level" techniques such as linear programs (LP) or combinatorial algorithms. Neither is ideal—using/tuning LP solvers is painful as they expose a very low-level interface, and combinatorial algorithms require significant theoretical expertise. Finally, translating the algorithm output into actual routing rules requires care to install volume-aware rules to truly reap the benefits of the optimization [47].

2.2 Service chaining

Networks today rely on a wide variety of middleboxes (e.g., IDS, proxy, firewall) for performance, security, and external compliance capabilities (e.g., [44]). The goal of service chaining is to ensure that each class of traffic is routed through the desired sequence of network functions. For example, in Fig. 3, class C1 is required to traverse a firewall and proxy in order. Such policy routing rules must be suitably encoded within the available TCAM on SDN switches [39]. Since middleboxes are often compute-intensive, they can get easily overloaded and thus operators would like to balance the load on these appliances [39, 15]. The key inputs to such applications are the service chaining requirements of different classes, traffic demands, and the available middlebox processing resources. The application then sets up the forwarding rules such that the service chaining requirements are met while respecting the switch TCAM and middlebox capacities. Furthermore, as many of these middleboxes are stateful, these rules must ensure flow affinity.

Challenges: Service chaining introduces more complex requirements when compared to TE applications.

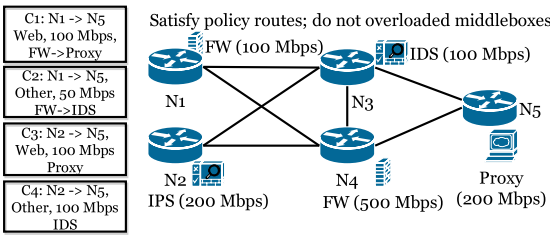


Figure 3: ***Service chaining applications***

First, modeling the consumption of switch TCAM introduces discrete components into the optimization, which impacts scalability [39]. Second, such service processing requirements fall outside the scope of existing network flow abstractions [8]. Third, service chaining highlights the complexity of combining different requirements; e.g., reasoning about the interaction between the load balancing algorithm and the switch TCAM constraints is non-trivial [25]. Existing service chaining efforts developed custom heuristics [7] or new theoretical extensions [8]. Furthermore, as observed previously, ensuring flow affinity can be quite tricky [21, 20].

2.3 Flexible topology management

SDN enables topology modifications that would be difficult to implement with existing control plane capabilities. For instance, ElasticTree [19] and Response [46] use SDN to dynamically switch on/off network links and nodes to make datacenters more energy efficient. In Fig. 4, these applications might shut down node N3 during periods of low utilization, if classes C1 and C2 can be routed via N4 without significantly impacting end-to-end performance. Topology reconfiguration is especially feasible in rich topologies with multiple paths between every source and destination. Such applications take as input the demand matrix (similar to the TE task) and then compute the nodes and links that should be active and traffic-engineered routes to ensure performance SLAs.

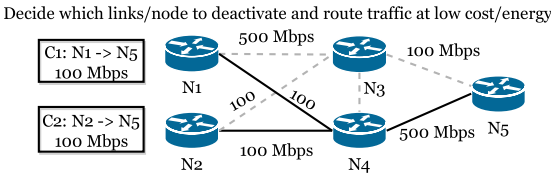


Figure 4: ***Topology reconfiguration applications***

Challenges: The on-off requirement on the switches/links once again introduces discrete constraints, yielding integer-linear optimizations that are theoretically intractable and difficult to express using max-flow like abstractions. Solving such problems requires significant computation even on small topologies and thus forces developers to design new, heuristic solving strategies; e.g., ElasticTree uses a greedy bin-packing algorithm [19].

2.4 Network function virtualization

Prior work has leveraged SDN capabilities to offload or outsource network functions to leverage clusters or clouds [44, 16, 40]. This is especially useful for expensive deep-packet-inspection services [20]. The key decision here is to decide how much of the processing on each path to offload to the remote datacenter — e.g., in Fig. 5, how much of class C1 traffic should be routed to the datacenter between N4 and N5 for IPS processing, versus processing it at N3. Offloading can increase user-perceived latency and impose additional load on network links. Moreover, some active functions (e.g., WAN optimizers or IPS) induce changes to the observed traffic volumes due to their actions. Thus, optimizing such offloading must take into account the congestion that might be introduced, as well as latency impact and any traffic volume changes induced by such outsourced functions. Further generalizations have considered not only offloading middlebox services but also elastically scaling them [36, 14, 34, 6], exacerbating these issues.

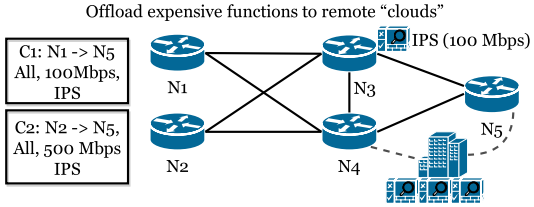


Figure 5: ***Offloading network functions***

Challenges: Such offloading and elastic scaling opportunities introduce new dimensions to optimization that are difficult to capture. For instance, offloading requires rerouting the traffic and thus optimizations must model the impact on link loads, downstream nodes, and TE objectives. If done naively, this can introduce non-linear dependencies since the actions of downstream nodes depend on control decisions made upstream. The active changes to traffic volumes by some functions (e.g., compression for redundancy elimination or drops by IPS) also introduce non-linear dependencies in the optimization. Finally, elastic scaling introduces a discrete aspect to the problem similar to the topology modification application, further decaying the problem's tractability.

2.5 Motivation for SOL

Drawing on the above discussion (and our own experience), we summarize a few key considerations:

- Network applications have diverse and complex optimization requirements; e.g., service chaining requires us to reason about valid paths while topology modification needs to enable/disable nodes.
- Designers of these applications have to spend significant effort in expressing and debugging these prob-

lems using low-level optimization libraries.

- It can take non-trivial expertise to ensure that the problems can be solved fast enough to be relevant for operational timescales, e.g., recomputing TE every few minutes or periodically solving the large integer-linear programs (ILPs) supporting topology reconfiguration (e.g., [19]).

3 SOL Overview

Our overarching vision in developing SOL is to raise the level of abstraction in developing new SDN applications and specifically to eliminate some of the black art in developing SDN-based optimizations, making them more accessible for deployment by network managers. To do so, SOL abstracts away low-level details of optimization solvers and SDN controllers, allowing the developer to focus on the high-level application goals (recall Fig. 1). SOL takes as inputs the network topology, traffic patterns, and optimization requirements in the SOL API. It then translates these into constraints for optimization solvers such as `CPLEX` or `Gurobi`. Finally, SOL interfaces with existing SDN control platforms such as `ONOS` to install the forwarding rules on the SDN switches. SOL does not require modifications to the existing control or data plane components of the network. Our vision for SOL stands in stark contrast to the state of affairs today, in which a developer faces programming a new SDN optimization either directly for a generic and low-level optimization solver such as `CPLEX` or using a heuristic algorithm designed by hand, after which she must translate the decision variables of the optimization to device configurations.

Path abstraction: For SOL to be useful and robust, we need a unifying abstraction that can capture the requirements of diverse classes of SDN network optimization applications described in the previous section. SOL is built using *paths* through a network as a core abstraction for expressing network optimization problems. This is contrary to how many optimizations are formulated in the literature — using a more standard edge-centric approach [1]. In our experience, however, an edge-centric approach forces complexity when presented with additional requirements, especially ones that attempt to capture path properties [29, 19].

