Palette: Distributing Tables in Software-Defined Networks

Yossi Kanizo, David Hay, and Isaac Keslassy

Abstract—**In software-defined networks (SDNs), the network controller first formulates abstract network-wide policies, and then implements them in the forwarding tables of network switches. However, fast SDN tables often cannot scale beyond a few hundred entries. This is because they typically include wildcards, and therefore are implemented using either expensive and power-hungry TCAMs, or complex and slow data structures.**

This paper presents the Palette distribution framework for decomposing large SDN tables into small ones and then distributing them across the network, while preserving the overall SDN policy semantics. Palette helps balance the sizes of the tables across the network, as well as reduce the total number of entries by sharing resources among different connections. It copes with two NP-hard optimization problems: *Decomposing* a large SDN table into equivalent subtables, and *distributing* the subtables such that each connection traverses each type of subtable at least once. To implement the Palette distribution framework, we introduce graph-theoretical formulations and algorithms, and show that they achieve close-to-optimal results in practice.

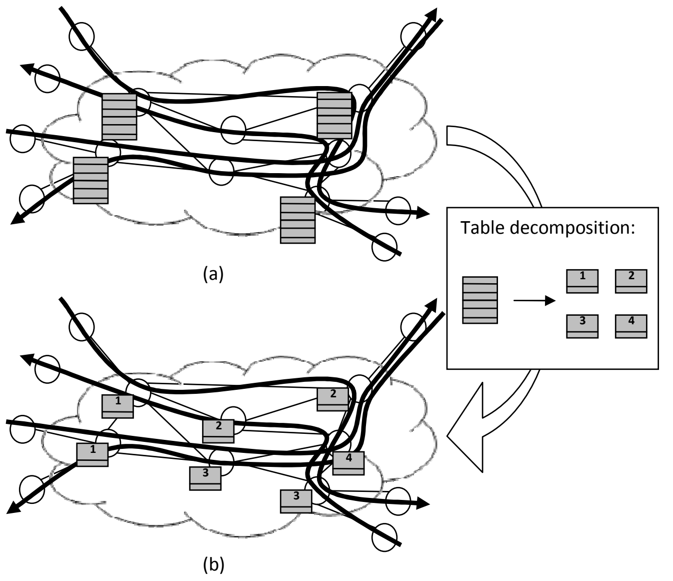


Fig. 1. (a) A common setting in which tables are installed at the network ingress nodes; (b) The result of applying Palette. Tables are decomposed into smaller subtables of different types (a.k.a. colors), which are then distributed across the network. A packet along each path meets each type of subtable at least once.

I. Introduction

A. Background

Software-defined networking (SDN) in general, and OpenFlow [1], [2] in particular, provide an abstraction of network devices and operations. This abstraction eases the development of new network protocols and policies. These protocols are implemented through the network *controller*, a single centralized device with a global view of the entire network. The network controller can be seen as a compiler that translates the abstract policies provided by network designers into specific rules in the table of each network switch.

Previous works typically assumed that the table of each switch can hold an infinite number of rules, which makes the compiler easy to design. In practice, however, this assumption does not hold, and the switch table sizes can become a significant bottleneck to scaling SDN networks. We note that many of these tables are implemented using ternary content-addressable memory (TCAM), which is extremely power-hungry and therefore of limited size. Typical implementations of OpenFlow, for example, limit the number of entries in each such table to only 750 [3], while handling about 100,000 concurrent flows.

This paper introduces the Palette framework for distributing these rules into a network of heterogeneous switches with tables of limited size, while preserving the semantics of the SDN policy. The Palette distribution framework is generic in the sense that it does not rely on the exact meaning of the rules, as long as the rules do not determine the routing of the packet. More specifically, the controller application should only specify whether the policy is routing/forwarding-agnostic or not, and should not deal with the implementation complexity of the distribution across the network switches. This is especially useful when the network topology changes, or equipment is replaced.

B. The Palette Framework

We turn now to describing our proposed Palette distribution framework. We define an SDN policy as a collection of rules. Each rule consists of a *(pattern, action)* pair, i.e. a pattern of specific bits in the packet header along with an action to take upon a pattern match (e.g. drop the packet or increment a counter of some measurement). For aggregation purposes, *don't-care* bits, denoted by "*", are allowed in the pattern. Therefore, a given packet header may match more than one rule, and in that case an action is taken according to the highest-priority rule. Typically, the SDN tables evolve over time (that is, new rules are added and some rules are deleted).

Y. Kanizo is with the Dept. of Computer Science, Technion, Haifa, Israel. Email: ykanizo@cs.technion.ac.il.

D. Hay is with the School of Computer Science and Engineering, Hebrew University, Jerusalem, Israel. Email: dhay@cs.huji.ac.il

I. Keslassy is with the Dept. of Electrical Engineering, Technion, Haifa, Israel. Email: isaac@ee.technion.ac.il.

In addition, occasionally a switch can send notifications (e.g., measurements taken in one of its built-in counters) to the controller.

Palette takes advantage of the fact that the controller has a global view of the network, and therefore, knows the paths taken by all packets. This allows us to share resources among different paths in an efficient way, by using the same rules for different paths in any common switch.

Fig. 1 illustrates our approach in a common setting used for *access control*. Access control consists of determining whether a given packet is permitted in the network or should be dropped. It is usually made by a switch, a router, or a designated Network Intrusion Detection/Prevention System (NIDS/NIPS) middlebox *at the edge of the network*. Specifically, as illustrated in Fig. 1(a), some access control is performed on all ingress nodes of the network. In an SDN setting, this typically translates into installing a table with all access control rules in all ingress nodes. However, as shown in Fig. 1(b), using our Palette framework, the rules can be distributed across the network switches.

Note that an orthogonal approach based on cSamp [4] would have been to divide the traffic among switches (that is, each switch deals only with part of the packets), by applying at each switch a hash function on certain packet header fields. Then, each switch handles a different range of hash values. This approach improves the data forwarding, as it reduces the load on each switch. However, it does not solve our problem, since each switch should still store the entire table, as the partitioning of the traffic does not take the rules into account.

C. *Our Contributions*

In this paper, we show how to split the rules across the network, such that each switch will have a smaller SDN table. Our basic approach is to divide this problem into two separate subproblems.

The *first subproblem* is to decompose a large table that contains all rules into a predetermined number of smaller tables. We denote each smaller table by a different color. Our decomposition ensures that *the Palette implementation preserves the overall network behavior*. In particular, it has the following two properties.

- *Order-oblivious:* The order in which the smaller tables are accessed does not change the global action of the network.
- *Semantically-invariant:* This global action of the network is the same as the one taken when using the initial single large table.

Naturally, this implies that not all actions can be decomposed and distributed: for example, a forwarding or a routing action (e.g., "send the packet through port i") must be taken in its original switch. We assume that the controller application specifies whether the action is safe to decompose or not. Incidentally, this subproblem is also useful in other contexts, such as achieving parallelism and power-efficiency in TCAMs [5], [6].

After obtaining a set of small tables (or colors), the *second subproblem* is to ensure that *each packet* traverses all the types of small tables (i.e. all colors), so that the resulting setting would be semantically equivalent to a single lookup in the large table.

We model this second problem in a graph-theoretic manner, in which the switches are nodes and the links are edges. We assume that all the possible end-to-end paths are known to the controller. Thus, our goal is to assign colors to the nodes such that each path is a *rainbow path*—a path whose nodes include all the colors. The objective is to maximize the number of colors that we can use, and therefore, to minimize the table size residing at each node.

Although constructing an optimal assignment is NP-hard, we show that when the network is a tree (which corresponds to datacenters) the problem is tractable. We then propose and evaluate sub-optimal greedy algorithms, and show that they achieve close-to-optimal results in practice.

In addition, we also study the multicolored node case where more than one type of subtable can reside in a single node; we show how this additional degree of freedom enhances the flexibility of our Palette distribution framework.

Finally, we evaluate the performance of our greedy algorithms, both in dividing TCAMs and in distributing tables across the network.

We note that dividing the problem into these two subproblems is done since both problems are fundamental and may be used in other contexts. In addition, we believe that it simplifies the presentation and evaluation of our Palette distribution framework. However, a joint optimization may yield better (yet more complex) results. We leave this joint optimization to future research.

