New Alternatives to Optimize Policy Classifiers

Vitalii Demianiuk, Sergey Nikolenko, Pavel Chuprikov, Kirill Kogan

***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, where each class consists of ternary-bit filters determining a set of matched packet headers. Various vendors already support the notion of classes in policy declarations [2], [3] 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 [4]. In real deployments, the number of classes per policy ranges from tens to hundreds depending on the application model [4]. The size of a class depends on the complexity of represented pattern.

Traditionally, a separate class instance is allocated for each policy instance that contains it (see policies P_1, P_2, and P_3 in Fig. 1a). Each allocated class instance has an attached action specified by the corresponding policy (in Fig. 1a, an action $A_{i,j}$ is attached to the instance of c_j in the policy P_i). 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 [5] 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.

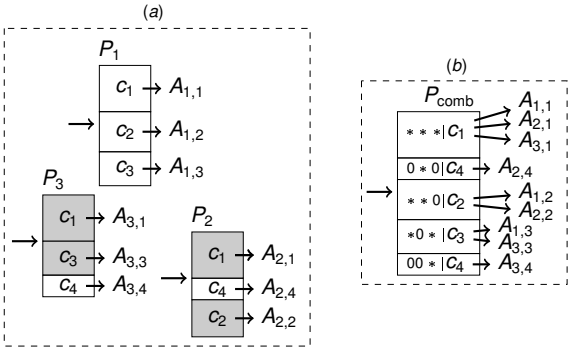


Fig. 1: (a) separate policies P_1, P_2, P_3; (b) a representation P_{comb} that emulates the policies; class instances that have been cut in P_{comb} are shown in gray.

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.

To emulate the classification of an incoming packet header by a policy P_i, the lookup process in P_{comb} should be performed only on class instances corresponding to the classes in P_i. For instance, in Fig. 1b, during the classification by P_3 the first instance of c_4 and c_2 in P_{comb} should be ignored. To achieve this, we prepend filters of every class instance in P_{comb} and incoming headers by special extra bits as described in Section III; e.g., on Fig. 1b the filters in c_3 are prepended by *0*, which indicates that c_3 belongs to P_1 and P_3.

In this work we propose equivalent combined representations for a given set of policies. We show the condition for the existence of *ideal representations* 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 filters in

The preliminary version of this paper entitled "New Alternatives to Optimize Policy Classifiers" has been appeared at ICNP 2018 [1]. The work V. Demianiuk and K. Kogan was partially supported by the Ariel Cyber Innovation Center in cooperation with the Israel National Cyber Directorate in the Prime Minister's Office, and by the Data Science and Artificial Intelligence Research Center at Ariel University. The work of Sergey Nikolenko shown in Sections III, IV, and V (in particular, Theorems 2, 8, 9, 10) has been supported by the Russian Science Foundation grant no. 17-11-01276.

V. Demianiuk and K. Kogan are with Ariel University, 40700 Ariel, Israel (e-mail: vitalii@ariel.ac.il, kirillk@ariel.ac.il). S. Nikolenko is with National Research University Higher School of Economics, 199034 St. Petersburg, Russia, and also with Steklov Institute of Mathematics at St. Petersburg, 191023 St. Petersburg, Russia (e-mail: snikolenko@gmail.com). P. Chuprikov is with IMDEA Networks Institute, 28918, Leganes, Spain, and also with USI Lugano, 6900 Lugano, Switzerland (email: pavel.chuprikov@usi.ch).

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, the lookup time complexity in such a representation is the same as in a single policy of the same size.

The paper is organized as follows. In Section III we explore 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), otherwise, they *intersect*.

To define a *policy* P over a given set of classes $\mathcal{C}$, one needs to select a set of classes $\mathcal{C}_P \subseteq \mathcal{C}$ belonging to P, specify a sequence $\mathbb{S}(P)$ containing each class from $\mathcal{C}_P$ only once, and associate an action with each class in $\mathcal{C}_P$. For an incoming header, the action of a first matched class in $\mathbb{S}(P)$ is returned. If an incoming header is not matched by any class in $\mathbb{S}(P)$ then the policy *default* action is returned. Since classes can intersect, a policy is defined by a sequence rather than a set. Originally, classes were introduced to define common classification patterns [2], [3] 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 $\mathcal{C}_p$. For instance, Figure 2a shows a policy P defined by the sequence $\mathbb{S}(P) = c_3, c_2, c_4, c_1$, which is equivalent to P' defined by $\mathbb{S}(P') = c_3, c_2, c_1, c_4$ since c_1 and c_4 are disjoint.

To define policies that are equivalent to P on $\mathcal{C}_P$, we introduce the following partial order $\prec_P$ of classes in $\mathcal{C}_P$.