In contrast, path-based formulations capture these requirements more naturally. For instance, much of the complexity in modeling service chaining or network function offloading applications from §2 is in capturing the path properties that need to be satisfied. With a path-based abstraction, we can simply define predicates that specify valid paths — e.g., those that include certain waypoints or that avoid a certain node (to anticipate that node's failure). In addition, we can model path-based resource use with ease. For example, usage of TCAM space in a switch corresponds to a traffic-carrying path traversing that switch (and thus a rule to accommodate that path). Without the path abstraction, modeling such constraints is difficult (cf., [39]). Finally, expressing constraints on nodes and edges does not introduce increased difficulty compared to edge-centric approach.

Scalability: In a pure flow-routing scenario, an edge-based formulation admits simple algorithms that guarantee polynomial-time execution. Path-based formulations, on the other hand, are often dismissed because of their inefficient appearance — after all, in the worst case, the number of paths in the network is exponential in the network size — or due to the complexity of algorithms to solve path based formulations (column-generation, decompositions, etc. [1]). However, in many practical scenarios, the number of valid paths (as defined by the application) is likely to be significantly smaller. Furthermore, multipath routing can provide only so much network diversity before its value diminishes [30]. So, the set of paths that need to be considered is not large.

SOL leverages an off-line path generation step to determine valid paths (step 1 of Fig. 6). Since for most applications, the set of valid paths is fairly static and does not need to be recomputed every time the optimization is run, we expect this step is infrequent. Next, SOL *selects* a subset of these paths (step 2) using a selection strategy (see §5) and runs the optimization with only the selected paths as input (step 3), to ensure that the optimization completes quickly. We show in §8 that this strategy still permits inclusion of sufficiently many paths for the optimization to converge to a (near) optimal value. So, while the efficiency of path-based optimization is a valid theoretical concern, in practice we show that there are practical heuristics to address this issue.

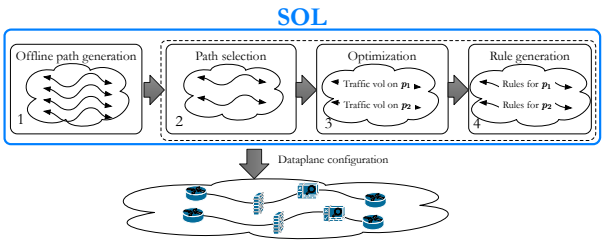


Figure 6: ***SOL architecture, overview of the workflow***

Generating device configurations: SOL translates the decision variables from the SOL optimization to network device configurations to implement appropriate flow routing (step 4 of Fig. 6). The algorithm utilized in SOL to perform this translation is based on that in previous work [47, 20]. However, because the optimization is path-based, the algorithm is more straightforward and requires fewer steps.

`nodes`: Set of all nodes, part of the topology

`links`: Set of all links, part of the topology

`classes`: Set of all traffic classes

`paths`(c): Paths available for class $c \in$ `classes`; output by path-selection stage (§5)

Figure 7: *Network data input*

	Var.	Description
Decision	$x_{c,p}$	Fraction of class-c flows allocated to path $p \in$ `paths`(c); non-integer
	b_p	Is path p used; binary
	b_v	Is node v used; binary
	b_l	Is link l used; binary
	$capvar_v^r$	Capacity allocated for resource r at node v; non-integer
Derived	a_c	Fraction of c's "demand" routed; non-integer
	$load_l^r$	Amount of resource r consumed by flows routed over link l; non-integer
	$load_v^r$	Amount of resource r consumed by flows routed via node v; non-integer

Figure 8: Variables internal to the optimization

4 SOL Detailed Design

In this section, we present the detailed design of SOL. We focus on the high-level API that the SDN application developer would use to express applications via SOL, and the impact of these API calls on the SOL's internal representation of the optimization problem. Note, however, that the developer "thinks" in terms of the high-level API rather than low-level details of dealing with the solver-level variables, how paths are identified, etc.

A developer begins a new optimization in SOL by instantiating an `opt` object via the `getOptimization` function and then building the optimization using *constraint templates*, which we explain below.

4.1 Preliminaries

Data inputs: There are two basic data inputs that the developer needs to provide to any network optimization. First, the network topology is a required input, specified as a graph with `nodes` and `links`. It also contains metadata of node/edge types or properties; e.g., nodes can have designated functions like "switch" or "middlebox". Second, SOL needs a specification of *traffic classes*, where each class c has associated ingress and egress nodes and some expected traffic volume. Each class can (optionally) be associated with a specification of the "processing" required for traffic in this class, e.g., service chaining. Finally, to each traffic class c is associated a set `paths`(c) available to route flows in class c; `paths`(c) is output by a path-selection preprocessing step described in §5.

Internal variables: SOL internally defines a set of variables summarized in Fig. 8. We reiterate that the developer does not need to reason about these variables and uses a high-level mental model as discussed earlier. There are two main kinds of variables:

- *Decision variables* that identify key optimization control decisions. The most fundamental decision variable is $x_{c,p}$, which captures traffic routing decisions and denotes the fraction of flow for a traffic class c that path $p \in$ `paths`(c) carries. This variable is central to various types of resource management applications as we will see later. To capture topological requirements (e.g., §2.3), we introduce three binary decision variables b_p, b_v, and b_l that denote whether each path, node or link (respectively) is enabled ($= 1$) or disabled ($= 0$). The variable $capvar_v^r$ is the SOL-assigned allocation of resource-r to node v.
- *Derived variables* are functions defined over the above decision variables that serve as convenient "shorthands". a_c denotes the total fraction of flow for class c that is carried by all paths. The load variables $load_v^r$ and $load_l^r$ model the consumption of resource r on node v and link l, respectively.

There are low-level API calls [1] that return the names of these internal variables, which can be used to access each one's value in a public map of names to values, if needed.

4.2 Routing requirements

Routing constraints control the allocation of flow in the network. `addAllocateFlowConstraint` creates the necessary structure for routing the traffic through a set of paths for each traffic class. Some network applications try to satisfy as much of their flow demands as possible (e.g., max-flow) while others (e.g., TE) want to "saturate" demands. For example, a developer of a TE application (§2.1) would like to route all traffic though the network, and thus she would add the following high-level routing constraint templates to her empty `opt` object:

```
opt.addAllocateFlowConstraint()
opt.addRouteAllConstraint()
```

In contrast, a simple max-flow would only need `addAllocateFlowConstraint` since there is no requirement on saturating demands in that case.

The `addEnforceSinglePath`(C) constraint forces a single flow-carrying path per class $c \in C$, preventing flow-splitting and multipath routing.

Internals: `addAllocateFlowConstraint` ensures that the total traffic flow across all chosen paths for the class c matches the variable a_c.

$$\forall c \in \texttt{classes}: \quad \sum_{p \in \texttt{paths}(c)} x_{c,p} = a_c$$

[1] We also expose low-level APIs (see Appendix A) for advanced users.

Group	Function	Description

Routing ($C \subseteq$ `classes`)	`addAllocateFlowConstraint`
	`addRouteAllConstraint`
	`addEnforceSinglePath` (C)
		Allocate flow in the network
		Route all traffic demands
		For each $c \in C$, at most one $p \in$ `paths`(c) is enabled.