D. *Paper Organization*

We start with the background on related work in Section II. Then, in Section III, we deal with our first subproblem, that is, we show how the large TCAM classifier can be divided into smaller TCAMs. In Section IV, we give the formulation of the second subproblem, which is NP-hard, analyze some special cases that can be solved efficiently, and present efficient greedy algorithms for approximating the general case. In Section V, we consider cases where more than one color can be assigned to each of the switches. Finally, Section VI provides experimental results.

II. Related Work

Software Defined Networking (SDN) has become an important paradigm in contemporary networks. Its key concept lies in the management of the entire network as a unified abstraction (e.g., in a network controller), and the remote control of the network devices (namely, its switches and routers) through open protocols (such as OpenFlow) [7]. In recent years, SDN technology has been widely deployed in real-life large-scale networks, e.g. Google's G-scale network [8]. Switches and routers that support SDN/OpenFlow are now offered by a large number of vendors (e.g., [3], [9], [10]).

One of the major challenges in SDN is to develop a programming language for its software development. On one hand, this programming language should be sufficiently flexible and rich to allow new network applications, but on the other hand, it should be simple and modular to reduce development and debugging times. Frenetic [11] is a prime example of such a network programming language that gives high-level abstractions to the network programmer. For example, it allows systematic updates [12] and task composition [11]. Perhaps the most closely-related work to ours is the extension of Frenetic to allow policy transformation of rules across the networks [13]. The authors developed a complete and sound set of axioms to allow semantically-preserving rule-rewriting in a single switch or in a chain of switches. Our paper complements this work by providing a specific algorithmic framework in which such a rewriting system can work, and also shows how to spread the resulting rules across the network. An additional key aspect of our paper that can be useful for Frenetic is our order-oblivious decomposition, which facilitates the distribution of subtables across the network.

Another approach for distributing table rules across the network is DIFANE [14]. In DIFANE, non-overlapping flow ranges are allocated using a decision-tree based algorithm. Then, each such range is assigned to a different predetermined subset of the switches, called authority switches. Following that, rules are installed to these authority switches according to the corresponding flow range.

To assure that each packet is matched against all relevant rules, ingress switches redirect the packets to the corresponding authority rule. If some rule is matched, a cache rule is generated using a technique described in [15], and then installed in the ingress switch, such that future packets from the same flow can be managed instantly in the ingress router.

Our approach avoids the management and redirection overhead by assuring that each packet is matched against all possible rule in the path it traverses. Further, it avoids duplications of the rules to the ingress routers, exploiting better the available space in the switches.

While both DIFANE and our approach do all rule processing in the data plane, a recent paper proposes to combine the rule processing both in the data plane and control plane [16]. The main approach is to partition the rules into non-overlapping sets of rules, and then to distribute it to both the switches (i.e., data plane) and the hypervisors (i.e., control plane), such that the volume of flows need to be redirected from the data plane to the control plane is minimized.

Finally, we note that splitting the workload between the switches in a coordinated network has been proposed in the past in many contexts. Such contexts includes traffic engineering [17], network diagnosis [18], [19], intrusion detection [20], and traffic monitoring [4], [21]–[26]. However, these solutions are not directly applicable to our case. For example, cSamp [4] is a generic framework for network measurement, where each flow is monitored only in one of the network routers. It uses a hash function with a certain distribution at each router to determine whether the current router has to perform the measurement. Since the hash functions are orthogonal to the monitoring rules, it implies that each such router should hold the entire monitoring table (but only access the table on a subset of the packets).

III. Order-Oblivious Table Decomposition

This section analyzes two approaches to dividing a large table into c subtables: the *Pivot Bit Decomposition* (PBD) and the *Cut-Based Decomposition* (CBD).

A. Decomposition Rules

Before starting, note that we only divide rules that correspond to policies which are *marked as safe to divide*. The rest of the rules remain in the corresponding sub-table, and will be re-composed with the rules assigned to that table after decomposition. There is a large body of work of how to compose several policies in one table (e.g., using a Cartesian product of the rules [27], [28]).

Hence, we are left with an arbitrary table that is safe to divide. We assume that this table must be able to match all possible strings. For this, we distinguish between *default* and *non-default* rules. The default rule consists of *don't-care* bits only, and it uses a default blank action (e.g., a *permit* action in ACLs). The non-default rules are all other rules in the table. Clearly, if a default rule exists in the original table, it may be placed only at the end of the table. To follow our convention, after the decomposition we add a default rule automatically to each of the resulting tables.

We further note that after decomposing the original table, the resulting tables may not be optimal. Therefore, it is possible to apply *logic minimization* on the resulting tables using off-the-shelf solutions developed in the context of TCAMs (e.g. [29]–[31]).

The correctness of the decomposition implies that each string that matches a non-default rule in the original table, must match a non-default rule in exactly one of the subtables (and the default rule in the other resulting tables). Moreover, strings that match the default rule in the original table, must also match the default rule in all subtables.

B. Pivot Bit Decomposition

The first method, called *Pivot Bit Decomposition (PBD)*, works by iteratively decomposing one table into two equivalent tables, thus increasing the total number of tables by 1.

This iterative decomposition is done by selecting one *pivot bit* (equivalently, one column) in the table, and splitting the rules into two sets: the first table holds all rules in which the pivot bit is 0, while the second table holds all rules in which the pivot bit is 1. Rules in which the pivot bit is "don't care" ("$*$") are rewritten as two complementary rules: one in which the pivot bit is replaced by 0 (and therefore, is part of the first table) and another in which it is replaced by 1 (and therefore, is part of the second table).

Note that while PBD decomposes the table along certain bits like previously-known methods [5], [6], it does not predetermine some P pivot bits and then decomposes the table into the corresponding 2^P tables. On the contrary, it adds a single new subtable at each iteration. Therefore, the different subtables may have resulted from different sets of pivot bits.

Naturally, the efficiency of the decomposition depends on the joint selection of the table and of the pivot bit at each

	0	1	2	3	4	5	6
φ_1	*	0	1	0	*	0	0
φ_2	0	*	1	*	*	*	0
φ_3	*	1	*	*	1	0	1
φ_4	1	1	1	*	1	*	*
φ_5	1	1	*	0	*	*	*
φ_6	1	0	0	1	0	1	*
φ_7	*	*	*	*	*	*	*

Fig. 2. An example rule-set of an SDN table.

iteration. *Our goal is to greedily minimize the maximum table size among all the small tables*. Therefore, we always decompose the largest table. The pivot bit is then selected to minimize the size of the largest table among the two resulting subtables.

Specifically, for $b \in \{0, 1, *\}$, let $n_i(b)$ be the number of bits with value b in the i-th column (that is, the number of rules whose i-th bit is b). Then, the pivot bit is:

$$\begin{aligned} \text{pivot} &= \operatorname*{argmin}_i \left(\max\left\{n_i(0) + n_i(*), n_i(1) + n_i(*)\right\}\right) \\ &= \operatorname*{argmin}_i \left(n_i(*) + \max\left\{n_i(0), n_i(1)\right\}\right) \qquad (1) \end{aligned}$$

Finally, note that if after the decomposition and possibly the logic minimization, the maximum table size is not reduced, we refrain from decomposing the table and move to the next table. The process ends either when we reach c tables, or when all possible decompositions do not result in table size reduction.

Example 1. *We demonstrate our decomposition using the example table depicted in Fig. 2. Assume we want to divide the table into $c = 2$ partitions. We disregard the default rule (φ_7), and choose bit number 1 as the pivot bit, since it minimizes the expression in Equation (1). Rule φ_2 has * in bit 1, and therefore it is duplicated to rule $\varphi_2' = 001{*}{*}{*}0$ and $\varphi_2'' = 011{*}{*}{*}0$. After the division, the rules φ_1, φ_2', and φ_6 are assigned to the first sub-table, while the rules $\varphi_2'', \varphi_3, \varphi_4$, and φ_5 are assigned to the second sub-table. We also need to add default rule φ_7 to both of the resulting sub-tables, so their final sizes would be 4 and 5, respectively.*

We next show the correctness of our decomposition and the fact that it is order-oblivious.

Lemma 1. *If a non-default rule is returned when applying a packet header h on the original table, then, after the PBD is applied, there is exactly one table that returns a non-default rule for h; all the other tables will return the default rule.*

Proof: Recall that packet headers are binary strings (that is, they do not contain *'s). In addition, notice that our decomposition procedure induces a binary tree structure among the tables, where each node in the tree represents a table, whose two descendants are the tables resulting in the decomposition; the root of the tree is the original table, while the leaves of the tree are the c subtables.