(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)

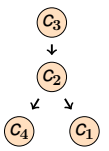


Fig. 2: (a) definition of the policy $\mathbb{S}(P) = c_3, c_2, c_1, c_4$; (b) partial order $\prec_P$: $c_3 \perp c_1, c_3 \perp c_4$, and $c_1 \perp c_4$.

We say that $c_i \prec_P c_j$ if at least one of the following conditions is satisfied:

- c_i intersects with c_j, c_i appears before c_j in $\mathbb{S}(P)$, and the actions assigned to c_i and c_j in P are different (disjointness constraint);
- there is a class $c_k \in P$ such that $c_i \prec_P c_k$ and $c_k \prec_P c_j$ (transitivity of the partial order).

For instance, Fig. 2a defines a policy P whose corresponding partial order $\prec_P$ is illustrated on Fig. 2b. In all illustrations of partial orders we omit arrows showing that $c_i \prec_P c_j$ if c_i precedes c_j in $\prec_P$ that follow from transitivity.

Observation 1. *The policy P' is equivalent to a policy P if $\prec_{P'}$ coincide with $\prec_P$ and each class in P' has the same action as in P (the default actions in P and P' should coincide).*

Informally, if we represent a policy P as a graph G with vertices corresponding to classes of P and edges corresponding to partial order constraints of $\prec_P$, then any topological order on the vertices of G forms a sequence of classes in the policy that is equivalent to P.

We denote by $\mathcal{P} = \{P_1, P_2, \ldots P_{|\mathcal{P}|}\}$ a set of $|\mathcal{P}|$ policies over the same set of classes $\mathcal{C}$; by $|\mathcal{C}|$, the number of classes in $\mathcal{C}$. Note that for an incoming header H, the corresponding policy P is retrieved from internal switch data structures, where the classification of H should be performed; in this case, we say that H is coming in the *context* of the policy P. A *combined policy* P_{comb} representing a group of policies $\mathcal{P}' \subseteq \mathcal{P}$ *emulates* $\mathcal{P}'$ if for any header H and any policy $P_i \in \mathcal{P}'$, the lookup of H in P_i and the lookup of H in P_{comb} in the context of P_i yield the same action. Informally, this means that P_{comb} mimics the behaviour of any policy in $\mathcal{P}'$.

In general, a *representation* $\mathcal{P}_{comb}$ implementing the classification in $\mathcal{P}$ can consist of more than one combined policy. Due to the single lookup constraint, a policy in $\mathcal{P}$ should be represented only by a single combined policy in $\mathcal{P}_{comb}$. Therefore, in order to construct $\mathcal{P}_{comb}$, the policies in $\mathcal{P}$ should be assigned into multiple disjoint groups $\mathcal{P}^1, \mathcal{P}^2, \ldots, \mathcal{P}^m$, where each group $\mathcal{P}^i$ is represented by a separate combined policy P^i_{comb}. By m we denote the number of groups. For every header incoming in the context of a policy $P \in \mathcal{P}$, the lookup is done in P^i_{comb} corresponding to the group $\mathcal{P}^i$ containing P. We say that $\mathcal{P}_{comb}$ *emulates* $\mathcal{P}$ if each $P^i_{\text{comb}} \in \mathcal{P}_{comb}$ emulates the corresponding group of policies $\mathcal{P}^i$.

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 say that a combined representation $\mathcal{P}_{comb}$ of multiple policies $\mathcal{P}$ is ideal if $\mathcal{P}_{comb}$ contains a single instance of every class from $\mathcal{C}$. For a given $\mathcal{P}$, we propose a criteria of the existence of ideal representations (satisfying the single lookup constraint) and explain how to construct them. At this point, we assume that original policies from $\mathcal{P}$ are represented by $\mathcal{P}_{comb}$ consisting of 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 $\mathcal{C}$ do not match the same headers. In this case we can construct an ideal policy P_{comb} that contains all classes from $\mathcal{C}$ in any order. A header H can be looked up in P_{comb} instead of a configured policy P_i, and if the first matched class c belongs to P_i (this can be verified with any set membership data structure), the action of c in P_i is returned. Otherwise, the classification result is the default action in P_i since only a single class in P_{comb} can match H.

