New Alternatives to Optimize Policy Classifiers

Vitalii Demianiuk[1,2], Sergey Nikolenko[2], Pavel Chuprikov[1,2] and Kirill Kogan[1]
[1]IMDEA Networks Institute, Madrid, Spain
[2]Steklov Institute of Mathematics at St. Petersburg, Russia
Email: {vitalii.demianiuk, pavel.chuprikov, kirill.kogan}@imdea.org, snikolenko@gmail.com

***Abstract*—Growing expressiveness of services increases the size of a manageable state at the network data plane. A service policy is an ordered set of classification patterns (classes) with actions; the same class can appear in multiple policies. Previous studies mostly concentrated on efficient representations of a single policy instance. In this work, we study space efficiency of multiple policies, cutting down a classifier size by sharing instances of classes between policies that contain them. In this paper we identify conditions for such sharing, propose efficient algorithms and analyze them analytically. The proposed representations can be deployed transparently on existing packet processing engines. Our results are supported by extensive evaluations.**

I. Introduction

Transport networks satisfy requests to forward data in a given topology. To guarantee desired data properties during forwarding, network operators impose economic models implementing various policies such as security or quality-of-service. As network infrastructure becomes more intelligent, the complexity of these policies is constantly growing.

Unfortunately, increasing manageable state on the data plane has its limitations. Traditionally, service policies are represented by packet classifiers whose implementations are usually expensive (e.g., *ternary content-addressable* memories, or TCAMs). Most existing works optimize each policy instance separately (see Section VIII). In this work, we exploit other alternatives to achieve additional efficiency of policy state represented on the data plane. Our ideas hinge on the fact that similar "classification patterns" (classes) are reused in different policies. Various vendors already support the notion of classes in policy declarations [1], [2] allowing to abstract and manage classification patterns more efficiently. For instance, Cisco IOS supports up to 256 different QoS policies and up to 4096 classes per box [3]. In real deployments, the number of classes per policy ranges from tens to hundreds depending on the application model [3]. The size of a class depends on the complexity of represented pattern.

The work of Vitalii Demianiuk, Pavel Chuprikov, and Kirill Kogan was partially supported by a grant from the Cisco University Research Program Fund, an advised fund of Silicon Valley Community Foundation. The work of Sergey Nikolenko shown in Sections III, IV, and V (in particular, Theorems 1, 4, 5, 6) has been supported by the Russian Science Foundation grant no. 17-11-01276. The work of Kirill Kogan was also partially supported by the Region al Government of Madrid on Cloud4BigData grant S2013/ICE-2894.

https://doi.org/10.1109/ICNP.2018.00022

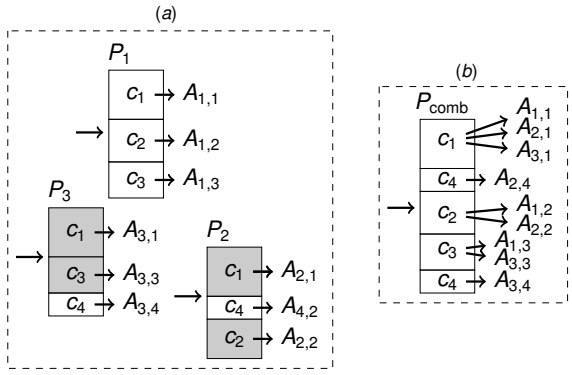


Fig. 1: (a) separate policies; (b) a representation P_{comb} that emulates the policies; class instances that have been cut in P_{comb} are shown in gray.

Traditionally, a separate class instance is allocated for each policy instance that contains it (see Fig. 1a). Since classes are used in different policies, it allows us to look at combined service policy representations, where ideally each class appears only once, providing substantial savings in representations of underlying classifiers in expensive memory such as TCAM. Usually, the complexity of structural properties of classifiers can be alleviated with additional classification lookups, but this is a shareable resource for the overall processing. The number of classification lookups per packet is one of the major constraints limiting line-rate characteristics. For instance, Cisco C12000 [4] supports at most six TCAM lookups per packet at line-rate for all services. As a result, in this work we prefer to consider combined policy representations that do not increase the number of classification lookups. Informally, proposed combined policy representations "emulate" the behaviours of represented policies.

Figure 1 illustrates major differences between the traditional attachment model, where a class instance is allocated per policy containing it, and the proposed combined representation P_{comb}. Note that P_{comb} stores a single instance of classes c_1, c_2, c_3. Class c_4 is duplicated since c_4 should be applied before c_2 in policy P_2 and after c_3 in policy P_3.

In this work we propose *semantically equivalent* combined representations for a given set of policies. We show a necessary and sufficient condition for the existence of *ideal representa-*

tions that contain only a single instance for every class of all policies and methods for constructing these representations. For the general case, we propose methods for minimizing the total number of rules in duplicated class instances. All proposed representations do not increase the number of classification lookups, that is, we say that they satisfy the *single lookup constraint*; in other words, they do not increase lookup time complexity versus lookup time in a single policy.

The paper is organized as follows. In Section III we explore necessary and sufficient conditions of ideal representations containing a single instance of every class in combined policy representations. Section IV proves that the proposed problem is intractable in the general case and shows how to deal with non-ideal representations. In Section V we propose two approximation algorithms and study them analytically for the offline case. Although the proposed algorithms can be extended for dynamic updates, in Section VI we propose a new algorithm that captures the right balance between time complexity and optimization results with dynamic updates. All proposed algorithms are evaluated in various settings in Section VII.

II. Model Description

In this section we first define the entities involved in the packet classification process and introduce our notation. A packet *header* $H = (h_1, \ldots, h_w)$ is a sequence of bits $h_i \in H$, $h_i \in \{0, 1\}$, $1 \leq i \leq w$; e.g., $(1\ 0\ 0\ 0)$ is a 4-bit header. We denote by $\mathcal{H}$ the set of all possible headers. A filter $F = (f_1, \ldots, f_w)$ is a sequence of w values corresponding to the header bits, but with possible values 0, 1, or $*$ ("don't care"). A header H *matches* a filter F if for every bit of H the corresponding bit of F has either the same value or $*$. Two filters are *disjoint* if there is no header that matches both filters.

Classes represent an intermediate level of abstraction: a *class* c is a set of filters. We denote by $w(c)$ the number of filters in c. A header H *matches* a class c if H matches at least one filter in c. Two classes (sets of filters) c and c' are *disjoint*, denoted by $c \perp c'$, if there are no headers matching both c and c' (all filters of c and c' are pairwise disjoint).

To define a *policy* P, one needs to specify a sequence $\mathbb{S}(P)$ over a set of classes $\mathcal{C}_P$ (where each class appears only once), a set of actions $\mathcal{A}_P$, and a function $\alpha_P : \mathcal{C}_P \to \mathcal{A}_P$ associating an action with every class. For an incoming packet, the action of a first matched class in $\mathcal{C}_P$ is returned. Since classes can intersect (match the same headers), a policy is defined by a sequence rather than a set. Originally, classes were intoduced to define common classification patterns [1], [2] that can significantly simplify policy management. In this way a single classification pattern should not be redefined during the declaration of another policy. Two policies P_1 and P_2 are *equivalent* if for every given header both yield the same action. Note that different sequences on the same set of classes $\mathcal{C}_P$ can lead to several equivalent policies due to possible pairwise disjointness of classes in $\mathcal{C}_p$. We denote by $\prec_P$ a partial order of classes in $\mathcal{C}_P$ corresponding to all semantically equivalent policies on $\mathcal{C}_P$. For instance, Fig. 2a defines a policy P whose corresponding partial order $\prec_P$ is illustrated on Fig. 2b.

(a)

Class	Filter	#1	#2	#3	#4	Action
c_3	F_1	0	1	$*$	$*$	A_1
	F_2	$*$	1	$*$	0	A_1
c_2	F_3	$*$	$*$	0	0	A_2
c_1	F_4	1	0	$*$	$*$	A_3
c_4	F_5	0	0	$*$	$*$	A_4

(b)

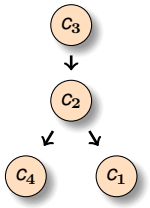


Fig. 2: (a) definition of the policy $\mathbb{S}(P) = c_3, c_2, c_1, c_4$; (b) partial order $\prec_P$: $c_3 \perp c_1, c_3 \perp c_4$, and $c_1 \perp c_4$.

We denote by $\mathcal{P} = \{P_1, P_2, \ldots P_{|\mathcal{P}|}\}$ a set of policies over the same set of classes $\mathcal{C}$; by $|\mathcal{C}|$, the number of classes in $\mathcal{C}$. Also we denote by $\mathcal{A}$ a set of actions of all policies in $\mathcal{P}$.

We say that P_{comb} *emulates* $\mathcal{P}$ if for any header H and any policy $P_i \in \mathcal{P}$, the lookup of H in P_i and lookup of H in P_{comb} in the context of P_i yield the same action. Informally, P_{comb} mimics the behaviour of $\mathcal{P}$. For simplicity in some places we call P_{comb} representation as a policy.

III. Ideal Representations

In traditional policy representations, if a single class (classification pattern) participates in multiple policies, per-policy instances of the class are allocated for every policy. Intuitively, structural properties of induced policy classifiers should have a significant impact on memory requirements. We call a combined representation of multiple policies $\mathcal{P}$ that contains only one instance of every class from $\mathcal{C}$ an *ideal* representation. For a given $\mathcal{P}$, we identify necessary and sufficient conditions for the existence of ideal representations (satisfying the single lookup constraint) and explain how to construct them. At this point we assume that policies initially given in $\mathcal{P}$ are represented by a single combined policy P_{comb}; we will reconsider this assumption in Section III-D.