Capacities	`addLinkCapacityConstraint` (r, *lnCap*, `linkCapFn`) `addNodeCapacityConstraint` (r, *ndCap*, `nodeCapFn`) `addNodeCapacityPerPathConstraint` (r, *ndCap*, `nodeCapFn`) `addCapacityBudgetConstraint` (r, N, *totCap*)	If l is in *lnCap*, then limit utilization of link resource r on link l to *lnCap*$[l]$. If v is in *ndCap*, then limit utilization of node resource r on node v to *ndCap*$[v]$. If v is in *ndCap*, then limit utilization of node resource r on node v by enabled paths to *ndCap*$[v]$. Limit total type-r resources allocated to nodes in $N \subseteq$ `nodes` to *totCap*. Used when SOL is allocating capacities.
Topology control ($C \subseteq$ *classes*)	`addRequireAllNodesConstraint` (C) `addRequireSomeNodesConstraint` (C) `addRequireAllEdgesConstraint` (C) `addPathDisableConstraint` (C) `addBudgetConstraint` (`nodeBudgetFn`, k)	For each $c \in C$ and each $p \in$ `paths`(c), p can be enabled iff all nodes on p are enabled. For each $c \in C$ and each $p \in$ `paths`(c), p can be enabled iff some node on p is enabled. For each $c \in C$ and each $p \in$ `paths`(c), p can be enabled iff all links on p are enabled. For each $c \in C$ and each $p \in$ `paths`(c), p can carry traffic only if it is enabled. Total cost of enabled nodes, as computed using `nodeBudgetFn`, is at most k.
Objective	`setPredefinedObjective` (*name*)	Set one of the predefined functions as the objective (see Fig. 11).

Figure 9: ***Selected constraint template functions for building optimizations; see Fig. 10 for `linkCapFn`, `nodeCapFn`, and `nodeBudgetFn`***

Similarly, `addRouteAllConstraint` implies:

$$\forall c \in \texttt{classes}: \; a_c = 1$$

Due to space limitations, we do not provide the formal basis for `addEnforceSinglePath`.

4.3 Resource capacity constraints

As we saw in §2, SDN optimizations have to deal with a variety of capacity constraints for network resources such as link bandwidth, switch rules, and middlebox CPU and memory. SOL allows users to write custom resource management logic by specifying several "cost" functions, depicted in Fig. 10. These functions prescribe how to compute the cost of routing traffic through a link, a node, or a given path. SOL provides default implementations of these for common tasks, but allows the user to specify their own logic, as well, as we will show later (§6).

These cost functions can then be passed into constraint templates. For example, to add a constraint that limits link usage, the user can invoke the template function `addLinkCapacityConstraint` with a resource that we are constraining (e.g., 'bandwidth'), a map of links to their capacities,[2] and optionally, a custom `linkCapFn` to compute the cost of traffic on a link.

```
opt.addLinkCapacityConstraint ('bandwidth',
    {(1,2): 10**7, (2,3): 10**7},
    defaultLinkFunction)
```

This indicates that bandwidth should not exceed 10 Mbps for links 1-2 and 2-3. Note that the default function is purely for illustration; the developer can write her own `linkCapFn` (recall Fig. 10).

`addNodeCapacityPerPathConstraint` generates constraints on the nodes that do not depend on the traffic, but rather on the routing path. That is, the cost of routing at a node does not depend on the volume or type of traffic being routed; it depends on the path and its properties. The best example of such usage is accounting for the limited rule space on a network switch (e.g., §2.2). If a path is "active", the rule must be installed on each switch to support the path.

Internals: `addLinkCapacityConstraint` and `addNodeCapacityConstraint` rely on `linkCapFn` and `nodeCapFn`, respectively, to compute the cost of using a particular resource at a link or node if all of the class-c traffic was routed to it. Internally, the load is multiplied by the $x_{c,p}$ variable to capture the load accu-

[2] When capacities should be allocated by the optimization itself, a capacity of `TBA` (meaning To Be Allocated) can be specified, instead.

`linkCapFn`(l,c,p,r): Amount of resource type r consumed if all class-c traffic is allocated to path $p \ni l$ for link l

`nodeCapFn`(v,c,p,r): Amount of resource r consumed if all class-c traffic is allocated to path $p \ni v$ for node v

`nodeBudgetFn`(v): Cost of using node v; required with `addBudgetConstraint`

`routingCostFn`(p): Cost of routing along path p; required with `minRoutingCost`

`predicate`(p): Determine whether any given path is valid by returning `True` or `False`

Figure 10: ***Customizable functions***

rately, then the load is capped by a user-provided *lnCap* (*ndCap*), which is a mapping of links (nodes) to capacities for a given resource r. (Similar node capacity equations not shown for brevity.)

$$\forall l \text{ in } lnCap:$$
$$load_l^r = \sum_c \sum_{p \in \texttt{paths}(c): l \in p} x_{c,p} \times \texttt{linkCapFn}(l,c,p,r)$$
$$load_l^r \leq lnCap[l]$$

The `addNodeCapacityPerPathConstraint` functions a bit differently, as it depends on enabled paths:

$$\forall v \text{ in } ndCap:$$
$$load_v^r = \sum_c \sum_{p \in \texttt{paths}(c): v \in p} b_p \times \texttt{nodeCapFn}(v,c,p,r)$$
$$load_v^r \leq capvar_v^r$$
$$\text{if } ndCap[v] \neq \texttt{TBA} \text{ then } capvar_v^r = ndCap[v]$$

4.4 Node/link activation constraints

Next set of constraints, when used, allow developers to logically model the act of *enabling* or *disabling* nodes, links, and paths; e.g., for managing energy or other costs (e.g., §2.3). We identify two possible modes of interactions between these topology modifiers, and the optimization developer can choose the one that is most suitable for their context. 1) `addRequireAllNodesConstraint` captures the property that disabling a node disables all paths that traverse it; and 2) `addRequireSomeNodesConstraint` captures the property that enabling a node permits any path traversing it to be enabled, as well. The latter version is suitable when, e.g., a node can still route traffic even if its other (middlebox) functionality is disabled, and so a path containing that node is potentially useful as providing middlebox functions if at least one of its nodes is enabled. There are analogous constraint templates for links, but we omit them here for brevity. A third constraint template, `addPathDisableConstraint`, restricts a path to carry traffic only if it is enabled.

For example, a developer trying to implement the application from §2.3 can model the requirements for shutting off datacenter nodes by adding the `addRequireAllNodesConstraint` and `addPathDisableConstraint` templates:

```
opt.addRequireAllNodesConstraint (trafficClasses)
opt.addPathDisableConstraint (trafficClasses)
```

Other efficiency considerations may enforce a budget on the number of enabled nodes, to model constraints on total power consumption of switches/middleboxes, cost and budget of installing/upgrading particular switches, etc. These are captured via the `addBudgetConstraint` template function.

Internals: Internally, these topology modification templates are achieved using the binary variables we introduced earlier. Specifically, the above requirements can be formalized as follows:

	$\forall p \in \texttt{paths}(c):$
`addRequireAllNodesConstraint`	$\forall v \in p:\ b_p \leq b_v$
`addRequireSomeNodesConstraint`	$b_p \leq \sum_{v \in p} b_v$
`addPathDisableConstraint`	$x_{c,p} \leq b_p$

Naturally, similar constraints are constructed for links. Note that `addPathDisableConstraint` is crucial to the correctness of the optimization in that it enforces that no traffic traverses a disabled path. For brevity, we do not provide the formal equations for `addBudgetConstraint`.