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. Unlike the previous case, if the first class in P_{comb} matching a header H incoming in the context of a policy $P_i \in \mathcal{P}$ does not belong to P_i, there is no guarantee that P_{comb} does not contain a class from P_i matching H. Hence, to deal with more general structural 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. To implement this requirement, we prepend each filter of a class c in P_{comb} with the ternary *policy prefix* $\text{pp}(c)$, and each header H incoming in the context of P_i with the binary *header prefix* pp_i satisfying the following property: $\text{pp}(c)$ matches pp_i if and only if $c \in P_i$. Note that both policy and header prefixes have the same length. Such representations allow to match a header incoming in the context of P_i only against the classes in P_{comb} belonging to the original policy P_i. One of the possible variants of $\text{pp}(c)$ and pp_i satisfying the defined above property can be the following: $\text{pp}(c)$ is a ternary string of length $|\mathcal{P}|$ such that $\text{pp}(c)_i = *$ if $c \in P_i$ and $\text{pp}(c)_i = 0$ otherwise; pp_i is a bit string $0 \dots 010 \dots 0$ of length $|\mathcal{P}|$ that has a 1 only at position i.

Fig. 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 2 shows that adding $|\mathcal{P}|$ extra bits per filter in P_{comb} is unavoidable.

Theorem 2. *For any $l > 2$, there exists a set $\mathcal{P}$ of l policies such that $\mathcal{P}$ can be represented by an ideal P_{comb} and the length of prefixes should be at least l in any possible variant $\text{pp}(c)$ and pp_i satisfying that $\text{pp}(c)$ matches pp_i if and only if $c \in P_i$.*

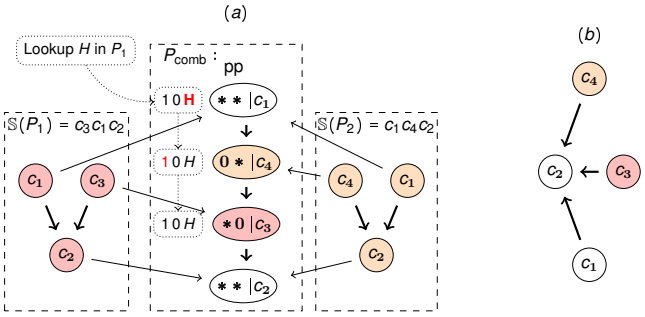


Fig. 3: (a) $\mathcal{P} = \{P_1, P_2\}$ with $\mathcal{C}$ containing non-disjoint classes, ideal P_{comb} and policy prefixes; (b) $G^{\text{jnt}}(\mathcal{P})$.

Proof. Consider a set $\mathcal{P}$ consisting of l policies over a set $\mathcal{C}$ of $2^l - 1$ different classes. A class $c_i \in \mathcal{C}$ belongs to the policy $P_j \in \mathcal{P}$ iff the jth bit in the binary representation of i is 1. E.g., $c_{5=00101_2}$ belongs to the policies P_1 and P_3. In each policy $P \in \mathcal{P}$, a class $c_i \in P$ precedes the class $c_j \in P$ ($c_i \prec_P c_j$) if $i < j$. Hence, for such $\mathcal{P}$ the order of classes in the ideal P_{comb} is unique and is defined by the class indices.

Note that each class c_i in $\mathcal{C}$ belongs to a unique set of policies, hence, there are should be at least 2^l different policy prefixes. Thus, the length of $\text{pp}(c)$ should be at least $\frac{l}{log_2 3} = O(l)$. In the remaining part of the proof, we are to show that the lower bound on the policy prefix length is exactly l.

We say that the kth bit of a policy prefix $\text{pp}(c_i)$ is *aggregating* if $\text{pp}(c_i)_k$ is '$*$' and at least two header prefixes with two different values in the kth bit position are matched by $\text{pp}(c_i)$. We denote by $|\text{pp}(c_i)|_*$ the number of aggregating bits in c_i. To prove the theorem, we are to show that for the set of policies introduced above, $|\text{pp}(c_{2^l-1})|_* \geq l$ in any valid assignment of header and policy prefixes.

Consider a sequence of classes $c_7, c_{15}, \dots, c_{2^i-1}, \dots, c_{2^l-1}$. The set of header prefixes matched by $\text{pp}(c_{2^i-1})$ consists of the header prefixes matched by $\text{pp}(c_{2^{i-1}-1})$ and the header prefix pp_i. Thus, $\text{pp}(c_{2^i-1})$ contains aggregating bits at the same bit positions as $\text{pp}(c_{2^i-1})$ and in at least one additional position. Hence, $|\text{pp}(c_{2^i-1})|_* > |\text{pp}(c_{2^{i-1}-1})|_*$ and $|\text{pp}(c_{2^l-1})|_* \geq |\text{pp}(c_7)|_* + l - 3$.