A. Disjoint classes

We begin with the simplest structural property, *class disjointness*, where any two different classes in $\mathcal{C}$ (except the *default* class with the *catch-all* filter that every policy in $\mathcal{P}$ is appended with) do not match the same headers. In this case we can construct an ideal policy P_{comb} that contains all non-default classes from $\mathcal{C}$ in any order appended with a single instance of the default class. A header can be looked up in P_{comb} instead of a configured policy P_i, and if the matched class belongs to P_i (this can be verified with any set membership data structure), the corresponding actions are executed. Otherwise, actions of the *default-class* in P_i are executed.

B. Price of generalization

We have seen that class disjointness guarantees the existence of ideal representations. In Section III-C we will show that this structural property is not a necessary condition for the existence of ideal representations. To deal with more general structural

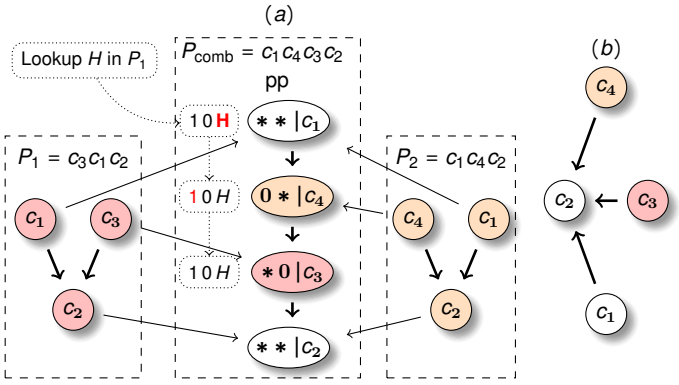


Fig. 3: (a) $\mathcal{P} = \{P_1, P_2\}$ with $\mathcal{C}$ containing non-disjoint classes, ideal P_{comb} and policy prefixes; (b) $G^{\text{jnt}}(\mathcal{P})$.

properties, where classes in $\mathcal{C}$ are not pairwise disjoint, we must guarantee that the matched class $c \in P_{\text{comb}}$ belongs to P_i and not to any other non-disjoint class with c in $\mathcal{C}$.

To implement this requirement, we prepend in the constructed P_{comb} each filter of a class c with the ternary *policy prefix* $\text{pp}(c) \in \{0, *\}^{|\mathcal{P}|}$, where $|\mathcal{P}|$ is the number of policies in $\mathcal{P}$ and $\text{pp}(c)_i = *$ if $c \in P_i$ and $\text{pp}(c)_i = 0$ otherwise. To classify a given header H in P_i, H is also prepended with a binary *header prefix* pp_i corresponding to the policy P_i, where pp_i is a bit string $0 \dots 010 \dots 0$ of length $|\mathcal{P}|$ that has a 1 only at position i. All prefixes have the same length. Note that a prefix $\text{pp}(c)$ matches pp_i if and only if $c \in P_i$. Such representations allow to match a header in P_{comb} only against the classes that belong to the original policy P_i.

Figure 3a shows a sample lookup of a header H to P_1 in P_{comb} representing $\mathcal{P} = \{P_1, P_2\}$, where $\mathcal{C}$ contains non-disjoint classes (e.g., c_1 and c_2 are not disjoint in P_1). Observe that P_{comb} with policy prefixes emulates $\mathcal{P}$ and is ideal. The values in policy prefixes guarantee that only classes from P_1 participate in the lookup.

Theorem 1 shows that adding $|\mathcal{P}|$ extra bits per filter in P_{comb} is unavoidable.

Theorem 1. *For any $l > 2$ there exists a set of policies $\mathcal{P}$, $|\mathcal{P}| = l$, such that at least $|\mathcal{P}|$ extra bits must be prepended to every filter of a class $c \in \mathcal{C}$ instance to satisfy the property that $\text{pp}(c)$ matches pp_i if and only if $c \in P_i$.*

Proof. Consider a set $\mathcal{P}$ consisting of l policies over a set $\mathcal{C}$ of $2^l - 1$ different classes. For any two different classes c_i and c_j, the sets of policies containing c_i and c_j differ; we denote them by $\mathcal{P}_{c_i}$ and $\mathcal{P}_{c_j}$ respectively. Note that each non-empty subset of $\mathcal{P}$ is a set of policies containing some class c.

Consider two classes c_i and c_j such that $\mathcal{P}_{c_i} \subset \mathcal{P}_{c_j}$; all header prefixes pp_k matching to $\text{pp}(c_i)$ should match to $\text{pp}(c_j)$. It means that $\text{pp}(c_i)$ and $\text{pp}(c_j)$ have either equal bits on corresponding positions or have one of them equal to $*$. Also there is no sense to set the bit $\text{pp}(c_i)_k$ to $*$ and the bit $\text{pp}(c_j)_k$ to 0 or 1 for some position k since in this case in all header prefixes matching $\text{pp}(c_i)$, the bit at position k must equal $\text{pp}(c_j)_k$. Thus, the number of $*$ in $\text{pp}(c_i)$, which we denote by $|\text{pp}(c_i)|_*$, is strictly less than in $\text{pp}(c_j)$.

Consider a sequence of classes $c_2, c_3, \dots, c_l$ such that $|\mathcal{P}_{c_i}| = i$ and $\mathcal{P}_{c_{i-1}} \subset \mathcal{P}_{c_i}$. Such a sequence can be constructed starting from any class appearing only in two policies. Then

$$|\text{pp}(c_l)|_* \geq |\text{pp}(c_{l-1})|_* + 1 \geq \dots \geq (l-2) + |\text{pp}(c_2)|_*$$

The length of prefixes is at least $|\text{pp}(c_l)|_* \geq |\text{pp}(c_2)|_* + l - 2$. To finish the proof, it suffices to show that there exists a class c_2 such that $|\mathcal{P}_{c_2}| = 2$ and $|\text{pp}(c_2)|_* \geq 2$.

If there exists a class $\tilde{c}$ appearing in only one policy $P_{\tilde{c}}$, and $|\text{pp}(\tilde{c})|_* \geq 1$, then as c_2 we can choose any class that belongs to $P_{\tilde{c}}$ and some other policy. If such $\tilde{c}$ does not exist, consider three classes $\tilde{c}_i, \tilde{c}_j, \tilde{c}_k$ such that $|\mathcal{P}_{c_i}| = |\mathcal{P}_{c_j}| = |\mathcal{P}_{c_k}| = 1$. Prefixes $\text{pp}(\tilde{c}_i), \text{pp}(\tilde{c}_j), \text{pp}(\tilde{c}_i)$ do not contain $*$; therefore, at least two of these three prefixes differ in at least two positions. Suppose that $\text{pp}(\tilde{c}_i)$ and $\text{pp}(\tilde{c}_j)$ differ in two positions, so for the class c_2 with $\mathcal{P}_{c_2} = \mathcal{P}_{\tilde{c}_i} \cup \mathcal{P}_{\tilde{c}_j}$ the prefix $\text{pp}(c_2)$ has $*$ at these two positions. □

C. Necessary and sufficient conditions for ideal representations

In this part we formulate necessary and sufficient conditions that still guarantee the existence of ideal representations and show how to build them. For this purpose, we introduce the notion of a *joint graph* G^{jnt} for a set of policies $\mathcal{P}$ over classes $\mathcal{C}$; this is a directed graph $G^{\text{jnt}}(\mathcal{P}) = (\mathcal{C}, E^{\text{jnt}})$, where E^{jnt} contains an edge from c_i to c_j for $c_i, c_j \in \mathcal{C}$ if and only if $c_i \prec c_j$ in at least one policy in $\mathcal{P}$ (see Fig. 4b for an example).

Theorem 2. *For a given set of policies $\mathcal{P}$, there exists an ideal P_{comb} if and only if the corresponding G^{jnt} is acyclic.*

Proof. If G^{jnt} contains a cycle, then any possible linear ordering of classes in P_{comb} will contradict the order of classes in some policy P_i, i.e., there will exist $c, c' \in P_i$ such that $c \prec_{P_i} c'$ but c' appear before c in P_{comb}. Therefore, an ideal representation does not exist in this case.

If G^{jnt} is acyclic, we can construct an ideal representation from any topological order of the vertices of G^{jnt}: we put classes into P_{comb} in this order and prepend them by class prefixes. This representation is correct since for every P_i and every $c, c' \in P_i$ such that $c \prec_{P_i} c'$ the class c appears in P_{comb} before c'. □

The proof of Theorem 2 implies a straightforward algorithm that allows to check if an ideal representation exists and construct it in time $O(|\mathcal{C}| + |E^{\text{jnt}}|)$.

D. Multiple combined policies

So far we have assumed that a given set of policies $\mathcal{P}$ is represented by a single P_{comb}. Note that the single lookup constraint requires that all classes of the same policy in $\mathcal{P}$ are assigned into the same representing P_{comb}. Note also that multiple policies P_{comb} in a final representation cannot make an ideal representation possible relative to a single P_{comb} since all policies containing the same class should belong to the same P_{comb} in an ideal representation.