Suppose that two non-default rules are returned from two different tables T_0, T_1, and let T' be the deepest common ancestor of the tables. Let i be the pivot bit selected in decomposing T'. Thus, all rules (but the default rule) of T_0 differ from the rules in T_1 by the pivot bit. Let T_b be the table with value b in the i-th bit of the non-default rule. Let b' be the value of the i-th bit of h, therefore h cannot match any non-default rule in $T_{1-b'}$ and hence a contradiction.

We next show that at least one table returns a non-default rule for h. Such a table can be found by traversing the decomposition tree according to the value of the pivot bits in h. At each node T, whose pivot is i, we check the i-th bit in h and go to either T_0 or T_1 according to this value; the process is stopped when reaching a leaf T'. Let I be the set of pivot indices along the path between T' and the root of the decomposition tree. Note that by construction, the value of these $|I|$ bits in h and in all non-default rules of T' is the same. Assume that h matches some rule φ in the original table, let I_* be the set of indices in this rule that are *, and let I_{01} be the set of indices in this rule that are either 0 or 1. Since h matches this rule, h was carried on to T' for all pivots in $I \cap I_{01}$, and was duplicated (with the same action) for all pivots in $I \cap I_*$. Thus h matches the rule also in T'.

Note that since this claim is about the semantics of the rules, the claim still holds even when considering the logic minimization at each decomposition step, since logic minimization must preserve the semantic of each table. ■

The next lemma complements Lemma 1 and shows that correctness is maintained when matching the default rule:

Lemma 2. *If the default rule is returned when applying a packet header h on the original table, then all tables return the default rule.*

Proof: It is straightforward since no rule matches the packet header h in the original table, ■

The next theorem establishing the correctness of PBD follows immediately from Lemma 1 and Lemma 2:

Theorem 1. *The PBD scheme preserves the semantics of the original table, no matter the order in which the tables are accessed.*

We point out two main drawbacks in the basic PBD scheme. First, the basic PBD scheme divides the table at each iteration so that the maximum size of the resulting two subtable sizes is minimized. Therefore, the sizes of the resulting two subtables after each iteration tend to be almost equal. As a result, when c is not a power of two, it is expected that the partition sizes would be *imbalanced*.

To solve this problem, we generalize the PBD scheme in the following way: Given the target number of subtables c, we first find the largest integer p such that $2^p < c$. Then, we find a pivot bit that attempts to divide the table such that the ratio between the resulting table sizes will be $2^p{:}(c{-}2^p)$. [1] We recursively use this generalization of PBD on each of the two subtables, aiming to decompose the first subtable into 2^p smaller subtables, and the second subtable into $c - 2^p$ ones.

Another potential drawback of PBD is reflected in the following result, which exhibits an example in which the

[1]For example, for $c = 7$, the goal is to have two tables, one holding approximately 4/7 of the entries and the other 3/7. Thus the ratio between the tables is 4:3, while the basic PBD scheme aims to achieve a ratio 1:1 between the tables.

largest resulting subtable is of size $N-c+1$, while the optimal decomposition would have resulted in c tables of size N/c.

Theorem 2. *PBD may result in a decomposition whose largest subtable is asymptotically c times larger than the largest subtable in the optimal decomposition.*

Proof: Our counterexample is based on rules that do not contain `*` bits at all. Since there is no dependency between the rules (each rule matches exactly one key), any partitioning of the rules into c sets of size N/c is a valid decomposition.

On the other hand, consider the set of rules $\{\varphi_i = 1^i 0 1^{W-1-i} \mid i \in \{0,\ldots,W-1\}\}$, where W is the width of the table. Namely, φ_i is a rule whose all bits are 1 except the i-th bit, which is set to 0. Assume our PBD chooses the bits $\{i_1,\ldots,i_{c-1}\}$ during its execution. The resulting subtables will be $c-1$ singleton tables: $\{\{\varphi_{i_1}\},\ldots,\{\varphi_{i_{c-1}}\}\}$, while the rest of the $N-c+1$ rules will reside in one table. ■

C. Cut-Based Decomposition

We now offer a second approach to decomposing the table, called *cut-based decomposition* (CBD). This decomposition is based on representing the set of rules in a *directed* dependency graph.

As illustrated in Fig. 3, which shows the dependency graph of the table in Fig. 2, the nodes in this graph represent the rules. Moreover, there is an edge from node u to node v if and only if rule u has higher priority than rule v, and there is at least one key that matches both rules. Namely, the edges of the graph represent *dependencies* between the rules. Our goal is to decompose the graph, which corresponds to the original table, into component subgraphs, which will correspond to the resulting subtables, such that there are no edges between the components. That is, no key matches rules in different components.

First, since all rules match the default rule, and it is allowed to match the default rule in all subtables, we omit the node corresponding to this default rule in the graph.

Second, we assign a weight to each edge. The weight corresponds to the cost of *breaking* this edge: an edge can be broken by changing the rules in such a way that no dependency remains between the rules, and the semantic is preserved.

Specifically, let b_i^u denote the i-th bit of node u. For any node v, define the following set of dependency bits: $C_{u,v} = \{i \mid b_i^v = * \text{ and } b_i^u \neq *\}$. The weight of the edge between node u and node v, denoted by $w(u,v)$, is $|C_{u,v}|-1$. The weight $w(u,v)$ corresponds to a possible way of resolving the dependency between u and v by adding $w(u,v)$ nodes to the graph: for each bit i in $C_{u,v}$, we can write a rule that is identical to v, except the i-th bit that is replaced by $1-b_i^u$. These rules do not have a dependency with u. In addition, each key that matches v in the original rule-set will match at least one of these rules. Note that when removing a single edge from the graph, we create a new graph: edges that touch node v in the original graph might be duplicated to the new $|C_{u,v}|-1$ nodes; the weight of these duplicated edges can only decrease.

Another operation that we also allow in this scheme is a *node expansion*, that is, given a set of t `*` bits in some rule,

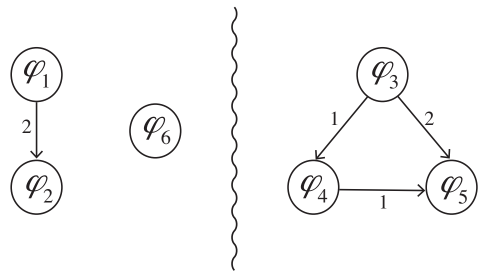


Fig. 3. The dependency graph and cut of the table in Fig. 2.

we replace the rule with 2^t new rules by replacing the `*` bits with a binary enumeration of possible combinations of 0s and 1s. By definition, this operation does not change the semantics of the original table. However, it reduces the connectivity of the dependency graph, facilitating the graph partitioning.

The next theorem shows that CBD is in fact a generalization of PBD:

Theorem 3. *Using the above-mentioned edge breaking and node expansion operations, CBD can exactly emulate PBD.*

Proof: When PBD partitions a table into two tables using some pivot bit, it duplicates the rules with a *don't-care* (`*`) value in the corresponding bit into the two resulting subtables, and replaces the corresponding `*` value by 0 and 1, respectively. This corresponds to a rule expansion in the CBD. This way, CBD can mimic the exact steps of PBD. Since at each step in PBD, no packet header matches two rules from different subtables, CBD can partition the large table into exactly the same resulting tables as the PBD. ■

We next discuss how we practically cut the graph into components. First, the problem of partitioning a graph into two equal-size components and minimizing the weights of edges among these components is known to be NP-hard [32]. Our problem is even more general, in the sense that in some cases we need to consider the sum of weights (for edges in the cut that are destined for different nodes) and in some cases we need to consider the product of weights (for edges that are destined for the same nodes). Moreover, we may want to first expand some of the rules.

In practice, we propose a greedy algorithm that solves this problem iteratively. At each iteration, we first try to partition the dependency graph into c equal-sized components, and minimize the weights of edges among these components. For this task, we use METIS [32], a tool to approximately partition graphs. Then, given the resulting partitioning, we evaluate it, and decide whether to expand one of the rules, and then go to the next iteration, or to finish by breaking all cut-edges. For instance, the wavy line in Fig. 3 depicts the cut of the dependency graph. After adding a default rule to each subtable, it results into two subtables of size 4 rules each (comparing to size 4 and size 5 in Example 1).