4.5 Specifying network objectives

The goal of SDN applications is eventually to optimize some network-wide objective, e.g., maximizing the network throughput, balancing load, or minimizing total traffic footprint. Fig. 11 lists the most common objective functions, drawing on the applications considered in §2. For instance, the developer of a TE application may want to implement the objective of minimizing the maximum link load and thus add the following code snippet:

```
opt.setPredefinedObjective (minMaxLinkLoad,
    'bandwidth')
```

Other optimizations (e.g., §2.4) may need to minimize the total routing cost and include a `minRoutingCost` objective. This objective is parameterized with `routingCostFn`(p); i.e., developers can plugin their own cost metrics such as number of hops or link weights. As shown, we also provide a range of natural load-balancing templates. SOL also exposes a low-level API for specifying other complex objective functions, which we describe in Appendix A.

5 Path generation and selection

Given these constraint templates, the remaining question is how we populate the path set `paths`(c) for each

`maxAllFlow`	maximize $\sum_{c \in \texttt{classes}} a_c$
`minMaxNodeLoad` (r)	minimize $\max_{v \in \texttt{nodes}} load_v^r$
`minMaxLinkLoad` (r)	minimize $\max_{l \in \texttt{links}} load_l^r$
`minRoutingCost`	$\sum_{c,p} \texttt{routingCostFn}(p) \times x_{c,p}$

Figure 11: ***Common objective functions***

traffic class c to meet two requirements. First, each $p \in \texttt{paths}(c)$ should satisfy the desired policy specification for the class c. Second, $\texttt{paths}(c)$ should contain paths for each class c that make the formulation tractable and yet yield near-optimal results. We describe how we address each concern next.

Generation: First, to populate the paths, SOL does an offline enumeration of all simple (i.e., no loops) paths per class.[3] Given this set, we filter out the paths that do not satisfy the user-defined predicate `predicate`, i.e., where $\texttt{predicate}(p) = \texttt{True}$ only if p is a valid path. (We can generalize this to allow different predicates per class; not shown for brevity.)

In practice, we implement the predicate as a flexible `Python` callable function rather than constrain ourselves to specific notions of path validity (e.g., regular expressions as in prior work [45]). Using this predicate gives the user flexibility to capture a range of possible requirements. Examples include waypoint enforcement (forcing traffic through a series of middleboxes in order); enforcing redundant processing (e.g., through multiple IDS, in case one fails open); and limiting network latency by mandating shorter paths.

Selection: Using all valid paths per class may be inefficient since the number of paths grows exponentially with the size of the network, meaning that the LP/ILP that SOL generates will quickly become too large to solve in reasonable time. SOL thus provides path selection algorithms that choose a subset of valid paths (number of paths denoted as `selectNumber`) that are still likely to yield near-optimal results in practice. Specifically, two natural methods work well across the spectrum of applications we have considered: (1) shortest paths for latency-sensitive applications (`selectStrategy = shortest`) or (2) random paths for applications involving load balancing (`selectStrategy = random`). SOL is flexible to incorporate other selection strategies, e.g., picking paths with minimal node overlap for fault tolerance. We find `random` works well for many applications that require load balancing. We conjecture this is because choosing random paths on sufficiently rich topologies yields a high degree of edge-disjointedness among the chosen paths, yielding sufficient degrees of freedom for balancing loads.

[3] This is to simplify the forwarding rules without resorting to tunneling or packet tagging [39].

```
SIMPLE_predicate = functools.partial(waypointMboxPredicate
    , order=('fw','ids'))
def SIMPLE_NodeCapFunc(node,tc,path,resource,nodeCaps):
  if resource=='cpu' and node in nodeCaps['cpu']:
    return tc.volFlows*tc.cpuCost/nodeCaps[resource][node]

capFunc = functools.partial(SIMPLE_NodeCapFunc, nodeCaps=
    nodeCaps)

def SIMPLE_TCAMFunc(node, tc, path, resource):
  return 1
# Path generation, typically run once in a precomputation
    phase
opt = getOptimization()
pptc = generatePathsPerTrafficClass(topo, trafficClasses,
    SIMPLE_predicate, 10, 1000,
     functools.partial(useMboxModifier, chainLength=2))
# Allocate traffic to paths
pptc = chooserand(pptc, 5)
opt.addDecisionVariables(pptc)
opt.addBinaryVariables(pptc, topo, ['path','node'])
opt.addAllocateFlowConstraint(pptc)
opt.addRouteAllConstraint(pptc)
opt.addLinkCapacityConstraint(pptc, 'bandwidth', linkCaps,
     defaultLinkFuncNoNormalize)
opt.addNodeCapacityConstraint(pptc, 'cpu',
    {node: 1 for node in topo.nodes() if 'fw' or
    'ids' in topo.getServiceTypes(node)}, capFunc)
opt.addNodeCapacityPerPathConstraint(pptc, 'tcam',
    nodeCaps['tcam'], SIMPLE_TCAMFunc)
opt.setPredefinedObjective('minmaxnodeload','cpu')
opt.solve()
obj = opt.getSolvedObjective()
pathFractions = opt.getPathFractions(pptc)
c = controller()
c.pushRoutes(c.getRoutes(pathFractions))
```

Figure 12: ***Code to express SIMPLE [39] in SOL***

Developer API: The developer can specify the path predicate and selection strategy, but she does not need to be involved in the low-level details of generation and selection. SOL also provides APIs for developers to add their own logic for generation and selection; we do not discuss these due to space limitations.

6 Examples

Next, we show end-to-end examples to highlight the ease of using the SOL APIs to write existing and novel SDN network optimizations. These examples are actual `Python` code that can be run, not just pseudocode. By comparison, the code is significantly higher-level and more readable than the equivalent `CPLEX` code would be, as it does not need to deal with large numbers of underlying variables and constraints.

Service chaining (§2.2): As a concrete instance of the service chaining example, we consider SIMPLE [39]. SIMPLE involves the following requirements: route all traffic through the network, enforce the service chain (e.g., "firewall followed by IDS") policy for all traffic, load balance across middleboxes, and do so while respecting CPU, TCAM, and bandwidth requirements. Fig. 12 shows how the SIMPLE optimization can be written in ≈ 25 lines of code. This listing assumes that topology and traffic classes have been set up, in the `topo` and

`trafficClasses` objects, respectively.

The first part of the figure shows function definitions and the path generation step, which would typically be performed once as a precomputation step. We start by defining a path predicate (line 1) for basic enforcement through middleboxes by using the SOL-provided function with the middlebox order. The next few lines (lines 2–4) show a custom node capacity function to normalize the CPU load between 0 and 1. This computes the processing cost per traffic class (number of flows times CPU cost) normalized by the current node's capacity. Similarly, the TCAM capacity function captures that each path consumes a single rule per switch (line 7). The user gets the optimization object (line 10), and generates the paths (line 11), obtaining the "paths per traffic class" (`pptc`) object. The path generation algorithm is parameterized with the custom `SIMPLE_predicate`, a limit on path length of 10 nodes, and a limit on the number of paths per class of 1000. It is also instructed to evaluate every possible use of two middleboxes on a routing path for inclusion as a distinct path in the output.