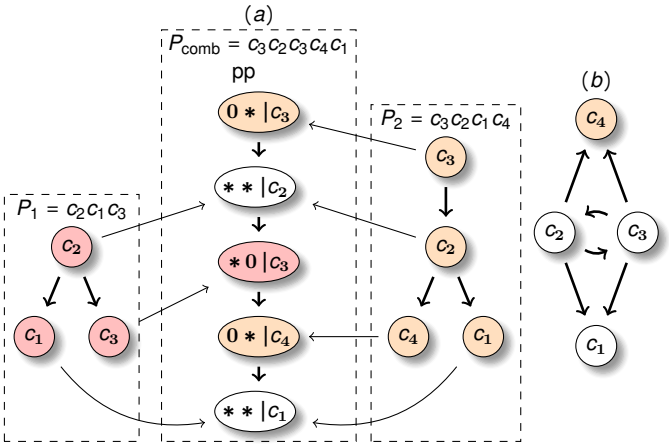


Fig. 4: (a) $\mathcal{P} = \{P_1, P_2\}$ and P_{comb} with duplicated c_3; (b) $G^{\text{jnt}}(\mathcal{P})$.

IV. Non-Ideal Representations

In this section we discuss how to deal with a given set of policies whose representations cannot be ideal.

A. Conflict resolution among partial orders

We begin with an example. Fig. 4a illustrates two policies P_1 and P_2. Since $c_2 \prec_{P_1} c_3$ and $c_3 \prec_{P_2} c_2$, there is no ideal P_{comb} satisfying partial orders of classes in both P_1 and P_2. Fig. 4b shows the corresponding joint graph, and it indeed contains a cycle. To satisfy the partial orders of P_1 and P_2 at the same time, we can add an additional instance of c_3 to P_{comb} with the corresponding bits of the policy prefix. In particular, the sequence of classes in P_{comb} shown on Fig. 4a is $\mathbb{S} = c_3c_2c_3c_4c_1$; its subsequence $c_2c_3c_1$ is compatible with partial order $\prec_{P_1}$, and another subsequence $c_3c_2c_4c_1$ is compatible with partial order $\prec_{P_2}$. Now P_{comb} contains two instances of c_3: the first is used during classification in P_2, and its policy prefix is $0*$; the second instance is used during classification in P_1, and its policy prefix is $*0$. In this case P_{comb} is non-ideal but still emulates P_1 and P_2.

In general, to deal with incompatible partial orders in policies we duplicate some instances of classes. Formally, a sequence of classes $\mathbb{S}$ is *compatible* with a policy P_i if there exists a subsequence $\mathbb{S}'$ of $\mathbb{S}$ that consists of a single instance of every class in P_i and for any two classes in P_i, if $c_i \prec_{P_i} c_j$ then c_i appears before c_j in $\mathbb{S}'$. Only instances of classes from this subsequence participate in the classification by policy P_i, i.e., only for them the ith bit of the policy prefix is set to $*$, while for all other instances the ith bit of the policy prefix is set to zero. Header prefixes are exactly the same as in the case of ideal policies. The following observation immediately follows.

Observation 3. *There exists a P_{comb} emulating a given $\mathcal{P}$ if duplications of classes from $\mathcal{C}$ are allowed.*

B. Problem statement

Clearly, the number of filters in classes should be taken into account during class duplications. We denote by $\mathrm{W}^+(\mathbb{S})$ the total overhead in filters from duplicated class instances in the resulting sequence of classes $\mathbb{S}$, i.e., the difference between the total number of filters in all class instances from $\mathbb{S}$ and the total number of filters in original classes without duplications.

Problem 1 (Policy Sequence Packing, PSP)**.** *Given a set of policies $\mathcal{P}$, find a sequence of classes $\mathbb{S}$ compatible with all policies in $\mathcal{P}$ that minimizes $\mathrm{W}^+(\mathbb{S})$.*

Theorem 4. *PSP is NP-hard even for two policies, $|\mathcal{P}| = 2$.*

C. Multiple combined policies again

Similar to ideal representations, multiple combined policies do not introduce additional savings compared to a single P_{comb}. On the other hand, two benefits can be achieved with multiple combined policies: (1) if all classes of some P_{comb} are disjoint, prepending extra bits to all filters of this P_{comb} is not required; (2) each group of combined policies can be optimized independently, which can allow to reduce the problem size. However, results of solving reduced problems separately can be combined back into a single unified P_{comb}, so in the following we assume a single P_{comb} in considered representations.

Algorithm 1 AO$(P_1 \dots, P_l)$

construct a graph $G^{\text{jnt}}(P_1, \dots, P_l)$;
$V^{\text{wfvs}} = WFVS(G^{\text{jnt}})$, with vertex weights $w(c) = |c|$;
initialize G^* as a subgraph of G^{jnt} induced by $V \setminus V^{\text{wfvs}}$;
for each $c \in V^{\text{wfvs}}$ **do**
    for each P_i containing c **do**
        add to G^* an instance $\tilde{c}_i$ of c;
for each P_i **do**
    for each $c \prec_{P_i} c'$ s.t. c or c' are in V^{wfvs} **do**
        add edge $(\tilde{c}_i, \tilde{c}_i')$ to G^*; ▷ here $\tilde{c}_i = c$ if $c \notin V^{\text{wfvs}}$
let $\mathbb{S}$ be a topological ordering of the vertices of G^*;
return $\mathbb{S}$.

V. Approximation algorithms

In this section, we introduce several approximation algorithms for PSP and study them analytically.

A. Feedback Vertex Set as a tool

Our algorithms for PSP will use algorithms for the *Weighted Feedback Vertex Set* (WFVS) problem [5], which is NP-complete. The *feedback vertex set* is a set of vertices in a directed graph $G = (V, E)$ with weighted vertices such that removing them forms an acyclic graph, and the WFVS problem is to find a feedback vertex set of minimal total weight. For instance, the work [6] proposes an algorithm for WFVS with approximation factor $O(\log |V| \log \log |V|)$, but there are other alternatives [7]. In what follows we denote by $\alpha(G)$ the approximation factor of an algorithm for the $WFVS$ problem on a graph G.

B. Algorithm AllOrOne

By Theorem 2 the main reason for class duplications are cycles in the joint graph. The algorithm AllOrOne (AO) constructs G^{jnt} and transforms it into an acyclic graph G^* whose topological order produces a valid sequence of classes $\mathbb{S}$ for P_{comb}.

AO finds a feedback vertex set V^{wfvs} in G^{jnt} with minimal total weight, where vertex weight equals the number of filters in the corresponding class. By $W(V)$ we denote the total weight of vertices in V. An induced subgraph on vertices that are not in V^{wfvs} is acyclic, therefore, the corresponding classes appear only once in $\mathbb{S}$. For a class $c \in V^{\text{wfvs}}$, the sequence $\mathbb{S}$ contains a separate c instance for each policy containing c.

To transform G^{jnt} into an acyclic graph G^*, the algorithm AO first removes all classes that are in V^{wfvs} (line 3 in Algorithm 1). Then for every class $c \in V^{\text{wfvs}}$ and every policy P_i containing c, a vertex $\tilde{c}_i$ is added into G^* (lines 4-6); other vertices in G^* will be connected with $\tilde{c}_i$ by edges induced by the partial order on $\prec_{P_i}$ (lines 7-9). A topological order of the vertices of G^* (line 10) forms a correct solution for the PSP problem (see Theorem 5). The running time of AO is $T_{\text{FVS}}(G^{\text{jnt}})$, where $T_{\text{FVS}}(G)$ is the running time of the algorithm for the WFVS problem.

Theorem 5. AO *correctly solves the PSP problem.*

Proof. If a graph G^* is acyclic, its topological order of vertices forms a correct $\mathbb{S}$ since all constraints introduced by partial orders of policies are represented by edges in G^*. So it is sufficient to show acyclicity of G^*. The first step of AO removes V^{wfvs} from V, making the graph G^* acyclic. Note that after adding a single vertex $\tilde{c}_i$ corresponding to the instance of c in P_i with incident edges, the graph G^* remains acyclic. This invariant holds since adding $\tilde{c}_i$ does not connect any new pair of vertices due to transitivity of $\prec_{P_i}$. Therefore, after adding $\tilde{c}_i$ a new cycle in G^* cannot appear. □

As we have already mentioned, a joint graph G^{jnt} contains edges induced by partial orders of originally given policies. To test whether G^{jnt} is acyclic, it suffices to maintain only edges for non-disjoint pairs of classes since other edges result from a transitive closure of policy partial orders and cannot introduce a cycle to G^{jnt}. On the other hand, for the correctness of AO it is necessary to consider all edges of G^{jnt}, otherwise the resulting feedback vertex set can lead to incorrect solutions.

Example 1. The following example illustrates AO running on two policies from Fig. 4. The joint graph for these policies has a cycle (see Fig. 5a); its FVS can be either $V^{\text{wfvs}} = \{c_2\}$ or $V^{\text{wfvs}} = \{c_3\}$. If $w(c_2) \le w(c_3)$ then $V^{\text{wfvs}} = \{c_2\}$ and c_2 is duplicated (Fig. 5b shows the corresponding G^* and $\mathbb{S}$). Otherwise, AO duplicates c_3 (see Fig. 5c).

Theorem 6. AO *has an approximation factor at most* $(|\mathcal{P}| - 1) \cdot \alpha(G^{\text{jnt}})$.