The decision whether to expand one of the rules or to finish by breaking all cut-edges depends on the quality of the partitioning. Namely, in case that the total weight of the edges in the cut exceeds some parameter w_0, we look at

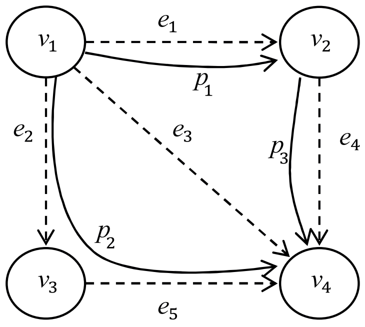


Fig. 4. Illustration of the model. Network $G = \langle V, E\rangle$ has vertex set $V = \{v_1, \ldots, v_4\}$ for the switches, and an edge set $E = \{e_1, \ldots, e_5\}$ for the links. There are three paths in the path set $P = \{p_1, p_2, p_3\}$, and, for example, $S(p_1) = \{v_1, v_2\}$ and $L(p_1) = \{e_1\}$. Finally, $G|_P$ is the same as G but without the link e_3, which does not belong to any of the paths.

the destinations of the edges in the cut, and pick a vertex whose sum of incoming cut-edges' weight is the largest. Then, we expand the rule corresponding to this vertex, thus eliminating all dependencies. Furthermore, since METIS does not necessarily return a perfectly balanced partition, we look at the ratio between the size of the largest component and the target size (that is, if a perfectly balanced partition would have been produced). If this ratio exceeds a certain parameter r_0 (namely, the partition is poorly balanced), we will try to break the largest component by expanding a rule within that component. Naturally, a good candidate for such an expansion is a rule that has few * bits (e.g., up to three) and many intra-component incoming edges (e.g., the one with the largest number of incoming edges).

Theorem 4. *The CBD algorithm stops.*

Proof: At each iteration, one rule is expanded using at least one bit, that is, in the resulting table there is at least one * less. A table with not even one * has a corresponding dependency graph with no edges. Therefore, it can be easily partitioned into c equally-sized partitions with cut edge of weight 0. ■

IV. The Rainbow Path Coloring Problem

After showing how to decompose the initial table into subtables, we now turn to show *how to spread the subtables in the network.*

We model the network as a directed graph $G = \langle V, E\rangle$ with a vertex set V and an edge set E, where $V = \{v_1, \ldots, v_n\}$ represents the set of n switches and $E = \{e_1, \ldots, e_m\}$ the set of m links. As a first step, we consider an homogeneous network where all switches are identical, and therefore have identical constraints on the table size. We will relax this assumption in Section V.

Let $P = \{p_1, \ldots, p_f\}$ be the set of all flow paths in the network. For each flow path p_i, $S(p_i)$ denotes its set of switches and $L(p_i)$ denotes its set of links.

Finally, given a graph G and a set of paths P, we denote by $G|_P$ the *projection* of G over P, namely $G|_P = \left\langle \bigcup_{p\in P} S(p), \bigcup_{p\in P} L(p)\right\rangle$ is the subgraph of G that contains only the switches and links that belong to at least one path in P. Fig. 4 illustrates the above definitions.

Recall that we aim to maintain full coverage of the original table semantics when dividing the work among switches. Informally, we would want to *color* each switch in one of c possible colors (or with no color), subject to the constraint that each path must contain all c colors. This would help us to divide the table among switches, so that each switch would need approximately $\frac{1}{c}$ of the entire table. Formally, the problem is defined as follows.

Definition 1. *Given a network $G = \langle V, E\rangle$, a flow path set P, and a number of colors c, the $\langle G, P, c\rangle$* RAINBOW PATH PROBLEM *is to decide whether there exists an assignment $\gamma : V \to \{\perp, 1, \ldots, c\}$, where $\perp$ corresponds to* no color, *such that each path $p \in P$ has at least one node of each color.*[2]

While the $\langle G, P, c\rangle$ RAINBOW PATH PROBLEM is defined as a *decision problem*, in practice, *our goal is to maximize c*. Notice that the number of colors c is clearly at most the length of the shortest path in P, which is in turn bounded by n. Thus, the maximum number of colors can be obtained by applying the decision problem with any number of colors c up to the shortest relevant path size.

In the appendix, we show that for general graphs, the decision problem is *NP-hard* even for two colors. We now obtain efficient solutions for special graph topologies, as well as heuristics for the general case.

A. The Rainbow Path Coloring in Trees

Nowadays, OpenFlow-equipped networks are often deployed in data centers, where the network itself has a regular structure. A prime example of such a topology is a *tree*: a fully connected graph that has no cycles. [3]

In some cases, there is also a restriction on the relevant paths that should be considered; e.g., assuming all paths originate from or are destined to a single node.

In this section, we first tackle these single-source (equivalently, single-sink) trees. Our results are slightly more general, as they require that only the projection of the paths on the original topology is a tree. For example, if all paths are the shortest possible, a single-source (single-sink) setting always forms a tree (a.k.a. the shortest-paths tree). We show a simple valid coloring of size $s+1$, where s is the shortest path size. In this coloring, the color is simply given by the distance to the single ingress (or egress) switch.

Theorem 5. *Given a network G, and a flow path set P such that all paths originate from or are destined to a single node and follow the shortest-path scheme, then there is a valid color assignment γ with $s+1$ colors to the $\langle G, P, c\rangle$* RAINBOW PATH PROBLEM, *where s is the shortest path size in P. Furthermore, γ can be computed in $O(m+n)$ time.*

[2]This last condition on the paths can be formally written as follows. Let $\overline{\gamma} : 2^V \to 2^{\{\perp,1,\ldots,c\}}$ denote the extension of γ to a set of nodes: $\overline{\gamma}(V') = \{c \mid v \in V', \gamma(v) = c\}$. A valid assignment γ implies that for all paths $p \in P$, $\{1, \ldots, c\} \subseteq \overline{\gamma}(S(p))$.

[3]Some datacenters have a *fat-tree* topology. Extending our results to fat-tree is part of our future research.

Proof: Without loss of generality, assume that all paths in P originate from a single source x, and denote by $d(x,y)$ the distance in edges between node x and some node y. Let γ be the following assignment function: for each $y \in \bigcup_{p\in P} S(p)$,

$$\gamma(y) = \begin{cases} d(x,y)+1 & d(x,y) \le s \\ \perp & \text{otherwise} \end{cases}$$

where $\perp$ indicates that no color is given to the node, and therefore, no table should be installed at that node. Notice that γ can be computed in $O(m+n)$ time using the Breadth-first search (BFS) algorithm [33] originating from x.

Consider a path $p \in P$, which starts at x and follows a shortest-path scheme. Thus, $S(p)$ contains nodes of increasing distances until the length of p, which is at least s. Thus, $\overline{\gamma}(S(p)) = \{1,\ldots,s+1\}$, and the claim follows. ∎

Next, we deal with the more general case in which $G|_P$ is a tree. In such a case, we show a valid coloring with $\lceil \frac{s}{2}+1\rceil$ colors, where s is the shortest path size in P.

Theorem 6. *Given a network G and a flow path set P with a shortest path of length s, if $G|_P$ is a tree, then there is a valid color assignment with $\lceil \frac{s}{2}+1\rceil$ colors to the $\langle G,P,c\rangle$* RAINBOW PATH PROBLEM. *It can be computed in $O(m+n)$ time.*

Proof: Let $c = \lceil \frac{s}{2}+1\rceil$, and pick an arbitrary node $x \in S(P)$. Consider the following color assignment γ:

$$\gamma(y) = (d(x,y)-1 \mod c)+1.$$

Note that, by definition, the values of γ are in $\{1,\ldots,c\}$. γ can be computed using a BFS from node x in $O(n+m)$ time.

We next show that each path contains all the colors. Consider a path $p \in P$, and let $y \in S(p)$ be the node with minimal distance to x. Since $G|_P$ is a tree, for each other node $y' \in S(p)$, $d(y',x) = d(x,y) + d(y,y')$ (otherwise, there is a path between y' and x that does not go through y, implying that there is a cycle in $G|_P$). Let p_1, p_2 be the division of path p into two paths: p_1 starts in the first node of path p and ends in y, p_2 starts in y and ends in the last node of path p. Without loss of generality, assume that p_1 is longer than p_2; thus the length of p_1 is at least $\lceil \frac{s}{2}\rceil$ edges, implying that $|S(p_1)| \ge \lceil \frac{s}{2}+1\rceil$. Furthermore, since there is only a single simple path between each two nodes in the tree, the set of distances between nodes in $S(p_1)$ and y is $\{0,\ldots,|S(p_1)|\}$, immediately implying that $\overline{\gamma}(S(p_1)) = \{(i+d(x,y)-1 \mod c)+1 \mid i \in \{0,\ldots,|S(p_1)|\} = \{1,\ldots,c\}$, and the claim follows. ∎

Another special case where all paths are of length 2 is considered in the appendix.