The remaining lines show what would be executed whenever a new allocation of traffic to paths is desired. Line 13 selects 5 random paths per traffic class; lines 14–20 add the routing and capacity constraints. We use the default link capacity function for bandwidth constraints, and our own functions for CPU and TCAM capacity. Because the CPU capacity function normalizes the load, the capacity of each node is now 1 (line 19). The program selects a predefined objective to minimize the CPU load (line 21) and calls the solver (line 22). Finally, the program gets the results and interacts with the SDN controller to automatically install the rules (line 26).

ElasticTree [19]: Due to space limitations we only show the most important differences between ElasticTree and SIMPLE. There is no requirement on paths, and so `nullPredicate` is used for path generation. We use link binary variables (see line 1 below) and the node/link activation constraints (lines 2–4). Finally, we use the low-level API (see App. A) to define power consumption for switches and links (lines 5, 6, wherein "`opt.bn(node)`" and "`opt.be(u,v)`" retrieve the names of variables $b_{\texttt{node}}$ and $b_{(\texttt{u},\texttt{v})}$ from Fig. 8, respectively) and use these variables to define a custom objective function (line 7).

```
1 opt.addBinaryVariables(pptc,topo,['path','node','edge'])
2 opt.addRequireAllNodesConstraint(pptc)
3 opt.addRequireAllEdgesConstraint(pptc)
4 opt.addPathDisableConstraint(pptc)
5 opt.defineVar('SwitchPower', {opt.bn(node):switchPower[
      node] for node in topo.nodes()})
6 opt.defineVar('LinkPower', {opt.be(u, v): linkPower[(u, v
      )] for u, v in topo.links()})
7 opt.setObjectiveCoeff({'SwitchPower': .75, 'LinkPower':
      .25}, 'min')
```

We refer the reader to Appendix B for other examples that include new and more complex applications.

7 Implementation

Developer interface: We currently provide a `Python` API for SDN optimization that is an extended version of the interface described in §4.

Invoking solvers: We use `CPLEX` (via its existing `Python` API) as our underlying solver. This choice largely reflects our familiarity with the tool, and we could substitute `CPLEX` with other solvers like `Gurobi`. SOL offers APIs to exploit solver capabilities to use a previously computed solution and incrementally find a new solution. This approach is typically faster than starting from scratch and so is useful for faster reconfigurations. SOL also allows hard-limiting of the optimization runtime, albeit affecting the optimality of the solution.

Path generation: Path generation is an inherently parallelizable process; we simply launch separate `Python` processes for different traffic classes. We currently support two path selection algorithms: `random` and `shortest`. It is easy to add more algorithms as new applications emerge.

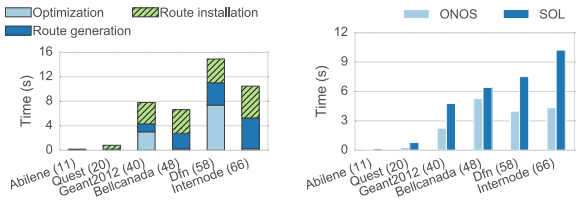
Rule generation and control interface: We implement applications for `ONOS` [5] and use custom REST API to allow remote batch installation of the relevant rules. We generate the rules based on the optimization output, using network prefix splitting to implement the fractional responsibilities represented by the $x_{c,p}$ variables. This step is similar to prior work that map fractional processing and forwarding responsibilities onto network flows (e.g., [47, 20]), and so we do not repeat it here. With `ONOS`, we leverage *path intents* [5]: while not required, it facilitates easier integration.

Minimizing reconfiguration changes: Networks are in flux during reconfigurations with potential performance or consistency implications, and thus it is desirable to minimize unnecessary configuration changes. SOL supports constraints that bound (or minimize) the logical distance between a previous solution and the new solution to help minimize the number of flows that have to be assigned a new route. In this way, SOL supports path selection that gives priority to previously selected paths.

8 Evaluation

In this section we show that SOL

- performs well with the `ONOS` controller;
- computes optimal solutions for published applications order(s) of magnitude faster than their original optimizations; allows to minimize traffic churn
- is either faster or has richer functionality than state-of-the-art related work;



(a) ***Time for SOL to configure the network using the ONOS controller for a traffic engineering application.***

(b) ***Route generation & installation time of SOL traffic engineering app vs. ONOS all-pair shortest paths***

Figure 13: ***Deployment benchmarks using the ONOS controller***

- significantly reduces development effort in comparison to manually coding optimization applications; and
- scales well, because it computes near-optimal solutions using few paths per traffic class.

Setup: We evaluate the effect of using SOL to implement three existing SDN applications: ElasticTree [19], SIMPLE [39], and Panopticon [29]. For each application, we implemented the original formulation presented in prior work or obtained the original source code. We refer to these as "original" formulations (and solutions). We chose topologies of various sizes from the TopologyZoo dataset [28]; when indicating a topology, we generally include the number of nodes in the topology in parentheses, e.g., "Abilene (11)" for the 11-node Abilene topology. For ElasticTree, we also constructed FatTree topologies of various sizes [2]. We synthetically generated traffic matrices using a uniform traffic matrix for the FatTree networks and a gravity-based model [43] for the TopologyZoo topologies. We used randomly sampled values from a log-normal distribution as "populations" for the gravity model. Unless otherwise specified, we used 5 paths per traffic class when running SOL. All times below refer to computation on computers with 2.4GHz cores and 128GB of RAM. For deployment benchmarks, we used the default Mininet [31] virtual machine to emulate topologies.

8.1 Deployment benchmarks

We setup a variety of emulated networks using `Mininet` and `ONOS`. We measured time for SOL to run the optimization for a traffic engineering goal and compute and install network routes. Fig. 13a shows the times to perform each step. SOL exhibits low optimization and route generation times, making route installation the most time-consuming part of the configuration process. This bottleneck is caused by the number of rules that must be installed and by the controller platform. Fig. 13b shows the time to generate and install routes for a traffic engineering application using SOL, in contrast to installing shortest path routes using methods available in `ONOS`. The difference is insignificant, and exists due to the additional optimization time and because of the multiple paths per source-destination pair in the SOL case.

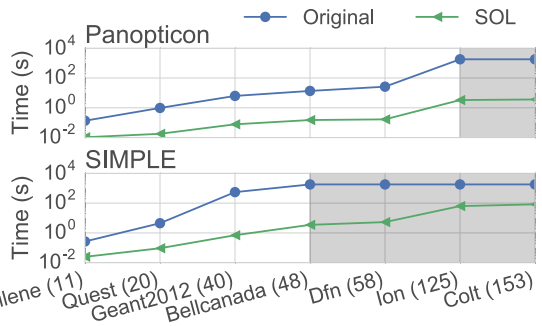


Figure 14: ***Optimization runtime of SOL and original formulations; gray regions show where original formulation could not be solved within 30 mins***

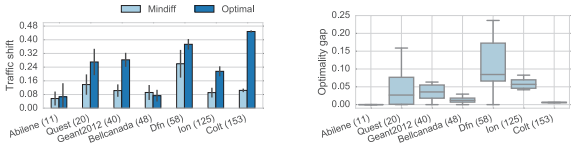
8.2 Optimality and scalability

Comparing to optimal: Next, we examine how well SOL's results match original solutions, which are optimal (by definition). In all cases except ElasticTree, SOL finds the optimal solution. Due to complexity of ElasticTree's optimization, SOL suffers a 10% optimality gap: the relative error in the objective value computed by SOL (i.e., relative to the true optimal objective value).