Proof. For any $\mathbb{S}$ produced by AO, the value of $\mathrm{W}^+(\mathbb{S})$ cannot exceed $(|\mathcal{P}| - 1) \cdot W(V_{\mathbb{S}})$, where $V_{\mathbb{S}}$ is the set of classes appearing in $\mathbb{S}$ at least twice. Since $V_{\mathbb{S}}$ is an FVS of G^{jnt}, the weight $W(V_{\mathbb{S}}) \le \alpha(G^{\text{jnt}}) V^{\text{wfvs}}_{\text{OPT}}$, where $V^{\text{wfvs}}_{\text{OPT}}$ is an FVS with the minimal total weight. On the other hand, the overhead in an optimal solution $\mathbb{S}_{\text{OPT}}$ is $\mathrm{W}^+(\mathbb{S}_{\text{OPT}}) \ge W(V_{\mathbb{S}_{\text{OPT}}}) \ge V^{\text{wfvs}}_{\text{OPT}}$. The theorem follows from these inequalities. □

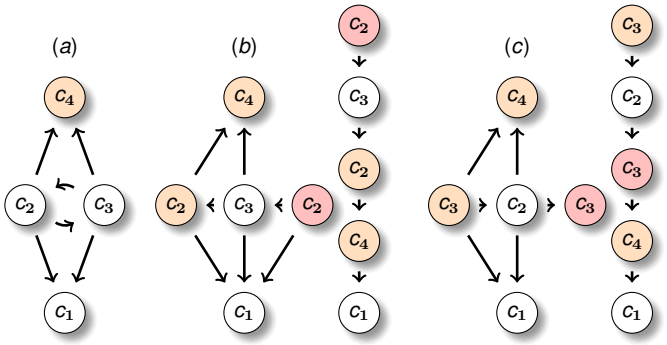


Fig. 5: (a) G^{jnt}; (b)-(c) two solutions depending on the values of $w(c_3)$ and $w(c_2)$.

Theorem 7. *The approximation factor of the* AO *algorithm is at least* $|\mathcal{P}| - 1$.

Proof. The proof is by showing a hard example, where $|P| = l$ policies are constructed from n different classes; each class contains exactly one filter. The partial order of the first $l-1$ policies is linear $c_1 c_2 \dots c_n$; the partial order of the last policy is also linear but contains the same classes in the reversed order $c_n c_{n-1} \dots c_1$. Any feedback vertex set of G^{jnt} consists of $(n-1)$ vertices, therefore, the total overhead $\mathrm{W}^+(\mathbb{S})$ incurred by AO is equal to $(n-1)(l-1)$. For an optimal solution $\mathbb{S}_{\text{OPT}} = c_1 c_2 \dots c_n \dots c_2 c_1$, the overhead is equal to $\mathrm{W}^+(\mathbb{S}_{\text{OPT}}) = n - 1$. □

Note that AO either creates a separate instance of a class c in $\mathbb{S}$ for every policy or has a common instance of c in $\mathbb{S}$ for all policies; this limits the optimization capabilities of the algorithm. In the proof of Theorem 7, AO finds a suboptimal $\mathbb{S}$ due to this limitation. One possible way to fix this is to apply additional optimization described in Section V-E. For the PSP instance in the proof of Theorem 7 these optimizations allow to produce an optimal $\mathbb{S}$, but in the general case they do not provide guarantees on $\mathrm{W}^+(\mathbb{S})$. In Section V-D we will introduce algorithms based on alternative principles that do not require unnecessary constraints.

C. Can we do better in the worst case?

In this section we show the inapproximability of PSP by reduction from the *Shortest Common Supersequence* (SCS) problem [8]. For a set of strings, SCS finds a string with minimal total length that contains all these strings as subsequences.

Theorem 8. *Unless* P = NP*, there is no polynomial algorithm for the PSP problem with a constant approximation factor on* $\mathrm{W}^+(\mathbb{S})$.

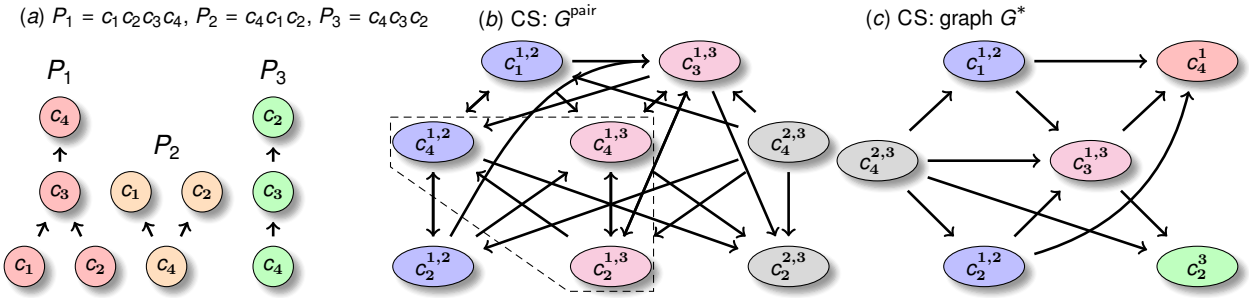


Fig. 6: (a) the input $\mathcal{P} = P_1, P_2, P_3$; (b) the graph G^{pair}; the dashed line encloses V^{wfvs}; (c) the graph G^*.

Algorithm 2 CS$(P_1 \ldots, P_l)$

1: construct the graph $G^{\text{pair}}(P_1, \ldots, P_l)$;
2: $V^{\text{wfvs}} = WFVS(G^{\text{pair}})$, with vertex weights $w(c) = |c|$;
3: **for** every $c \in \mathcal{C}$ **do**
4: $\mathbb{P}_c$:= min. size partition of $\mathcal{P}_c$ into admissible subsets;
5: construct G^* from $\bigcup_{c\in\mathcal{C}} \mathbb{P}_c$ and $\mathcal{P}$;
6: let $\mathbb{S}$ be a topological ordering of the vertices of G^*;
7: **return** $\mathbb{S}$.

Proof. We reduce SCS on alphabet Σ to PSP of the same size by setting $\mathcal{C} = \Sigma$, and assigning a unit weight $w(c_i) = 1$ to every class $c_i \in \mathcal{C}$. Also, we interpret each string $s \in \Sigma^*$ as a separate policy in $\mathcal{P}$ whose partial order is linear and coincides with s.

It is known that there is no algorithm for SCS with a constant factor on the length of SCS unless P = NP [9]. The reduction described above is correct only for SCS instances where all letters in the same input string are different, which corresponds to the natural constraint for classifiers that classes are not repeated in the same input policy. However, the instance of SCS used in [9] to show inapproximability of SCS never uses a letter twice in the same input string. Thus, there is no algorithm for the PSP problem with a constant approximation ratio on the total weight $W(\mathbb{S})$ of class instances in $\mathbb{S}$ and on $\mathrm{W}^+(\mathbb{S})$ since $\mathrm{W}^+(\mathbb{S}) \le W(\mathbb{S})$ (unless P = NP). □

Existence of sublinear approximation algorithms with respect to $|\mathcal{P}|$ for the PSP problem is unclear; due to the reduction in Theorem 8, such an algorithm would solve a special case of the SCS problem with a sublinear approximation factor. To the best of our knowledge, even for this special case the existence of sublinear approximation algorithms for SCS is an open problem.

D. Algorithm CLIQUESHARE

In AO, a vertex in G^{jnt} indicates that all instances of the same class in different policies can share a common instance in $\mathbb{S}$. In the CLIQUESHARE (CS) algorithm we construct another graph G^{pair} allowing to operate with a better resolution. Denote by $\mathcal{P}_c$ a set of policies containing a class c. For each class c and each subset A of two policies in $\mathcal{P}_c$, G^{pair} contains a vertex c^A. For each P_i and any two classes $c_1 \prec_{P_i} c_2$, G^{pair} has an edge $(c_1^A, c_2^{A'})$ for all pairs of policies $A, A' \in \mathcal{P}$ containing P_i (e.g., Fig. 6b shows a G^{pair} graph for the input $\mathcal{P}$ shown on Fig. 6a).

At the beginning, CS finds a feedback vertex set V^{wfvs} in G^{pair} with minimal total weight, where the weight of a vertex is equal to the number of filters in the corresponding class (line 2 in Algorithm 2). If c^A is in V^{wfvs} then the resulting $\mathbb{S}$ contains different instances of c for the policies $P_i, P_j \in A$. The set of policies can share the same instance of a class c if for any two policies from this set, the corresponding vertex for a class c in G^{pair} is not in V^{wfvs}, we call such sets *admissible* subsets of $\mathcal{P}_c$.

For each class c, CS computes a partition $\mathbb{P}_c$ of $\mathcal{P}_c$ into admissible subsets, minimizing the total number of sets in $\mathbb{P}_c$ (line 4 in Algorithm 2). For a class appearing only in a single policy in $\mathcal{P}$, the partition consists of a single admissible subset containing this policy. Each set in $\mathbb{P}_c$ corresponds to a separate instance of c in $\mathbb{S}$. After that CS constructs an acyclic G^*, for which a topological order of vertices forms a valid $\mathbb{S}$. For each admissible subset $B \in \mathbb{P}_c$, the graph G^* has a vertex c^B. The edges of G^* are defined similarly to G^{pair}: there is an edge $(c_1^B, c_2^{B'})$ for all $c_1, c_2 \in \mathcal{C}$, $B \in \mathbb{P}_{c_1}, B' \in \mathbb{P}_{c_2}$ such that there exists a policy $P \in B \cap B'$ for which $c_1 \prec_P c_2$.

To find a partition into admissible subsets, CS can use the algorithm that greedily constructs admissible subsets with running time $O(|\mathcal{P}_c|^2)$. Alternatively, it can use an algorithm based on dynamic programming that finds a partition with minimal number of subsets in time $O(3^{|\mathcal{P}_c|}|\mathcal{P}_c|^2)$. For both algorithms, CS has the same approximation factor but the first one has better time complexity, while the second algorithm finds an optimal partition into admissible subsets.