B. The Rainbow Path Coloring in General Graphs

Since the $\langle G,P,c\rangle$ RAINBOW PATH PROBLEM is not tractable in general graphs, we present a greedy heuristic that might yield a suboptimal solution. Yet, our simulations show that in practice the margin of error is on average within 2% (see Section VI).

More specifically, our GREEDY algorithm (see Algorithm 1 for pseudo-code) works in iterations. At each iteration, which corresponds to a new color, GREEDY continuously picks uncolored nodes one by one, until each path contains at least one of the picked nodes in this iteration. In such a case, the nodes picked are colored with a new color, and the algorithm continues to the next iteration. If at some iteration, even after picking all uncolored nodes, there is at least one path that does not contain any of the picked nodes, then those nodes remain uncolored ($\perp$), and the algorithm stops. Note that, in any case, the algorithm never stops in the first iteration, that is, it always succeeds to color the nodes using at least one color.

Algorithm 1: Pseudo-code for the q-GREEDY algorithm

```
Input: node set $V$, path set $P$
Output: a valid color assignment $\gamma$, number of colors $c$
1  $c = 0$ ;
2  set $\gamma(v) = \perp$ for all $v \in V$;
3  while $V \neq \emptyset$ do
4      $P' = P$ ; $V' = \emptyset$ ;
5      $Search$ = TRUE ;
6      while $Search$ do
7          $V^0 = \text{argmax}_{\tilde{V} \subseteq V', |\tilde{V}| \le q}$
           $\left\{\left|\left\{p|\tilde{V} \cap S(p) \neq \emptyset, p \in P'\right\}\right|\right\}$;
8          $P^0 = \left\{p|\tilde{V} \cap S(p) \neq \emptyset, p \in P'\right\}$ ;
9          $V = V \setminus V^0$ ; $V' = V' \cup \{V^0\}$ ; $P' = P' \setminus P^0$ ;
10         $Search = (P' \neq \emptyset)$ and $(V^0 \neq \emptyset)$ ;
11     if $P' = \emptyset$ then
12         $c = c+1$ ;
13         set $\gamma(v) = c$ for all $v \in V'$ ;
14         $V = V \setminus V'$ ;
15     else
16         $V = \emptyset$;
```

We next present two variants of this algorithm, which differ in the way the nodes are selected at each iteration. In the first variant, which we call 1-GREEDY, at each choice, we pick the node v that maximizes the number of paths that contain v but do not contain the new color. The following theorem captures the time complexity of this algorithm:

Theorem 7. 1-GREEDY *runs in $O\left(n^2 \cdot f\right)$ time complexity, where n is the number of nodes and f is the number of paths.*

Proof: When we pick a node to color, we first need to consider all other remaining nodes to ensure that this node is the one which belongs to the largest number of paths; each such comparison takes at most f steps (counting all the paths). Hence, the total number of steps required to choose all n nodes, is $f \cdot ((n-1)+(n-2)+\ldots+1) = O(n^2 \cdot f)$. ∎

The second variant, q-GREEDY, generalizes the 1-GREEDY algorithm by considering, at each step, a set of up to q nodes (instead of a single node). It chooses the set of nodes that maximizes the number of paths for which there is at least one node in the set.

The next example shows an execution of 1-GREEDY and an execution of 2-GREEDY that differ in their outcome. This demonstrates the tradeoffs in fixing the parameter q.

Example 2. *Consider the example in Fig. 4. We first run* 1-GREEDY*. At the first iteration, nodes v_1, v_2, and v_3 belong to two paths; assume that* 1-GREEDY *first picks v_1 and colors it in the first color. Then, in order to color p_3, it picks v_2 and colors it in the first color as well. Note that all nodes of the first path p_1 are now colored, implying that any additional iteration will fail, resulting in a valid coloring of only one color.*

*In contrast, q-*GREEDY *with $q = 2$ first picks nodes v_2 and v_3, since all three paths traverse through either one of these nodes. Then, v_1 and v_4 can be colored with an additional color, resulting in a valid coloring with 2 colors.*

The following theorem is a simple generalization of Theorem 7:

Theorem 8. *q-*GREEDY *runs in $O\left(n^{q+1} \cdot f\right)$ time complexity, where n is the switch set size and f is the path set size.*

C. An Optimal Solver

Although the $\langle G, P, c\rangle$ RAINBOW PATH PROBLEM is NP-hard, it is possible to compute the optimal solution for small instances of the problem. In this section, we present an algorithm, based on dynamic-programming, that practically solves the $\langle G, P, c\rangle$ RAINBOW PATH PROBLEM with projected graphs of up to 17 nodes. Note that each (isomorphic) coloring corresponds to a specific partition of a set of n labeled elements [34]. The number of possible partitions of n elements is the n-th Bell number, whose value for $n = 16$ is approximately 10^{10}. Thus, a naïve approach of testing all possibilities is intractable in a reasonable time. The main motivation is two-folded. First, it may be satisfactory for some real-life instances of the $\langle G, P, c\rangle$ RAINBOW PATH PROBLEM. Second, this algorithm is used in Section VI as a baseline for evaluating the performance of the 1-GREEDY and q-GREEDY.

Our algorithm works in two phases. In the first phase, we find a set S of all subsets of the switch set that have the following property: Given a subset $s \in S$, coloring all switches in s with a specific color results in coloring all paths in the path set. We also make sure that for all $s \in S$, the subset s is *minimal* in the sense that there is no subset of s that has the same property.

To find the set S, we start with a set of n singletons. Then, at each step, for each of the subsets so far, we try to add a new switch with index larger than the largest switch index of this subset. If adding a new switch makes the number of paths that are colored larger, then the new subset is saved for the next iteration, otherwise it is dismissed. Also, if the new subset covers all paths, then it is added to S.

Next, after having computed the set S, we go to the second phase of our algorithm, where we find the maximum size set of disjoint subsets in S, whose size corresponds to the optimal number of colors that can be used in the original $\langle G, P, c\rangle$ RAINBOW PATH PROBLEM. We note that the second phase is essentially an instance of the *maximum set packing* problem, and can also be solved using a dynamic programming technique.

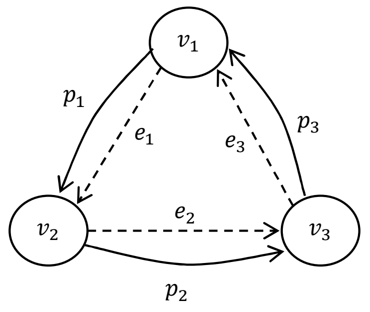


Fig. 5. A network with no valid coloring to the $\langle G, P, c\rangle$ RAINBOW PATH PROBLEM with $c = 2$

V. MULTICOLORED SWITCHES

Up until now, we have considered the $\langle G, P, c\rangle$ RAINBOW PATH PROBLEM where a *single* color is assigned to each switch. However, in practice, *we may want to assign more than one color to each switch*.

The motivation for such an assignment is three-folded.

First, this would facilitate the implementation of a *heterogeneous software-defined network*, in which nodes with a larger-capacity table can be assigned multiple colors.

Second, there is additional degree of freedom in the problem, implying that there are more feasible colors assignments. To see this, consider Fig. 5, which shows a ring graph of size 3 in which each path consists of a single edge. The only valid solution of the $\langle G, P, c\rangle$ RAINBOW PATH PROBLEM is when all switches are colored with the same color. However, the graph can be colored in three colors, when each switch is allowed to be colored with two colors (namely, $\gamma(v_1) = \{1, 2\}\,; \gamma(v_2) = \{2, 3\}\,; \gamma(v_3) = \{3, 1\}$). Specifically, we are interested in the ratio of the number of colors in each switch to the total number of colors. In this example, the ratio is $\frac{2}{3}$, implying that each switch should hold approximately two thirds of the table (as opposed to the entire table in the $\langle G, P, c\rangle$ RAINBOW PATH PROBLEM solution).