SOL solution times are at least one order of magnitude faster than solving the original formulations, and are often two or even three orders of magnitude faster. Fig. 14 shows run times to find original solutions. The runtime was capped at 30 min (1800 s), after which the execution was aborted. Several original formulations did not complete in that time, such as SIMPLE for topologies Bellcanada and larger, and Panopticon for Ion and larger. The topologies for which original solutions could not be found are indicated in the gray regions in Fig. 14.

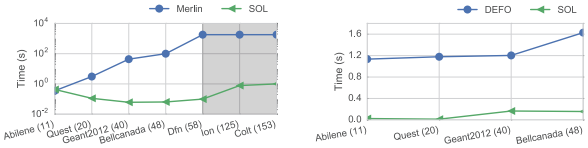
Comparing to specialized heuristics: We found that SOL performs fairly well even compared to specialized heuristics. Specifically, we compared the performance of SOL to the custom heuristic for SIMPLE, obtained from its authors. The runtime of SOL is comparable to that of the SIMPLE heuristic algorithm, with a performance gap of at most 3 seconds on the largest topologies we considered (up 58 nodes, namely the "Dfn" topology). We believe the benefit of not having to design custom heuristics outweighs this performance gap.

Responding to traffic changes: We explore the benefits of the reconfiguration minimization capabilities of SOL, for simplicity dubbed "mindiff." We first computed an optimal solution for a traffic engineering application; then, a random permutation of the traffic matrix triggered the re-computation with mindiff enabled. When computing the new solution, we assigned 4× greater priority to the TE objective than the mindiff objective. Fig. 15a shows that with mindiff enabled, up to an additional 35%



(a) ***Fraction of traffic reassigned to different paths with and without "mindiff"*** **(b)** ***Optimality gap when using "mindiff"***

Figure 15: ***Traffic shift and optimality gap when using reconfiguration minimization capabilities of SOL***



(a) ***Optimization runtimes of SOL and Merlin; gray region indicates where Merlin did not complete in 30 mins*** **(b)** ***Optimization runtimes of SOL and DEFO, for a traffic engineering application***

Figure 16: ***Runtime of SOL vs. state-of-the-art optimization frameworks***

of *total* traffic stays on its original paths across reconfigurations, versus being reassigned to new paths by the optimal solution. Naturally, SOL sacrifices some optimality in the original TE objective (shown in Fig. 15b).

8.3 Comparison to Merlin and DEFO

Merlin [45] tackles problems of network resource management similar to SOL. While the goals and formulations of Merlin and SOL are quite different, we use this comparison to highlight the generality of SOL and the power of its path abstraction. Specifically, Merlin uses a more heavyweight optimization that is always an ILP and operates on a graph that is substantially larger than the physical network. We implemented the example application taken from the Merlin paper using both SOL and Merlin. Fig. 16a shows that SOL outperforms Merlin by two or more orders of magnitude.

DEFO [18] is an optimization framework that aims to simplify traffic engineering [18]. We obtained the DEFO authors' implementation and compared the optimization times of DEFO and SOL on a simple traffic engineering application. DEFO and SOL exhibit comparable runtimes (see Fig. 16b). However, DEFO lacks the ability to express more complex applications and objectives and to filter paths by arbitrary predicates.

8.4 Developer benefits

SOL is a much simpler framework for encoding SDN optimization tasks, versus developing custom solutions by hand. In an effort to demonstrate this simplicity somewhat quantitatively, Table 1 shows the number of lines of code (LOC) in our SOL implementations of various applications ("SOL lines of code"), and the ratio of the LOC of the original formulations to the LOC for our SOL implementations ("Estimated improvement"). We acknowledge that lines-of-code comparisons are inexact, but we do not know of other ways of comparing "development effort" without conducting user studies.

Name	SOL lines of code	Estimated improvement
ElasticTree	16	21.8×
Panopticon	13	25.7×
SIMPLE	21	18.6×

Table 1: ***Development effort benefits provided by SOL***

We believe that the improvements in Table 1 are conservative. First, producing original formulations is a much more complex and delicate process than writing SOL code. We primarily attribute this difference to needing to account for `CPLEX` (or other solvers, e.g., [17, 33]) particulars at all; with SOL, these particulars are completely hidden from the developer. Second, SOL translates its optimization results to device configurations, whereas this functionality is not even included in our scripts for producing original formulations. Producing device configurations from original solutions would require designing an extra algorithm to map the variables in each formulation to relevant device configurations.

8.5 Sensitivity

SOL solutions require the specification of both the number (`selectNumber`) and type (`shortest` or `random`) of paths to select per traffic class. In this section, we quantify how sensitive SOL is to these parameters.

Number of Paths: Fig. 17 shows the SOL's runtime and optimality gap as a function of the number of paths per class for two applications: SIMPLE and Panopticon. Unsurprisingly, with a larger number of paths, SOL's runtime increases. However, this is not a significant concern, since we find optimal solutions at `selectNumber` as low as 5. These numbers are representative of all applications and topologies we have considered.

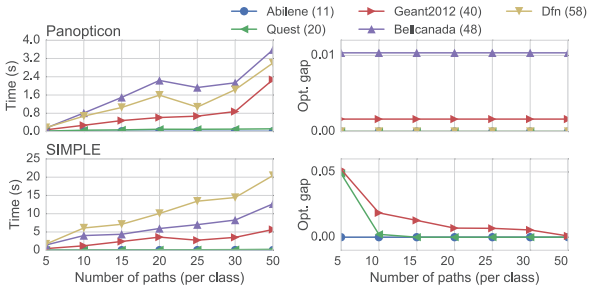


Figure 17: ***Runtime and optimality gap as function of paths; optimality is achieved in most cases with as few as 5 paths per class***

Path selection strategy: We evaluated different selection strategies across topologies and applications (omit-

ted for brevity). Our results were consistent with our experiences more generally that most problems lend themselves to a fairly obvious path selection strategy: those with need for load balancing are likely to benefit from `random` and those that are latency-sensitive benefit from `shortest`. If in doubt, however, both strategies can be attempted.

Path selection and generation costs: Since path selection is part of the optimization cycle, we ensure that path selection times are small, ranging from 0.1 to 3 seconds across topologies. Path selection is preceded by a path generation phase that enumerates the simple paths per class. Path generation is moderately costly for large topologies, e.g., taking <300 s for the largest presented topology, when parallelized to 60 threads. However, we emphasize that path generation can be relegated to an offline precomputation phase that is only performed once.

9 Discussion

Expressiveness of SOL: We make no claim that SOL is a panacea, capable of expressing *any* optimization, nor do we have a formal way to decide if a problem fits the SOL paradigm. We can provide guidelines as to which problems are well-suited (or not) for SOL. Optimizations with complex path predicates benefit greatly from SOL, as path generation and validation is performed offline, not during optimization. So do problems with resource constraints dependent on paths (e.g., SIMPLE with its TCAM constraints). Problems with no path predicates and very large interconnected topologies (i.e., large datacenter networks) are less likely to benefit from SOL. However, we plan to explore alternative approaches (e.g., hierarchical optimization) approaches to provide benefits in that space as well.