Example 2. The following example illustrates CS running on three policies (see Fig. 6a). The weights of all classes are the same. At first CS constructs G^{pair} (see Fig. 6b), which has $\binom{3}{2} = 3$ vertices for c_2 and c_4 and one vertex for c_1 and c_3. G^{pair} has many cycles; one of its feedback vertex sets with minimal total weight is $V^{\text{wfvs}} = \{c_2^{1,3}, c_4^{1,2}, c_4^{1,3}\}$. The partitions of $\mathcal{P}_{c_1}$ and $\mathcal{P}_{c_3}$ consist of a single set since the vertices for c_1 and c_3 in G^{pair} do not appear in V^{wfvs}. For c_2 and c_4 optimal partitions into admissible subsets can be $\mathbb{P}_{c_2} = \{\{P_1, P_2\}, \{P_3\}\}$ and $\mathbb{P}_{c_4} = \{\{P_1\}, \{P_2, P_3\}\}$. The

resulting G^* is acyclic (see Fig. 6c). Every topological order on G^* yields a valid $\mathbb{S}$, e.g., $c_4^{2,3} c_1^{1,2} c_2^{1,2} c_3^{1,3} c_4^1 c_2^3$.

Note that CS and AO coincide in the case of two policies. Observe that CS finds an optimal $\mathbb{S}$ for the example in the proof of Theorem 7. In the following we prove that CS works correctly and estimate its approximation factor.

Theorem 9. CS *correctly solves the PSP problem.*

Proof. Similar to Theorem 5, we only need to show that G^* is acyclic. The construction of G^* from G^{pair} is equivalent to the following three-step procedure: (1) initialize G^* as a subgraph of G^* induced by all vertices c^A such that the policies $P_i, P_j \in A$ belong to the same admissible subset of $\mathbb{P}_c$; (2) for each $c \in \mathcal{C}$ add vertices into G^* for all admissible subsets of $\mathbb{P}_c$ consisting of a single policy; (3) for each $c \in \mathcal{C}$, "shrink" vertices corresponding to policies belonging to the same admissible subset.

A graph G^* is acyclic after the first step since at least all vertices in found FVS of G^{pair} are not included in G^*. After the second step G^* remains acyclic due to transitivity of partial orders, which is similar to the proof of Theorem 5. To prove that G^* will remain acyclic after the third step, it is sufficient to show that G^* remains acyclic after every shrink. A shrink produces a cycle in G^* if and only if before this shrink G^* had a path between two vertices corresponding to class with policies in the same admissible subset. Assume that there is such path w for a class c. W.l.o.g. let P_1 be a policy whose partial order defines the first edge of w, and P_2 be a policy whose partial order defines the last edge of w. The vertex c^A, where $P_1, P_2 \in A$ has an outgoing edge to the second vertex of w and has an incoming edge from penultimate vertex of w. Hence, there is a cycle in G^* containing a vertex c^A which is a contradiction to the assumption that G^* has no cycles before the current shrink. □

Theorem 10. CS *has an approximation factor of at most* $\alpha(G^{pair})\lfloor \frac{|\mathcal{P}|^2}{4} \rfloor$.

Proof. First, we are to show that for a produced sequence $\mathbb{S}$ by CS, $W^+(\mathbb{S})$ does not exceed the weight of the corresponding V^{wfvs}. Each class c appearing t times in $\mathbb{S}$ increases the value of $\mathrm{W}^+(\mathbb{S})$ by $(t-1)w(c)$. On the other hand, the found FVS in G^{pair} should contain at least $t-1$ vertices for c. Otherwise, at least two admissible subsets corresponding to the instances of c in $\mathbb{S}$ can be merged into a bigger admissible subset. Note that such partitions cannot be constructed by CS. Therefore, $\mathrm{W}^+(\mathbb{S}) \le W(V^{\mathsf{wfvs}})$. As in Theorem 6, $W(V^{\mathsf{wfvs}}) \le \alpha(G^{pair}) W(V^{\mathsf{wfvs}}_{\mathsf{OPT}}) \le \alpha(G^{pair}) W(V_{\mathbb{S}_{\mathsf{OPT}}})$, where $V^{\mathsf{wfvs}}_{\mathsf{OPT}}$ is FVS in G^{pair} with the minimal total weight and $V_{\mathbb{S}_{\mathsf{OPT}}}$ is FVS by which CS produces an optimal sequence $\mathbb{S}_{opt}$.

To finish the proof, we need to show that $W(V_{\mathbb{S}_{\mathsf{OPT}}}) \le \lfloor \frac{l^2}{4} \rfloor W^+(\mathbb{S})$. Consider a class c belonging to l_c policies in $\mathcal{P}$ and appearing in $\mathbb{S}_{\mathsf{OPT}}$, t times. Denote by $s_{c,i}$ (where $1 \le i \le t$), a number of policies in an i-th admissible subset. A set $V_{\mathbb{S}_{\mathsf{OPT}}}$ can contain only c^A vertices such that $P_i, P_j \in A$ belong to different admissible subsets; the number of such vertices is at most $\frac{1}{2}\left(l_c^2 - \sum_{i=1}^t s_{c,i}^2\right)$. We can now see that for each $1 \le t \le l_c$

$$\max_{s_{c,1}+\ldots+s_{c,t}=l_c} \left(l_c^2 - \sum_{i=1}^{t} s_{c,i}^2\right) \le \left\lfloor \frac{l_c^2}{4} \right\rfloor \cdot (t-1) \le \left\lfloor \frac{|\mathcal{P}|^2}{4} \right\rfloor \cdot (t-1).$$

Hence, a class c increasing $W^+(\mathbb{S}_{\mathsf{OPT}})$ by $(t-1)\cdot w(c)$ can also increase $W(V_{\mathbb{S}_{\mathsf{OPT}}})$ by the value not exceeding $\lfloor \frac{|\mathcal{P}|^2}{4} \rfloor \cdot (t-1) \cdot w(c)$; summing this over all classes $c \in \mathcal{C}$, we find that $W(V_{\mathbb{S}_{\mathsf{OPT}}}) \le \lfloor \frac{|\mathcal{P}|^2}{4} \rfloor W^+(\mathbb{S})$. □

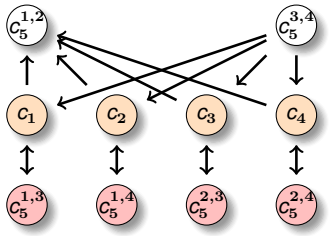


Fig. 7: G^{pair} for the lower bound of CS; vertices of different types are in different colors

Theorem 11. *The approximation factor of* CS *is at least* $\lfloor \frac{|\mathcal{P}|^2}{4} \rfloor$.

Proof. We provide an example with $|P| = l$ policies and $n = \lfloor \frac{l^2}{4} \rfloor + 1$ different classes. We add a class c_n to all policies. Also we enumerate all pairs of policies (P_i, P_j) such that $i \le \lfloor \frac{l}{2} \rfloor$ and $j > \lfloor \frac{l}{2} \rfloor$, there are $n-1$ such pairs. For each enumerated pair (P_i, P_j) we add a class c_k to policies P_i and P_j, where k is a number of this pair. The class $c_k \prec_{P_i} c_n$ and $c_k \succ_{P_j} c_n$. The number of rules in classes is the following: $|c_i| = x, i = 1 \ldots n-1, |c_n| = x+1$.

A graph G^{pair} consists of three categories of vertices (see Figure 7 for $l = 4$ policies): (1) the $n-1$ vertices corresponding to a class c_n in the enumerated pairs of policies; (2) the $n-1$ vertices for all other classes c_i, where $i < n$; (3) the vertices corresponding to c_n in non-enumerated pairs of policies which do not affect acyclicity of G^{pair}. The G^{pair} graph contains $n-1$ bidirectional edges between vertices of the first two types, which form a maximal matching.

An FVS of G^{pair} with the minimal total weight V^{wfvs} consists of all vertices of the second category. Therefore, CS produces $\mathbb{S}$ containing one copy of c_n and two copies for each other class; the total overhead is equal $\mathrm{W}^+(\mathbb{S}) = (n-1)\cdot x$. The optimal solution for this example is $\mathbb{S}_{\mathsf{OPT}} = c_n c_1 c_2 \ldots c_n$ with $\mathrm{W}^+(\mathbb{S}) = x+1$; taking x arbitrarily big, we show the stated lower bound.

To obtain an example with an arbitrarily large number of classes, we take multiple instances of the proposed example and combine them into one joint input: we merge policies P_i with the same index i, and classes from different instances of the example are different. □

The running time of CS is $T_{FVS}(G^{\mathsf{pair}}) + O(|\mathcal{C}| \cdot T_{\text{part}}(|\mathcal{P}|))$, where T_{part} is the time complexity of the algorithm finding partitions into admissible subsets. The approximation factor of CS is quadratic on $|\mathcal{P}|$ and worse than for AO for all $|\mathcal{P}| > 3$. Nevertheless, we will see in Section VII that CS performs better on average since it operates with a better resolution.