The third motivation for assigning multiple colors to a node is in cases where the graph G contains only *few very short paths*, while other paths are relatively long. This is because in the $\langle G, P, c\rangle$ RAINBOW PATH PROBLEM, the length of the shortest path is an upper bound on the number of colors that can be used. Therefore, it also determines a lower bound on the size of the table stored in the switches. This implies that switches on longer paths, which can potentially share their table with more switches, will still need to store a large table. In that case, it is appealing to assign multiple colors to switches on shorter paths (resulting in relatively large tables), while the rest of the switches are assigned only with few colors, which correspond to smaller tables.

We now formally define the multicolored problem.

Definition 2. *Given a network $G = \langle V, E\rangle$, a set of paths P, a number of colors c, and for each node v_i a maximum number d_i of colors that it can accept, the $\langle G, P, c, d\rangle$* RAINBOW PATH PROBLEM *is to decide whether there exists an assignment $\gamma : V \to 2^{\{1,\dots,c\}}$, such that each path $p \in P$ has at least one* *node of each color, and no node v_i has more than d_i colors.*[4]

Note that the fraction of the original table that is stored in node v_i is approximately $|\gamma(v_i)|/c$, assuming that the table decomposition algorithms (Section III) are efficient.

Clearly, as the $\langle G, P, c\rangle$ RAINBOW PATH PROBLEM is NP-hard, so is the $\langle G, P, c, d\rangle$ RAINBOW PATH PROBLEM. However, we can reuse our results on the $\langle G, P, c\rangle$ RAINBOW PATH PROBLEM using the following reduction: Given the instance of the $\langle G, P, c, d\rangle$ RAINBOW PATH PROBLEM, we split each node v_i into a chain of d_i nodes, and make each path that goes through v_i to go through the whole corresponding chain. Then, we get an instance of the $\langle G, P, c\rangle$ RAINBOW PATH PROBLEM that corresponds to the original $\langle G, P, c, d\rangle$ RAINBOW PATH PROBLEM. Note that in the $\langle G, P, c\rangle$ RAINBOW PATH PROBLEM, it is possible that a node remains uncolored. This corresponds to the case where, in the original problem, the number of colors assigned to some node v_i is lower than d_i.

VI. EXPERIMENTAL RESULTS

We now turn to evaluating our algorithms. We first check the decomposition algorithms (Section III) and then the table distribution algorithms (Sections IV–V).

A. Table Decomposition

We first consider the PBD and CBD algorithms for decomposing tables, as presented in Section III.

We define the *quality* of a table decomposition algorithm as the ratio between the number of rules in the original table, and the product of the largest resulting subtable size by the number of subtables c. The quality is therefore between 0 and 1, where higher quality values implies a better decomposition. Specifically, a quality of 1 means that the largest subtable size has exactly an ideal fraction $1/c$ of the number of the original rules. Note that this quality can only be used to compare among different algorithms for the same value of c. Furthermore, it is most likely that the quality is decreased when c is increased.

We compare PBD and CBD algorithms with a *bit groups* algorithm based on [6]. Through an exhaustive search, this algorithm selects the $\log_2 c$ pivot bits that maximize the quality. Thus, it only works for values of c that are powers of 2.

Fig. 6 shows the quality of the three algorithms as the number c of partitions grows. For the simulations, we have created 100 random logically-minimized rule-sets with 12 bits and 30 rules each. PBD slightly outperforms *bit groups*, except when $c = 2$, where they perform similarly. CBD clearly outperforms both over the entire range.

We also evaluated PBD and CBD with the twelve standard classification benchmark rule-sets of ClassBench [28], [35].

When the dependency graph of CBD is relatively sparse, PBD and CBD usually display a quality between 0.7 to 0.99 for various values of c. Note that for c that is not a power

[4]Formally, in this case, $\overline{\gamma}(V') = \{c \mid v \in V', c \in \gamma(v)\}$. A valid assignment γ implies that for all paths $p \in P$, $\{1, \ldots, c\} \subseteq \overline{\gamma}(S(p))$ and for each $v_i \in V$, $|\gamma(v_i)| \leq d_i$.

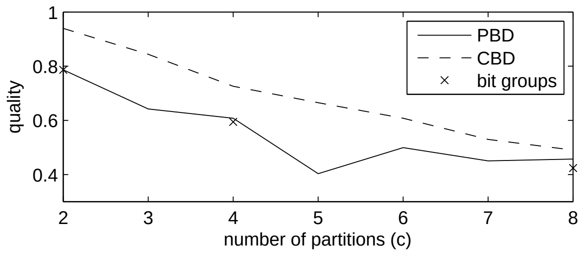


Fig. 6. Evaluation of the quality metric of the PBD, CBD, and a non-iterative algorithm that selects all pivot bits at once [5], [6]. The input is a synthetic rule set.

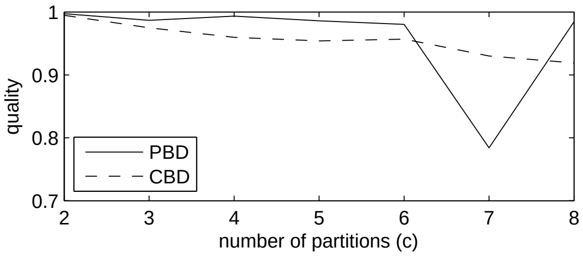


Fig. 7. Quality of partitioning of the PBD and CBD using a benchmark rule set

of 2, the quality of PBD may drop significantly, while CBD remains stable. Such a case is shown in Fig. 7.

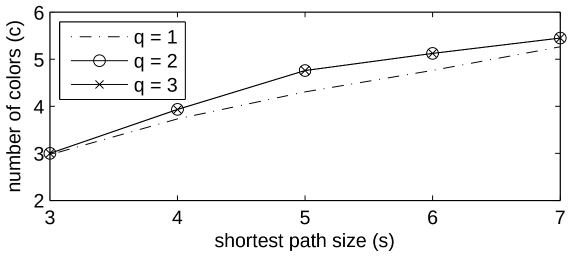
However, when the dependency graph of CBD is dense, CBD may show poor results. This is due to an instability of METIS; namely, expanding one rule in the input may result in significantly worse partitioning (where we would expect to have a partitioning at least as good as before the rule expansion). In future, we aim to overcome this problem by implementing our own graph partitioning mechanism.

B. Table Distribution

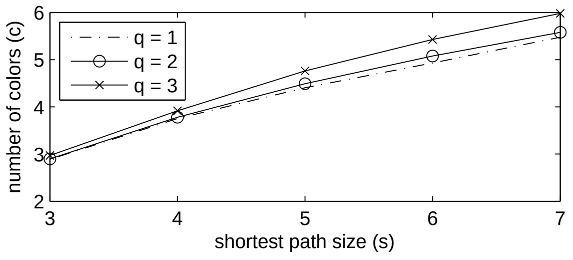
In this section, we evaluate the greedy algorithms for the single-color case, as introduced in Section IV.

To analyze their performance, we produce random instances of the $\langle G, P, c\rangle$ RAINBOW PATH PROBLEM in the following manner: given a number of switches n and a number of paths f, we add each switch to each path with probability p_n independently of the other switches or paths. Note that for a given path, the actual order of switches within the path and how they are connected to each other (namely, the exact network topology) is irrelevant to the $\langle G, P, c\rangle$ RAINBOW PATH PROBLEM. The number of switches in each path of our instances follows a Binomial distribution with parameters n and p_n. Also note that the length of the shortest path is an upper bound on the size of a valid coloring. We note that these random instances assume, among other things, independence between paths, which is not the case in real-life networks. Our future research includes evaluating the networks under real-life network topologies.

Fig. 8 shows the average size of the valid coloring obtained by our 1-GREEDY, 2-GREEDY, and 3-GREEDY algorithms, as a function of the shortest path in the random instance of the problem. We ran 10,000 random instances of the problem.