Analytical guarantees: While our empirical results suggest that `random` or `shortest` paths yield near-optimal solutions with a small `selectNumber`, they also raise interesting theoretical questions: can we prove that these selection strategies permit near-optimal solutions for specific classes of problems?

Very large/dynamic networks: For very large networks (>100 nodes) SOL might not perform as well as heuristic solutions, especially for applications that require an ILP, since the number of paths grows very large. In such cases, we can utilize general approximation heuristics, such as randomized rounding, to maintain its ability to react to network changes quickly.

Composition: Having a unified optimization layer atop SDN controllers exposes opportunities to compose applications. We plan to explore these in future work.

10 Related Work

We already discussed the optimization applications that motivated SOL. Here we focus on other related work.

Higher-layer abstractions for SDN: This work includes new programming languages (e.g., [41, 12]), testing and verification tools (e.g, [27]), semantics for network updates (e.g., [42]), compilers for rule generation (e.g., [26]), abstractions for handling control conflicts (e.g., [4]), and APIs for users to express requirements (e.g., [10]). These works do not address the optimization component, which is the focus of SOL.

Languages for optimization: There are several modeling frameworks such as `AMPL` [13], `Mosek` [33], `PyOpt` [37], and `PuLP` [32] for expressing optimization tasks. However, these do not specifically simplify network optimization. SOL is a domain-specific library that operates at a higher level of semantics than these "wrappers". SOL offers a path-based abstraction for writing network optimizations, exploits this structure to solve these optimizations quickly, and generates network device configurations that implement its solutions.

Network resource management: Merlin is a language for network resource management [45]. In terms of the applications that it can support, Merlin is restricted to using path predicates expressed as regular expressions. Our experiments suggest that SOL is three orders of magnitude faster than Merlin using the same underlying solvers. That said, Merlin's "language-based" approach provides other capabilities (e.g., verified delegation) that SOL does not (try to) offer. DEFO is another optimization framework that focuses on traffic engineering and service chaining applications [18]. Their goal is not to develop a general framework, but rather to support easy management of carrier-grade networks, which they accomplish using a two-layer architecture and support for networks that are not OpenFlow-enabled via segment routing. Other works focus on traffic-steering optimization (e.g., [39, 7]). SOL offers a unifying abstraction that covers many network management applications.

11 Conclusion

While network optimization is central to many SDN applications, few efforts attempt to make it accessible. Our vision is a general, efficient framework for expressing and solving network optimizations. Our framework, SOL, achieves both generality and efficiency via a path-centric abstraction. We showed that SOL can concisely express applications with diverse goals (traffic engineering, offloading, topology modification, service chaining, etc.) and yields optimal or near-optimal solutions with often better performance than custom formulations. Thus, SOL can lower the barrier to entry for novel SDN network optimization applications.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Ankit Singla, for helpful comments and guidance. This

work was supported in part by grant N00014-13-1-0048 from the Office of Naval Research, NSF grants 1535917 and 1536002, an NSF Graduate Research Fellowship, the Science of Security Lablet at North Carolina State University, and by Intel Labs' University Research Office.

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network flows: theory, algorithms, and applications*. Prentice hall, 1993.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 63–74, 2008.

[3] M. Allalouf and Y. Shavitt. Centralized and distributed algorithms for routing and weighted max-min fair bandwidth allocation. *ACM/IEEE Transactions on Networking*, 16(5):1015–1024, 2008.

[4] A. AuYoung, S. Banerjee, J. Lee, J. C. Mogul, J. Mudigonda, L. Popa, P. Sharma, and Y. Turner. Corybantic: Towards the modular composition of SDN control programs. In *ACM HotNets*, 2013.

[5] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, et al. ONOS: towards an open, distributed SDN OS. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 1–6. ACM, 2014.

[6] A. Bremler-Barr, Y. Harchol, D. Hay, and Y. Koral. Deep packet inspection as a service. In *ACM CoNEXT*, pages 271–282, 2014.

[7] Z. Cao, M. Kodialam, and T. Lakshman. Traffic steering in software defined networks: planning and online routing. In *ACM SIGCOMM Workshop on Distributed Cloud Computing*, pages 65–70, 2014.

[8] M. Charikar, Y. Naamad, J. Rexford, and K. Zou. Multi-Commodity Flow with In-Network Processing. Manuscript, www.cs.princeton.edu/~jrex/papers/mopt14.pdf.

[9] E. Danna, S. Mandal, and A. Singh. A practical algorithm for balancing the max-min fairness and throughput objectives in traffic engineering. In *IEEE Conference on Computer Communications*, pages 846–854, 2012.

[10] A. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory networking: An API for application control of SDNs. In *ACM SIGCOMM*, August 2013.

[11] B. Fortz and M. Thorup. Internet traffic engineering by optimizing OSPF weights. In *IEEE Conference on Computer Communications*, volume 2, 2000.

[12] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *ACM SIGPLAN Notices*, volume 46, pages 279–291, 2011.

[13] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A mathematical programming language*. AT&T Bell Laboratories Murray Hill, 1987.

[14] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, V. Sekar, and A. Akella. Stratos: A network-aware orchestration layer for virtual middleboxes in clouds. *arXiv preprint arXiv:1305.0209*, 2013.

[15] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling innovation in network function control. In *ACM SIGCOMM*, 2014.

[16] G. Gibb, H. Zeng, and N. McKeown. Outsourcing network functionality. In *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 2012.

[17] Gurobi. http://www.gurobi.com/.

[18] R. Hartert, S. Vissicchio, P. Schaus, O. Bonaventure, C. Filsfils, T. Telkamp, and P. Francois. A declarative and expressive approach to control forwarding paths in carrier-grade networks. In *ACM SIGCOMM*, 2015.

[19] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. ElasticTree: Saving energy in data center networks. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 19–21, 2010.

[20] V. Heorhiadi, S. K. Fayaz, M. K. Reiter, and V. Sekar. SNIPS: A software-defined approach for scaling intrusion prevention systems via offloading. In *10th International Conference on Information Systems Security*, Dec. 2014.

[21] V. Heorhiadi, M. K. Reiter, and V. Sekar. New opportunities for load balancing in network-wide intrusion detection systems. In *ACM CoNEXT*, 2012.

[22] V. Heorhiadi, M. K. Reiter, and V. Sekar. SOL bitbucket repository. https://bitbucket.org/progwriter/sol/, 2015.

[23] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. In *ACM SIGCOMM*, pages 15–26, 2013.

[24] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined WAN. In *ACM SIGCOMM*, pages 3–14, 2013.

[25] X. Jin, L. E. Li, L. Vanbever, and J. Rexford. Softcell: Scalable and flexible cellular core network architecture. In *ACM CoNEXT*, 2013.

[26] N. Kang, Z. Liu, J. Rexford, and D. Walker. Optimizing the one big switch abstraction in software-defined networks. In *ACM CoNEXT*, pages 13–24, 2013.

[27] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *USENIX Symposium on Networked Systems Design and Implementation*, 2012.

[28] S. Knight, H. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765 –1775, October 2011.

[29] D. Levin, M. Canini, S. Schmid, F. Schaffert, A. Feldmann, et al. Panopticon: Reaping the benefits of incremental sdn deployment in enterprise networks. In *USENIX Annual Technical Conference*, 2014.