E. Local descent

Both AO and CS algorithms can be further improved by additional optimizations. One of optimization procedures comes from the fact that proposed algorithms do not usually guarantee that $\mathbb{S}$ will be a *local minimum* solution, i.e., it might happen that one can remove some class instances from the resulting $\mathbb{S}$ and still get a valid sequence for P_{comb}. The *local descent* (LD) procedure is defined in the following way: given $\mathbb{S}$, try to remove classes from $\mathbb{S}$ one by one, while $\mathbb{S}$ remains compatible with all policies from $\mathcal{P}$. LD can be implemented in time $O\left(|\mathbb{S}| + \sum_i (|P_i| + D(P_i))\right)$, where $D(P_i)$ is the number of pairs of classes from P_i that are disjoint. We will see in Section VII that LD does bring improvements in practice, although it has no effect on the worst case bounds.

VI. Dynamic updates

Although economic models rarely change, support of dynamic updates in represented policies can become important in some deployment scenarios. We support two basic operations on policies in $\mathcal{P}$: (1) $\text{delete}(P, c)$, remove a class c from a policy P; (2) $\text{insert}(P, c, c_{succ})$, add a class c into $\mathbb{S}(P)$ just before the class c_{succ}.

Hypothetically, we can generalize the proposed algorithms in Section V to support dynamic operations by maintaining dynamically graphs $G^{\mathsf{jnt}}, G^{\mathsf{pair}}, G^*$ and the sequence $\mathbb{S}$. But running dynamic versions of AO and CS may be very time-consuming. They are better suited for environments where updates happen in batches. In this section we propose another algorithm implementing the right balance between time complexity and optimization efficiency in a dynamic environment.

Each insert/delete operation modifies a sequence of classes $\mathbb{S}$ which represents the corresponding P_{comb}. Note that after each operation policy prefixes should be updated assuring that P_{comb} emulates $\mathcal{P}$.

When we delete a class c from a policy P, we remove an instance of c in $\mathbb{S}$ if this instance corresponds only to P. After a delete operation $\mathbb{S}$ remains correct: if necessary, we can further optimize $\mathbb{S}$ by LD optimization to remove redundant class instances.

The case of an insert operation is more complicated. Let c be the class that is to be inserted, and let $\mathcal{C}_{\text{prec}}$ be the set of classes in $\mathcal{P}_i$ preceding c in $\prec_{P_i}$, and similarly $\mathcal{C}_{\text{succ}}$ be the set of classes in $\mathcal{P}_i$ succeeding c in $\prec_{P_i}$. If $\mathbb{S}$ is already compatible with the new P_i the insertion is done. Otherwise, to make $\mathbb{S}$ compatible with the new P_i, we insert to the jth position inside $\mathbb{S}$ an instance of c and instances of some classes

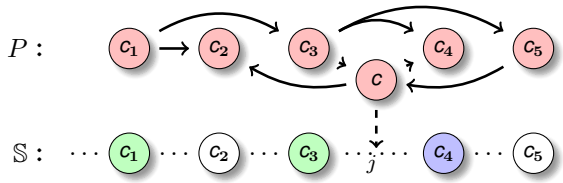


Fig. 8: Insert of an instance for a new class $c \in P$ into the j-th position of $\mathbb{S}$; the white instances are not in $\mathbb{L}_j$ or $\mathbb{R}_j$

in $\mathcal{C}_{\text{prec}}$ and $\mathcal{C}_{\text{succ}}$ that minimize the total number of filters in inserted instances.

Let $\mathbb{L}_j$ be the longest subsequence of $\mathbb{S}[0 \dots j]$ satisfying the following conditions:

(1) $\mathbb{L}_j$ contains at most one instance of every class from P_i;
(2) an instance of c_1 is in $\mathbb{L}_j$ only if for each $c_2 \prec_{P_i} c_1$ the instance of c_2 appears in $\mathbb{L}_j$ before c_1.

We insert instances of classes from $\mathcal{C}_{\text{prec}}$ that are not in $\mathbb{L}_j$ just before the added instance of c satisfying $\prec_{P_i}$. Similarly, we define a subsequence $\mathbb{R}_j$ in the suffix of $\mathbb{S}$ starting from the $j+1$-th position. But in this case condition (2) is reversed: an instance of c_1 is in $\mathbb{R}_j$ only if for each c_2, $c_1 \prec_{P_i} c_2$, the instance of c_2 appears in $\mathbb{R}_j$ after c_1. We insert all instances of classes from $\mathcal{C}_{\text{succ}}$ that are not in $\mathbb{R}_j$ just after the added instance of c satisfying $\prec_{P_i}$. Figure 8 illustrates this insertion procedure for a policy P_1: $\mathbb{L}_j = c_1 c_3$, $\mathbb{R}_j = c_4$, and class instances c_2, c_5 are inserted together with the instance of c.

Theorem 12. *All class instances inserted to the jth position of $\mathbb{S}$ make $\mathbb{S}$ compatible with a new P_i.*

Proof. Let $\mathbb{I}_j$ be an inserted sequence of class instances to the j-th position in $\mathbb{S}$. Let $\mathbb{R}'_j$ be a subsequence of $\mathbb{R}_j$ that does not contain class instances from $\mathbb{L}_j$. Note that condition (2) from the definition of $\mathbb{R}_j$ is also satisfied for $\mathbb{R}'_j$. Consider a subsequence of $\mathbb{S}$ constructed by the concatenation of $\mathbb{L}_j, \mathbb{I}_j, \mathbb{R}'_j$. This subsequence contains exactly one instance of each class from the new P_i and satisfies the partial order of the new P_i. □

Correctness of the insert operation immediately follows by Theorem 12. For a position j, the total number of rules in all inserted class instances equals

$$\mathcal{I}_j = |c| + \sum_{c' \in \mathcal{C}_{\text{prec}} \setminus \mathbb{L}_j} |c'| + \sum_{c' \in \mathcal{C}_{\text{succ}} \setminus \mathbb{R}_j} |c'|$$

Among all potential positions for insertion, we choose the one that minimizes $\mathcal{I}_j$. We denote $\mathcal{L}_j = \sum_{c' \in \mathcal{C}_{\text{prec}} \cap \mathbb{L}_j} |c'|$ and $\mathcal{R}_j = \sum_{c' \in \mathcal{C}_{\text{succ}} \cap \mathbb{R}_j} |c'|$; then the expression for $\mathcal{I}_j$ can be rewritten as

$$\mathcal{I}_j = |c| + \sum_{c' \in \mathcal{C}_{\text{prec}}} |c'| + \sum_{c' \in \mathcal{C}_{\text{succ}}} |c'| - \mathcal{L}_j - \mathcal{R}_j$$

To find an optimal insertion position, we compute $\mathcal{L}_j, \mathcal{R}_j$ for all positions j.

We calculate $\mathcal{L}_j$ in the order of increasing j. During this computation, we maintain the sets $\mathbb{L}_j$ and $\mathbb{L}_j \cap \mathcal{C}_{\text{prec}}$. If $\mathbb{S}[j] \notin$

$\mathbb{L}_{j-1}$ and all classes preceding $\mathbb{S}[j]$ in P_i belong to $\mathbb{L}_{j-1}$ then $\mathbb{L}_j = \mathbb{L}_{j-1} \cup \{\mathbb{S}[j]\}$, otherwise $\mathbb{L}_j = \mathbb{L}_{j-1}$. Therefore, we can compute $\mathbb{L}_j$ (and $\mathbb{L}_j \cap \mathcal{C}_{\text{prec}}$ with $\mathcal{L}_j$ respectively) from $\mathbb{L}_{j-1}$ in time proportional to the number of classes preceding $\mathbb{S}[j]$ in P_i. To speed up this process, when adding $\mathbb{S}[j]$ to $\mathbb{L}_j$ we look at all classes that succeed $\mathbb{S}[j]$ and mark those for which all classes preceding them in P_i are in $\mathbb{L}_j$. For a subsequent position j', we update $\mathbb{L}_{j'}$ if and only if $\mathbb{S}[j']$ is not in $\mathbb{L}_{j'-1}$ and corresponds to a marked class. This implementation allows to compute $\mathcal{L}_j$ for all positions in time $O(|\mathbb{S}| + |P_i| + D(P_i))$. The values of $\mathcal{R}_j$ can be computed in reverse order of j in a similar way. Therefore, the total time complexity of the insert operation equals $O(|\mathbb{S}| + |P_i| + D(P_i))$.

Actually, when $\mathbb{S}[j]$ is an instance of an inserted class c, we can remove the jth element of $\mathbb{S}$ since a new copy of c was inserted. For such positions we do not include $|c|$ into the value of $\mathcal{I}_j$.

We can achieve additional memory savings by running LD on the resulting $\mathbb{S}$. Since the time complexity of LD is $O\left(|\mathbb{S}| + \sum_i (|P_i| + D(P_i))\right)$, we can run it after each insert or delete operation.

VII. Experimental evaluation

A. Combined representations

Algorithms. We compare the algorithms AO, CS, weighted SCS, and UpperBound (UB), where UB is a heuristic that simply concatenates all $\mathbb{S}(P_i)$, $P_i \in \mathcal{P}$ into a single $\mathbb{S}$. For each considered algorithm we also evaluate its extended version, where we apply LD to its result.

Methodology. Unfortunately, de-facto standard frameworks to generate inputs such as ClassBench [10] do not allow for a sufficiently refined control over the actions that implicitly define classes. Hence, we experimented on synthetic data produced in a way similar to intended usage:

(1) generate sizes of classes from $\mathcal{C}$;
(2) pick which classes are non-disjoint;
(3) generate a set of policies $\mathcal{P}$ on classes from $\mathcal{C}$, with each policy consisting of the same number of classes.