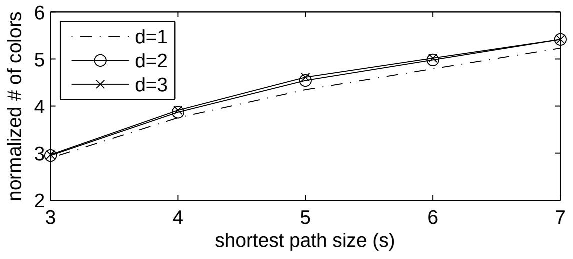


(a) $n = 10$, $f = 10$, $p_n = 0.8$

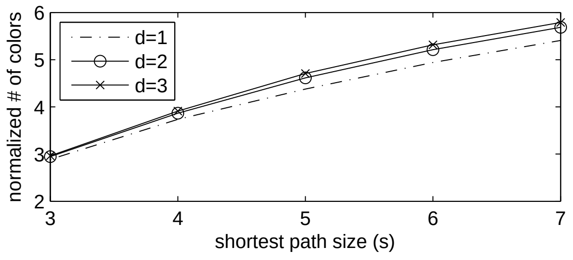


(b) $n = 20$, $f = 10$, $p_n = 0.4$

Fig. 8. Evaluation of 1-GREEDY, 2-GREEDY, and 3-GREEDY, for various values of n, f, and P_n.



(a) $n = 10$, $f = 10$, $p_n = 0.8$



(b) $n = 20$, $f = 10$, $p_n = 0.4$

Fig. 10. Evaluation of 1-GREEDY, 2-GREEDY, and 3-GREEDY for the multicolored switch case, with various values of n, f, and P_n.

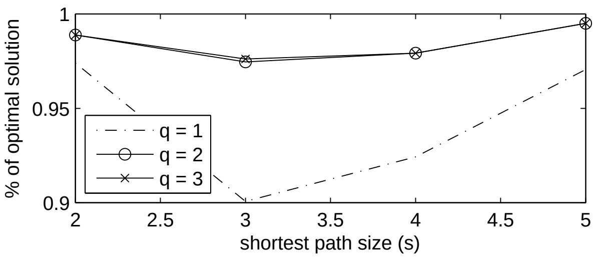


Fig. 9. Fraction of the optimal valid coloring size that can be achieved by the 1-GREEDY, 2-GREEDY, and 3-GREEDY algorithms. Parameters are $n = 7$, $f = 7$ and $p_n = \frac{5}{7}$.

Fig. 8(a) illustrates the average size of the valid coloring size obtained by our algorithms for $n = 10$, $f = 10$, and $p_n = 0.8$, while Fig. 8(b) shows it for $n = 20$, $f = 10$, and $p_n = 0.4$. Note that for a shortest path size 5 and $n = 10$ switches, the maximum valid coloring size is 5, and indeed, our algorithms achieve on average a valid coloring of size 4.3043, 4.7572 and 4.7573 for $q = 1$, $q = 2$, and $q = 3$, respectively. Clearly, a larger value of q results in a larger number of colors, on average.

We further study the our algorithms on smaller networks, where we are able to compute the *optimal* valid coloring size. Fig. 9 shows the number of colors found by the greedy approach in terms of percentage of the optimal solution. The parameters in this case are $n = 7$, $f = 7$ and $p_n = \frac{5}{7}$, and we ran the simulation 1000 times. Our results yield that, in these cases, that the greedy approach finds a valid coloring whose size exceeds (on average) 98% of the optimal solution.

C. Greedy Approach Evaluation - Multiple Color case

In this section we evaluate our solution to the $\langle G, P, c, d\rangle$ RAINBOW PATH PROBLEM. To compare the algorithm performance with various values of d, we normalize the valid coloring size found by d. This corresponds to the *share* of the work-load that each switch gets in the worst case.

Fig. 10 shows the *normalized* valid coloring size as a function of the shortest path in random instances of the problem. We ran 10,000 random instances of the problem. In Fig. 10(a) the parameters are $n = 10$, $f = 10$ and $p_n = 0.8$, while in Fig. 10(b) they are $n = 20$, $f = 10$ and $p_n = 0.4$.

The simulation results show that as d increases, the normalized number of colors also increases. For example, for a shortest path size $s = 5$ and $n = 10$ switches, a normalized valid coloring of size 4.3457, 4.5429 and 4.6101 was achieved for $d = 1$, $d = 2$ and $d = 3$, respectively.

VII. CONCLUSION

This paper proposed Palette, a framework to decompose and distribute SDN tables across the network. Palette is especially important as switch table sizes can become a bottleneck in scaling SDNs. Moreover, it facilitates handling the heterogeneity of switches in the network and the changes of equipment.

We modeled the problem in a graph-theoretic manner, and proposed several algorithms, both for decomposing one table to semantically-equivalent subtables and for spreading these subtables across the network. Our algorithms were evaluated both under random and real-life instances.

As future work, we now plan to implement Palette over OpenFlow controllers, thus providing an automatic tool to decompose and distribute SDN tables. We note that a major challenge in OpenFlow implementations is OpenFlow's restrictions on the structure of patterns in the table (in OpenFlow 1.0, two fields are allowed to be prefixes, and the other fields can be either exact or entirely `*`-bits [1]); this will require adaptation of our decomposition algorithm accordingly. Finally, we plan to extend our optimal coloring algorithms to handle additional network topologies (e.g., fat trees, which are common in contemporary datacenters).

Acknowledgment

The work was partly supported by the European Research Council Starting Grants n°210389 and n°259085, the Alon Fellowship, the ATS-WD Career Development Chair, the Loewengart Research Fund, the Intel ICRI-CI Center, the Israel Ministry of Science and Technology, and the Israeli Centers of Research Excellence (I-CORE) program (Center No. 4/11). We would like to thank Jennifer Rexford for her useful comments on an earlier version of this paper, and Anat Bremler-Barr for kindly accepting to run several simulations.

References

[1] "Openflow switch specification." [Online]. Available: http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.

[3] "NEC Univerge PF5240 and PF5820." [Online]. Available: http://www.openflow.org/wp/switch-nec/

[4] V. Sekar, M. K. Reiter, W. Willinger, H. Zhang, R. R. Kompella, and D. G. Andersen, "cSamp: A system for network-wide flow monitoring," in *USENIX NSDI*, 2008.

[5] K. Zheng, C. Hu, H. Liu, and B. Liu, "An ultra high throughput and power efficient TCAM-based ip lookup engine," in *IEEE Infocom*, 2004.

[6] K. Zheng, H. Che, Z. Wang, B. Liu, and X. Zhang, "Dppc-re: TCAM-based distributed parallel packet classification with range encoding," *IEEE Trans. Comput.*, vol. 55, pp. 947–961, 2006.

[7] S. Shenker, "The future of networking, and the past of protocols," in *Open Networking Summit*, 2011.

[8] "Google G-scale network." [Online]. Available: http://www.eetimes.com/electronics-news/4371179/Google-describes-its-OpenFlow-network

[9] "HP Procurve Switch." [Online]. Available: http://www.openflow.org/wp/switch-hp/

[10] "Interop 2012 openflow roundup," 2012. [Online]. Available: http://www.openflowhub.org/blog/blog/2012/05/10/interop2012/

[11] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *SIGPLAN ICFP*, 2011.

[12] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *ACM Sigcomm*, 2012, pp. 323–334.

[13] N. Kang, J. Reich, J. Rexford, and D. Walker, "Policy transformation in software defined networks," in *ACM Sigcomm*, 2012, pp. 309–310.

[14] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with difane," *ACM Comput. Commun. Rev.*, vol. 41, no. 4, 2010.

[15] Q. Dong, S. Banerjee, J. Wang, and D. Agrawal, "Wire speed packet classification without TCAMs: A few more registers (and a bit of logic) are enough," in *SIGMETRICS*, 2007.

[16] M. Moshref, M. Yu, A. Sharma, and R. Govindan, "vcrib: Virtualized rule management in the cloud," in *USENIX HotClouds*, 2012.

[17] Y. Zhang, M. Roughan, N. Duffield, and A. Greenberg, "Fast accurate computation of large-scale ip traffic matrices from link loads," in *SIGMETRICS*, 2003.

[18] A. Lakhina, M. Crovella, and C. Diot, "Diagnosing network-wide traffic anomalies," in *ACM Comput. Commun. Rev.*, vol. 34, 2004, pp. 219–230.

[19] X. Li, F. Bian, H. Zhang, C. Diot, R. Govindan, W. Hong, and G. Iannaccone, "Mind: A distributed multi-dimensional indexing system for network diagnosis," in *IEEE Infocom*, 2006.

[20] M. S. Kodialam and T. V. Lakshman, "Detecting network intrusions via sampling: A game theoretic approach," in *IEEE Infocom*, 2003.

[21] C. Chaudet, E. Fleury, I. G. Lassous, H. Rivano, and M.-E. Voge, "Optimal positioning of active and passive monitoring devices," in *CoNEXT*, 2005.