[30] X. Liu, S. Mohanraj, M. Pioro, and D. Medhi. Multipath routing from a traffic engineering perspective: How beneficial is it? In *IEEE International Conference on Network Protocols*, pages 143–154, 2014.

[31] Mininet. http://mininet.org/.

[32] S. Mitchell, M. O'Sullivan, and I. Dunning. Pulp: a linear programming toolkit for python, 2011.

[33] Mosek. https://mosek.com/.

[34] Network functions virtualisation – introductory white paper. http://portal.etsi.org/NFV/NFV_White_Paper.pdf.

[35] Opendaylight SDN controller. http://www.opendaylight.org/.

[36] S. Palkar, C. Lan, S. Han, K. Jang, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A runtime framework for network functions. In *ACM Symposium on Operating Systems Principles*, 2015.

[37] R. E. Perez, P. W. Jansen, and J. R. R. A. Martins. pyOpt: A Python-based object-oriented framework for nonlinear constrained optimization. *Structures and Multidisciplinary Optimization*, 45(1):101–118, 2012.

[38] M. Pióro, P. Nilsson, E. Kubilinskas, and G. Fodor. On efficient max-min fair routing algorithms. In *Computers and Communication*, pages 365–372. IEEE, 2003.

[39] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying middlebox policy enforcement using SDN. In *ACM SIGCOMM*, 2013.

[40] S. Raza, G. Huang, C.-N. Chuah, S. Seetharaman, and J. P. Singh. Measurouting: a framework for routing assisted traffic monitoring. *ACM/IEEE Transactions on Networking*, 20(1):45–56, 2012.

[41] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker. Modular SDN programming with pyretic. *;login: Magazine*, 38(5):128–134, 2013.

[42] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *ACM SIGCOMM*, 2012.

[43] M. Roughan. Simplifying the synthesis of internet traffic matrices. *ACM SIGCOMM Computer Communication Review*, 35, 2005.

[44] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *ACM SIGCOMM*, 2012.

[45] R. Soule, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster. Merlin: A language for provisioning network resources. In *ACM CoNEXT*, 2014.

[46] N. Vasić, P. Bhurat, D. Novaković, M. Canini, S. Shekhar, and D. Kostić. Identifying and using energy-critical paths. In *ACM CoNEXT*, 2011.

[47] R. Wang, D. Butnariu, and J. Rexford. Openflow-based server load balancing gone wild. In *Hot-ICE*, 2011.

A Advanced users and low-level interface

While the SOL API described in §4 is general and expressive enough to capture the diverse requirements of the broad spectrum of applications, we also expose a low-level API that gives more control to the user by giving access to the SOL internal variables. Advanced users can use this API for further customization.

For instance, API calls enable the names of the internal variables in Fig. 8 to be retrieved and their values determined. Similarly, using the `defineVar` (*name*, *coeffs*, *lb*, *ub*) function, the user can create a new variable with name *name*, specify numeric lower and upper bounds (*lb* and *ub*), and equate it to a linear combination of any other existing variables as specified by *coeffs*, a map from variable names to numeric coefficients. This is a useful primitive when specifying complex objectives. SOL also allows setting a custom objective function that is a linear combination of any existing variables, allowing for multi-objective optimization. This is done using the `setObjective` (*coeffs*, *dir*) function call, which accepts a mapping *coeffs* of variable names to their coefficients. The binary input *dir* indicates whether the objective should be minimized or maximized.

B Additional applications

New elastic scaling capabilities: Here, we show SOL can be used for novel SDN applications. Specifically, we consider an elastic NFV setting [14] that places middleboxes in the network and allocates capacities in response to observed demand. There could be additional constraints, such as the total number of such VM locations. As a simple objective, we consider an upper bound on the number of nodes used while still load balancing across virtual middlebox instances. We can easily add other objectives such as minimizing number of VMs. For brevity we highlight only the key parts of building such a novel application.

```
1 predicate = hasMboxPredicate
2 opt.addBinaryVariables(pptc, topo, ['path', 'node'])
3 opt.addNodeCapacityConstraint(pptc, 'cpu', {node: 'TBA'
      for node in topo.nodes()}, lambda node, tc, path,
      resource: tc.volFlows * tc.cpuCost)
4 opt.addRequireSomeNodesConstraint(pptc)
5 opt.addPathDisableConstraint(pptc)
6 opt.addBudgetConstraint(topo, lambda node: 1, topo.
      getNumNodes()/2)
7 opt.setPredefinedObjective('minmaxnodeload', resource='
      cpu')
```

First, we define a valid path to be one that goes though a middlebox; SOL provides a predicate for that (line 1). The main difference here is the definition of capacities with the `TBA` value on line 3; this indicates that our optimization must allocate the capacities to the nodes. (SOL ensures that disabled nodes have 0 capacity allocated.) Thus we require at least one enabled node per path (lines 4, 5), limit the number of enabled nodes (line 6), and set the objective (line 7).

```
1  def _MaxMinFairness_MCF(topology, pptc, unstaturated,
       saturated, allocation, linkCaps):
2      opt = getOptimization()
3      opt.addDecisionVariables(pptc)
4      # setup flow constraints
5      opt.addAllocateFlowConstraint({tc: pptc[tc] for tc in
           unstaturated})
6      for i in saturated:
7          opt.addAllocateFlowConstraint({tc: pptc[tc] for
               tc in pptc[i]}, allocation[i])
8      # setup link capacities:
9      def linkcapfunc(link, tc, path, resource):
10         return tc.volBytes
11     opt.addLinkCapacityConstraint(pptc, 'bandwidth',
           linkCaps, linkcapfunc)
12     opt.setPredefinedObjective("maxallflow")
13     opt.solve()
14     return opt

16 def iterateMaxMinFairness(topology, pptc, linkCaps):
17     # Setup saturated and unsaturated commodities
18     saturated = defaultdict(lambda: [])
19     unsaturated = set(pptc.keys())
20     paths = defaultdict(lambda: [])

22     t = []  # allocation values per each iteration
23     i = 0  # iteration index
24     while unsaturated:
25         # Run slightly modified multi-commodity flow
26         opt = _MaxMinFairness_MCF(topology, pptc,
               unsaturated, saturated, t, linkCaps)
27         if not opt.isSolved():
28             raise FormulationException('No solution')
29         alloc = opt.getSolvedObjective()
30         t.append(alloc)
31         # Check if commodity is saturated, if so move it
               to saturated list
32         for tc in list(unsaturated):
33             # NOTE: this is an inefficient non-blocking
                   test, based on dual variables
34             # More efficient methods are available
35             dual = opt.getDualValue(opt.al(tc))
36             if dual > 0:
37                 unsaturated.remove(tc)
38                 saturated[i].append(tc)
39                 paths[tc] = opt.getPathFractions()[tc]
40         i += 1
41     return paths
```

Figure 18: *`Python` code for Max-min fairness optimization*

Complex multi-part optimizations: We also show how one can model more complex optimizations, using SOL as a primitive to express certain blocks of the optimization. Fig. 18 provides code for solving a max-min fairness problem. It relies on expressing intermediate multi-commodity flow problems using SOL (see function `_MaxMinFairness_MCF`) and writing a small iterative algorithm (see function `iterateMaxMinFairness`) for arriving at the optimal solution. We model our code after the algorithm suggested by Pióro et al. [38], however there are more recent and efficient proposals that can also be expressed in SOL (e.g., [3, 9]).