For every setting, we performed 100 experiments with random instances and different random seeds (virtually all algorithms are randomized because the topological order on G^* is not unique in most cases); we show averaged results. Implementation of our experiments is available at [11], and the results are summarized on Figure 9. The Y-axis in all plots shows the relative overhead $\mathrm{W}^+(\mathbb{S})/W(\mathcal{C})$, where $W(\mathcal{C})$ is the total size of all classes from $\mathcal{C}$; we show relative values of the overhead because absolute values change a lot from instance to instance.

The number of rules in a combined representation significantly depends on the structure of given policies. There are three main characteristics of the input structure: (1) number of intersecting classes, (2) average number of policies that contain a class, (3) total number of policies. We generate inputs with different values of these characteristics. Fig 9a shows how the relative overhead grows as the average number of classes k intersecting with each $c \in \mathcal{C}$ increases. In Fig. 9b we vary the number of classes in each policy $|\mathcal{C}_P|$. In these experiments each class belongs to $\frac{|\mathcal{C}_P|}{|\mathcal{C}|} \cdot |\mathcal{P}|$ policies on average. Fig. 9c shows the relative overhead for inputs with different numbers of policies.

Algorithm CS with LD postprocessing (the strongest combination in our experiments) outperforms other algorithms regardless of input characteristics; this confirms our hypothesis that this algorithm is the best choice for a vast majority of inputs with different policy structures. Evaluations also show that CS with LD constructs a representation with only 20-50% of the rules in duplicated instances of classes compared to representations where policies are stored separately. In what follows we describe our evaluation results for all algorithms in detail.

LD: The evaluations show that class sharing introduces substantial savings, and changes the linear behavior to nearly logarithmic in Fig. 9.c; even UB with LD reduces the overhead for additional instances of classes (e.g., by 23-63% in Fig. 9a); still it is worse than the other considered algorithms. LD is especially effective for UB, SCS, and AO algorithms saving 47%, 30%, and 37% on average in the second set of experiments (see Fig. 9b); CS can be also improved by LD, but in this case its effect is not substantial (at most 12% in all experiments) since produced results are close to local minimum. Comparing SCS with and without LD in Fig. 9a where the former remains constant, one can see how exploiting partial orders can significantly affect optimization results, and how the effect diminishes as classes start to intersect more (k increases in Fig. 9a); this is due to optimality of SCS in the case of linear orders.

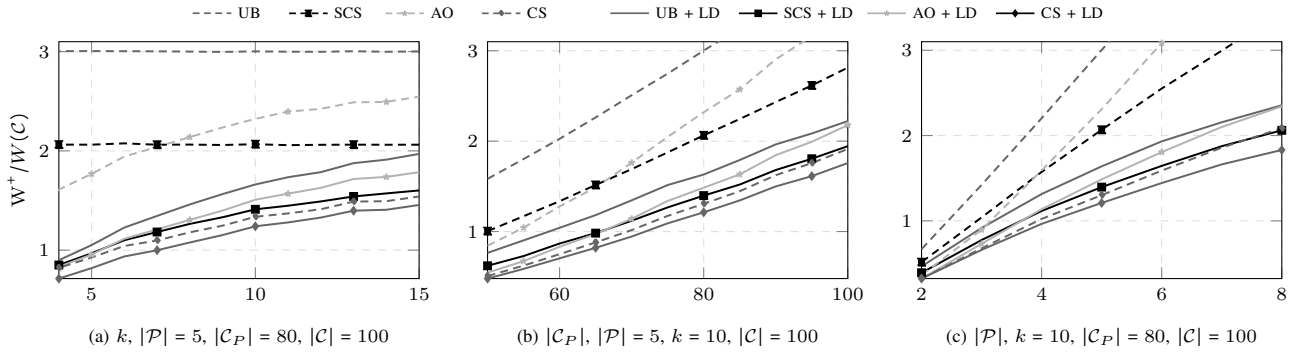
AO: In evaluations, AO outperforms UB by as much as 30% (see Fig. 9.b) but performs 37% worse than CS in the same experiment; Compared to SCS, AO is better only when the overhead is low (e.g., less than 150% in Fig. 9c). The main reason for this is a low resolution of AO adding for every class c either a single instance of c or a separate instance for every policy containing c. CS is proposed specifically to overcome this limitation.

CS: This algorithm significantly outperforms other evaluated algorithms. Moreover, even CS without LD outperforms all other algorithms with LD, except for the case of $|\mathcal{P}| = 8$ policies (see Fig. 9b), where CS without LD works a bit worse than SCS with LD since LD becomes more efficient in this case. For inputs generated with parameters ($k = 10$, $|\mathcal{C}_P| = 80$, $|\mathcal{C}| = 100$, $|\mathcal{P}| = 5$), CS with LD requires only 40% of the rules of the duplicated instances of classes versus UB (see Fig. 9c).

B. Dynamic updates

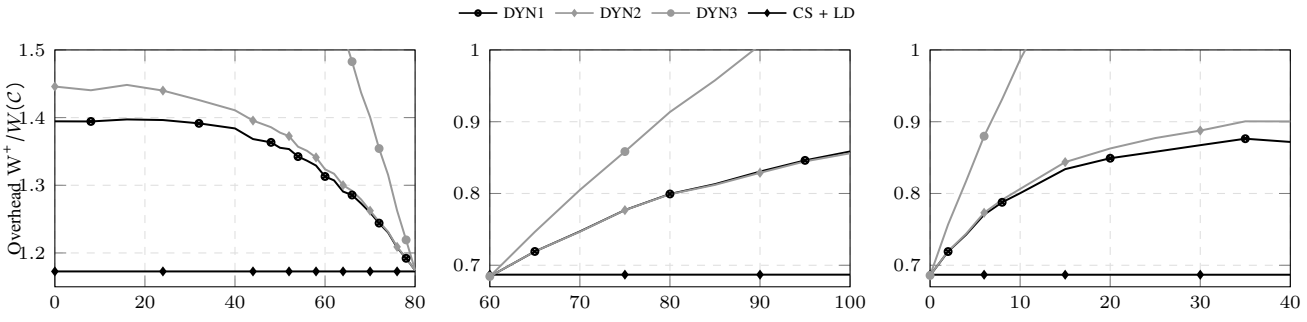
Algorithms. In this part we evaluate insert/remove operations introduced in Section VI and compare the following three variations:

(1) DYN1 applies LD after each insert/remove operation;
(2) DYN2 applies LD only after all operations;
(3) DYN3 never apply LD.



(a) k, $|\mathcal{P}| = 5$, $|\mathcal{C}_P| = 80$, $|\mathcal{C}| = 100$ (b) $|\mathcal{C}_P|$, $|\mathcal{P}| = 5$, $k = 10$, $|\mathcal{C}| = 100$ (c) $|\mathcal{P}|$, $k = 10$, $|\mathcal{C}_P| = 80$, $|\mathcal{C}| = 100$

Fig. 9: Relative overhead $\mathrm{W}^+/W(\mathcal{C})$ as a function of: (a) average degree of intersection k; (b) number of classes in each policy $\mathcal{P}$; (c) number of policies $|\mathcal{P}|$



(a) $|\mathcal{C}_P|$ for $P \in \mathcal{P}_s$, $|\mathcal{C}_{P'}| = 80$ for $P' \in \mathcal{P}_f$ (b) $|\mathcal{C}_P|$ for $P \in \mathcal{P}_s$, $|\mathcal{C}_{P'}| = 60$ for $P' \in \mathcal{P}_f$ (c) # of replaced classes, $|\mathcal{C}_{P'}| = 60$ for $P' \in \mathcal{P}_f$

Fig. 10: Experiments with dynamic updates: (a) insertion scenario; (b) removal scenario; (c) replacement scenario.

Denote by $\mathcal{P}_f$ a set of policies after all applied dynamic operations. We compare $\mathbb{S}$ obtained after all operations with offline results of CS with LD calculated directly on $\mathcal{P}_f$.

Methodology. We denote by $\mathcal{P}_s$ a set of policies just before dynamic operations. In all our experiments we first choose a final set of policies $\mathcal{P}_f$ in the same way as for the algorithms in the offline case. Then we generate an initial set of policies $\mathcal{P}_s$ from $\mathcal{P}_f$, and for $\mathcal{P}_s$ we construct $\mathbb{S}$ by CS with LD and then we apply insert/remove operations one by one in a random order transforming $\mathcal{P}_s$ into $\mathcal{P}_f$.

We evaluate three different scenarios, each scenario defined by the method constructing $\mathcal{P}_s$ from $\mathcal{P}_f$:

(1) insertion scenario (Fig. 10a): generate $\mathcal{P}_s$ by removing classes from $\mathcal{P}_f$, and then run dynamic updates to insert back the removed classes until we get $\mathcal{P}_f$.
(2) removal scenario (Fig. 10b): generate $\mathcal{P}_s$ by inserting new classes to $\mathcal{P}_f$, and then run dynamic updates to remove classes until we get $\mathcal{P}_f$;
(3) replacement scenario (Fig. 10c): obtain $\mathcal{P}_s$ by replacing a certain number of classes in each policy of $\mathcal{P}_f$, and run dynamic updates to remove classes that are in $\mathcal{P}_s$ and not in $\mathcal{P}_f$ and insert classes that are in $\mathcal{P}_f$ and not in $\mathcal{P}_s$.

All policies in $\mathcal{P}_f$ (or in $\mathcal{P}_s$) have the same number of classes. In all experiments $|\mathcal{C}| = 100, |\mathcal{P}| = 5, |k| = 10$.