[22] K. Suh, Y. Guo, J. Kurose, and D. Towsley, "Locating network monitors: complexity, heuristics, and coverage," in *IEEE Infocom*, 2005.

[23] V. Sekar, A. Gupta, M. Reiter, and H. Zhang, "Coordinated sampling sans origin-destination identifiers: Algorithms and analysis," in *COMSNETS*, 2010.

[24] S. Raza, G. Huang, C.-N. Chuah, S. Seetharaman, and J. Singh, "Measurouting: A framework for routing assisted traffic monitoring," in *IEEE Infocom*, 2010.

[25] A. di Pietro, F. Huici, D. Costantini, and S. Niccolini, "Decon: Decentralized coordination for large-scale flow monitoring," in *IEEE Infocom, Work In Progress Session*, 2010.

[26] M. Sharma and J. Byers, "Scalable coordination techniques for distributed network monitoring," in *PAM*, 2005.

[27] R. Sherwood, "A crash course in OpenFlow 1.1." Big Switch Internal Tech Talk Series, Aug. 2011.

[28] O. Rottenstreich, R. Cohen, D. Raz, and I. Keslassy, "Exact worst-case TCAM rule expansion," *IEEE Trans. Comput.*, 2012.

[29] A. X. Liu, C. R. Meiners, and E. Torng, "TCAM razor: a systematic approach towards minimizing packet classifiers in TCAMs," *IEEE/ACM Trans. Networking*, vol. 18, no. 2, pp. 490–500, 2010.

[30] A. Bremler-Barr and D. Hendler, "Space-efficient TCAM-based classification using gray coding," in *IEEE Infocom*, 2007.

[31] C. Meiners, A. Liu, and E. Torng, "Topological transformation approaches to TCAM-based packet classification," *IEEE/ACM Trans. Networking*, vol. 19, no. 1, pp. 237–250, Feb. 2011.

[32] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1999.

[33] D. Jungnickel, *Graphs, Networks and Algorithms*, 3rd ed. Springer Publishing Company, Incorporated, 2007.

[34] D. Berend and T. Tassa, "Improved bounds on bell numbers and on moments of sums of random variables," *Probability and Mathematical Statistics*, vol. 30, no. 2, pp. 185–205, 2010.

[35] D. E. Taylor and J. S. Turner, "Classbench: a packet classification benchmark," in *IEEE Infocom*, 2005.

[36] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990.

Appendix A
NP-hardness Proof

In this section we show that the $\langle G, P, c\rangle$ RAINBOW PATH PROBLEM is NP-hard in the general case, even for $c = 2$ colors. The proof is based on reducing the 3-SAT problem [36] to the $\langle G, P, c\rangle$ RAINBOW PATH PROBLEM.

Theorem 9. *Given a general network G, a path set P, and a number of colors c, the $\langle G, P, c\rangle$ RAINBOW PATH PROBLEM is NP-hard.*

Proof: Let the Boolean expression B denote an instance of the 3-SAT problem with variables $X_1, X_2, \ldots, X_n$. Thus, $B = C_1 \wedge C_2 \wedge \ldots \wedge C_k$, where $C_i = \left(X_i^1 \vee X_i^2 \vee X_i^3\right)$. We have to construct in polynomial time an instance of the $\langle G, P, c\rangle$ RAINBOW PATH PROBLEM so that it would have a proper coloring if and only if the Boolean expression B is satisfiable. In practice, our proof needs two distinct constructions, although with a slight variation. One construction holds for the case where there exists an assignment A such that B is satisfiable with $A(X_1) = 1$, while the other construction holds for the case where B is satisfiable with $A(X_1) = 0$. Since there is only a slight variation between the two constructions, we focus on the first case. An explanation on how to apply the proof with the slight variation is found in the end of this proof.

For each variable X_i, we allocate two distinct switches v_i and v_i', where v_i represents X_i and v_i' represents $\neg X_i$. Furthermore, let $M\left(X_i^j\right)$ be the switch representing j-th literal in the i-th clause. Finally, we add an additional baseline switch, denoted V^a. So, that the constructed switch set V is

$$V = \left(\bigcup_{i=1}^{n} \{v_i, v_i'\} \right) \bigcup \{v^a\} .$$

We now skip to the construction of the path set P. First, for each variable X_i we construct the path $\langle v_i, v_i' \rangle$. There are n such paths in total. Then, for each clause C_i, we construct the path $\left\langle v^a, M\left(X_i^1\right), M\left(X_i^2\right), M\left(X_i^3\right)\right\rangle$. For example, for the clause $(X_2 \vee \neg X_4 \vee \neg X_5)$ we construct the path $\langle v^a, v_2, v_4', v_5' \rangle$. Finally, we add the path $\langle v^a, v_1 \rangle$ to the path set. So the constructed path set P is

$$P = \left(\bigcup_{i=1}^{n} \{\langle v_i, v_i' \rangle\} \right) \bigcup \left(\bigcup_{i=1}^{k} \left\{ \left\langle v^a, M\left(X_i^1\right), M\left(X_i^2\right), M\left(X_i^3\right) \right\rangle \right\} \right) \bigcup \{\langle v^a, v_1 \rangle\} .$$

The construction of the link set E is easily deduced from the construction of the path set P, where there is an edge between two vertices if and only if this edge is within one of the paths.

Overall, we construct $2 \cdot n + 1$ switches and $k + n + 1$ paths. Therefore, the reduction is clearly polynomial.

Let $c = 2$, and $G = \langle V, E \rangle$. The Boolean expression B is satisfiable with $X_1 = 1$ if and only if there is a proper coloring that solves the constructed instance of the $\langle G, P, c \rangle$ RAINBOW PATH PROBLEM.

Given an assignment A (with $A(X_1) = 1$) that satisfies B, we color the switches in V as follows ($i \in \{1, \ldots, n\}$): If $A(X_i) = 1$ then $\gamma(v_i) = 2$ and $\gamma(v_i') = 1$, while in the case where $A(X_i) = 0$, $\gamma(v_i) = 1$ and $\gamma(v_i') = 2$. $\gamma(v^a)$ is set to 1. This implies that for all $i \in \{1, \ldots, n\}$, $\gamma(v_i) \neq \gamma(v_i')$, and thus there are 2 distinct colors in the first n paths in P. Since A satisfies B, then for all $i \in \{1, \ldots, k\}$ it satisfies the clause C_i. It implies that at least one of the literal in C_i equals 1. Therefore, the corresponding path in P has the color 2 in addition to $v^a = 1$. The last path, that is, $\langle v^a, v_1 \rangle$, is easy to verify.

On the other hand, assume an assignment of colors γ to the switches in V that satisfy the condition of c colors per path. Without loss of generality, due to the last path in P, we assume that $\gamma(v^a) = 1$, and $\gamma(v_1) = 2$. We build the following variable assignment A: $A(X_i) = \gamma(v_i) - 1$. First, we get that $A(X_1) = 1$. Second, from the definition of the construction, for all $i \in \{1, \ldots, k\}$, the clause C_i is satisfied at least by one literal (whose corresponding switch color is 2).

Finally, we may repeat the same proof with a slight variation in the last path of the path set. Instead of $\langle v^a, v_1 \rangle$, we add to the path set P the path $\{v^a, v_1'\}$. Following this variation, it is possible to prove that the Boolean expression B is satisfiable with $v_1 = 0$ if and only if there is a proper coloring that solves the constructed instance of the $\langle G, P, c \rangle$ RAINBOW PATH PROBLEM, with $c = 2$.

Having covered the two cases, we get the claimed result. ■

APPENDIX B
SPECIAL CASE: ALL PATHS ARE OF LENGTH 2

In this section we consider the case where all paths in P are of size 2. In this case, the problem is not NP-hard, and in fact can be solved in linear time. This is reflected in the following theorem.

Theorem 10. *Given a general network G, a path set P with paths of size 2, and $c = 2$, the $\langle G, P, c \rangle$* RAINBOW PATH PROBLEM *can be solved in $O(|V'| + |E'|)$ time, where V' and E' are the switch and link sets of the projection $G|_P$ of G over P).*

Proof: If all paths are of size 2, it clearly follows that there exists a valid coloring if and only if $G|_P$ is a bipartite graph. To determine whether a given graph is bipartite, we simply use the Breadth-first search (BFS) algorithm [33], whose complexity is $O(|V'| + |E'|)$. ■