As we can see in Figure 10, the usage of LD is extremely important after modification operations. In the case of class removals (Fig. 10b), one can apply LD after all operations, and the result will stay the same. In the insertion scenario (Fig. 10a), if no more than 25% of new classes are added (i.e. the $|\mathcal{C}_P|$ is at least 60), running LD after every operation (DYN1) do not introduce additional gains versus running LD after the whole batch (DYN2). Otherwise, the results become worse if we apply LD after all inserts, e.g., the difference goes up to 4% as we decrease $|\mathcal{C}_P|$ (see, again, Fig. 10a); the algorithm that never uses LD (DYN3) looses in all scenarios, e.g., by 17% in the removal scenario in Fig. 10b, $|\mathcal{C}_P| = 85$, and much more as we increase $|\mathcal{C}_P|$. Finally, the algorithm constructing P_f with dynamic inserts from scratch ($|\mathcal{C}_P| = 0$ in Fig. 10a) builds a combined representation that has 23% more rules in duplicated class instances then CS with LD.

VIII. Previous Work

Finding efficient representation of a single instance of a packet classifier is a well-known problem. Approaches to this problem fall into two major categories: software-based and hardware-based solutions that mainly spans three techniques: decision trees, hashing, and coding-based compression [12]–[18]. In decision trees, finding a matching rule is based on tracing a path in a decision tree (e.g., [12]; however, there is an inherent tradeoff between space and time complexity in these approaches. Hash-based solutions that match a packet to its possible matching rules have also been considered [14]. Other

works discuss efficient hardware implementations that have no native representation. E.g., TCAMs have no native support for ranges, so one has to translate ranges into TCAM-friendly prefix matching representations [15]–[18]. Unfortunately, in most cases these methods apply only to rules with a limited number of fields or perform poorly as it increases. The works [19]–[21] exploits structural properties of classifiers, in particular order-independence, to create equivalent classifiers with a fewer number of fields. Representations of order-independent classifiers as Boolean expressions and the relation to the MinDNF problemis studied in [22]. In [23], per-flow per-policy class state is implemented when the same policy can be attached to multiple flows. In general, the previous works mostly concentrate on optimizing a single instance of policy classifier, whereas we concentrate on combined optimizations of multiple different policies. The problem of splitting a policy into several lookup tables while minimizing the maximal local table size has been broadly studied in [24], [25] and found to be an intractable optimization problem. The main contribution of [26] is an optimal algorithm with linear time complexity that can handle dynamic fields at the price of a single bit of metadata prepended to every packet. Note that all these proposals are orthogonal to our combined policy representations and can be used alongside with it.

IX. Conclusion

In this work, we exploit new alternatives to optimize policy classifiers, introducing novel techniques that operate on the inter-policy level. We show how to share classes among policies and analyze the proposed algorithms analytically. Our evaluation study has shown significant gains from sharing classes and using partial policy orders on a single network element varying structural properties of represented policies on a single network element. We hope to study efficiency of the proposed algorithms on policies created from network-wide economic models in the further study.

Appendix

Proof of Theorem 4. The proof is by reduction from the WFVS problem [5]. For a directed weighted on vertices graph $G = (V, E)$ we construct $\mathcal{P} = \{P_1, P_2\}$ such that $\mathbb{S}$ with a minimal overhead corresponds to FVS in G with the minimal total weight.

For each vertex $v \in V$ we create three classes: an *input* class c_v^1, a *middle* class c_v^2, and an *output* class c_v^3. For each edge $e \in E$ we create a class c_e. The size of input classes is equal to the weights of the corresponding vertices in G, the size of all other classes is equal to the total weight of all vertices in V plus 1. For each $v \in V$, the class c_v^2 is non-disjoint with both c_v^1 and c_v^3. For each $e = (u, v) \in E$, the class c_e is non-disjoint with both c_u^3 and c_v^1. The all other classes are pairwise disjoint. The policy P_1 consists of the classes c_v^1, c_v^3, c_e for each $v \in V$ and $e \in E$. The partial order of P_1 is defined as follows: for each edge $e = (u, v) \in E : c_u^3 \prec_{P_1} c_e \prec_{P_1} c_v^1$. The policy P_2 consists of all classes c_v^1, c_v^2, c_v^3 for each $v \in V$. The partial order of P_2 is defined as follows: for each $v \in V : c_v^1 \prec_{P_2} c_v^2 \prec_{P_2} c_v^3$.

Each feedback vertex set V_G in a graph G corresponds to FVS $V_{G^{\mathsf{jnt}}}$ in a joint graph G^{jnt} for P_1, P_2: $c_v^1 \in V_{G^{\mathsf{jnt}}} \equiv v \in V_G$. Since the size of each non-input class is bigger than the weight of all vertices in G, an optimal FVS in G corresponds to the optimal FVS in G^{jnt}. Each valid $\mathbb{S}$ corresponds to FVS in G^{jnt} consisting of the classes appearing twice in $\mathbb{S}$. On the other hand, each FVS in G^{jnt} produces a valid $\mathbb{S}$ by AO. In the case of two policies $\mathrm{W}^+(\mathbb{S})$ is equal to the weight of the corresponding FVS in G^{jnt}. Thus, the algorithm for PSP on two policies finds an optimal FVS in the graphs G^{jnt} and G. □

References

[1] "Qos: Modular qos command-line interface configuration guide, cisco ios," https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/qos_mqc/configuration/15-mt/qos-mqc-15-mt-book/qos-mqc.html.

[2] "Class of service configuration guide," https://www.juniper.net/documentation/en_US/junos11.1/information-products/topic-collections/security/software-all/class-of-service/index.html.

[3] "Configuring modular QoS packet classification," http://www.cisco.com/c/en/us/td/docs/routers/xr12000/software/xr12k_r4-2/qos/configuration/guide/qc42clas.html.

[4] "Cisco xr 12000 series gigabit ethernet line cards," https://www.cisco.com/c/en/us/products/collateral/routers/xr-12000-series-router/product_data_sheet0900aecd803f856f.html.

[5] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. NY: W. H. Freeman & Co., 1979.

[6] G. Even, J. (Seffi) Naor, B. Schieber, and M. Sudan, "Approximating minimum feedback sets and multicuts in directed graphs," *Algorithmica*, vol. 20, no. 2, pp. 151–174, Feb 1998.

[7] C. Demetrescu and I. Finocchi, "Combinatorial algorithms for feedback problems in directed graphs," *Inf. Process. Lett.*, vol. 86, no. 3, pp. 129–136, 2003.

[8] D. E. Foulser, M. Li, and Q. Yang, "Theory and algorithms for plan merging," *Artif. Intell.*, vol. 57, no. 2-3, pp. 143–181, 1992.

[9] T. Jiang and M. Li, *On the approximation of shortest common supersequences and longest common subsequences*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 191–202.

[10] D. E. Taylor and J. S. Turner, "Classbench: a packet classification benchmark," *IEEE/ACM Trans. Netw.*, vol. 15, no. 3, pp. 499–511, 2007.

[11] "New alternatives to optimize policy classifiers," https://github.com/PolicyCompressor/PolicyCompr.

[12] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar, "EffiCuts: optimizing packet classification for memory and throughput," in *SIGCOMM*, 2010, pp. 207–218.

[13] H. Song and J. S. Turner, "ABC: Adaptive binary cuttings for multidimensional packet classification," *IEEE/ACM Trans. Netw.*, vol. 21, no. 1, pp. 98–109, 2013.

[14] S. Dharmapurikar, H. Song, J. S. Turner, and J. W. Lockwood, "Fast packet classification using bloom filters," in *ANCS*, 2006.

[15] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and scalable layer four switching," in *SIGCOMM*, 1998.

[16] A. Bremler-Barr and D. Hendler, "Space-efficient tcam-based classification using gray coding," *IEEE Trans. Computers*, vol. 61, no. 1, pp. 18–30, 2012.

[17] O. Rottenstreich and I. Keslassy, "Worst-case TCAM rule expansion," in *INFOCOM*, 2010.

[18] O. Rottenstreich, I. Keslassy, A. Hassidim, H. Kaplan, and E. Porat, "On finding an optimal TCAM encoding scheme for packet classification," in *INFOCOM*, 2013.

[19] K. Kogan, S. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster, "SAX-PAC (Scalable And eXpressive PAcket Classification)," in *SIGCOMM*, 2014.

[20] K. Kogan, S. I. Nikolenko, P. Eugster, A. Shalimov, and O. Rottenstreich, "FIB efficiency in distributed platforms," in *ICNP*, 2016, pp. 1–10.

[21] P. Chuprikov, K. Kogan, and S. I. Nikolenko, “General ternary bit strings on commodity longest-prefix-match infrastructures,” in *ICNP*, 2017, pp. 1–10.
[22] C. Umans, “The minimum equivalent DNF problem and shortest implicants,” *J. Comput. Syst. Sci.*, vol. 63, no. 4, pp. 597–611, 2001.
[23] K. Kogan, S. Nikolenko, P. Eugster, and E. Ruan, “Strategies for Mitigating TCAM Space Bottlenecks,” in *HOTI*, 2014.
[24] Y. Kanizo, D. Hay, and I. Keslassy, “Palette: Distributing tables in software-defined networks,” in *INFOCOM*, 2013, pp. 545–549.
[25] N. Kang, Z. Liu, J. Rexford, and D. Walker, “Optimizing the ”one big switch” abstraction in software-defined networks,” in *CoNEXT*, 2013, pp. 13–24.
[26] P. Chuprikov, K. Kogan, and S. I. Nikolenko, “How to implement complex policies on existing network infrastructure,” in *SOSR*, 2018, pp. 9:1–9:7.