To finish the proof, it suffices to show that $|\text{pp}(c_7)|_* \geq 3$. Consider the policy prefixes $\text{pp}(c_1), \text{pp}(c_2), \text{pp}(c_4)$ of the classes $c_1 \in P_1, c_2 \in P_2, c_4 \in P_3$, where each one of them belongs to a single policy. These prefixes do not contain aggregating bits and at least two of them differs in at least two positions not containing $*$. W.l.o.g., we can assume that $\text{pp}(c_1), \text{pp}(c_4)$ differ in at least two bit positions not containing '$*$', hence, $\text{pp}(c_5)$ ($c_5 \in P_1, P_2$) has two aggregating bits at these positions. The policy prefix $\text{pp}(c_7)$ matches the same header prefixes as $\text{pp}(c_5)$ and the prefix pp_2. Thus, $|\text{pp}(c_7)|_* \geq |\text{pp}(c_5)|_* + 1 \geq 3$ and $|\text{pp}(c_{2^l-1})|_* \geq l$. □

C. When are representations ideal?

In this part we formulate the condition 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 3. *For a given set of policies $\mathcal{P}$, there exists an ideal P_{comb} if the corresponding G^{jnt} is acyclic.*

Proof. 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 policy 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'. □

If G^{jnt} contains a cycle, then any possible linear ordering of classes in P_{comb} contradicts the order of classes in some policy P_i, i.e., for each P_{comb} there exists $c, c' \in P_i$ such that $c \prec_{P_i} c'$ but c' appears before c in P_{comb}. In this case, the ideal representation P_{comb} is correct only if for each pair of classes c, c' in P_{comb} contradicting $\prec_{P_i}$, each header belonging to the intersection of c and c' is matched by a class preceding c and c' in $S(P_i)$. Such property of P_{comb} appears rarely on practice and can not be verified in a polynomial time. Hence, we suggest to use the acyclicity of G^{jnt} as a criteria of the ideal representation existence.

The proof of Theorem 3 implies an algorithm that constructs an ideal representation if it exists. The time complexity of this algorithm equals to the time $T_{G^{\text{jnt}}}(\mathcal{P}) = O(\sum_{P \in \mathcal{P}} \left(|P|^2 + |P| \cdot D(P) \right))$ needed to construct G^{jnt}, where $D(P)$ is the number of intersecting class pairs from P that have different attached actions.

D. Multiple combined policies

So far we have assumed that a given set of policies $\mathcal{P}$ is represented by $\mathcal{P}_{comb}$ consisting of a single P_{comb}. In this subsection, we consider $\mathcal{P}_{comb}$ consisting of m combined policies $P^1_{\text{comb}}, P^2_{\text{comb}}, \ldots, P^m_{\text{comb}}$.

Theorem 4. *For a given set of policies $\mathcal{P}$, if there exists an ideal representation $\mathcal{P}_{comb}$ consisting of multiple combined policies, then there exists an ideal representation of $\mathcal{P}$ consisting of a single combined policy P_{comb}.*

Proof. To prove the theorem, we construct an ideal P_{comb} from the ideal $\mathcal{P}_{comb}$ in the following way: to obtain the sequence $\mathbb{S}$ of classes in P_{comb} we concatenate all class sequences defining combined policies in $\mathcal{P}_{comb}$; then we prepend class instances in $\mathbb{S}$ by policy prefixes as described in Section III-B. Since each class in $\mathcal{C}$ appears only in a single combined policy of $\mathcal{P}_{comb}$, the constructed P_{comb} is an ideal representation. □

Multiple combined policies in $\mathcal{P}_{comb}$ can reduce two different characteristics:

- $plen(\mathcal{P}_{comb})$, the total length of all extra prefixes in $\mathcal{P}_{comb}$ (in particular, if all class instances in $P^i_{\text{comb}} \in \mathcal{P}_{comb}$ are disjoint then extra prefixes for P^i_{comb} are unnecessary);

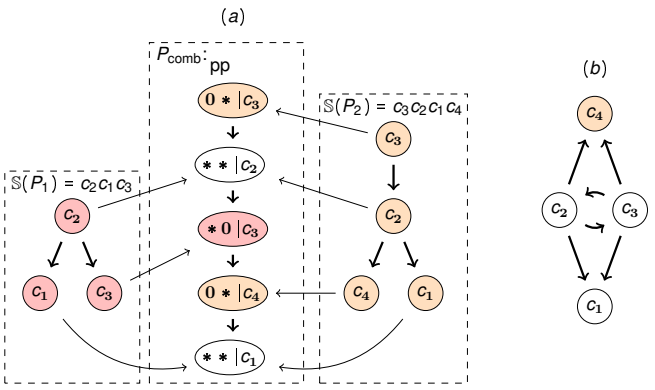


Fig. 4: (a) $\mathcal{P} = \{P_1, P_2\}$ and P_{comb} with duplicated c_3; (b) $G^{\text{jnt}}(\mathcal{P})$.

- $lsize(\mathcal{P}_{comb})$, the maximum number of filters involved in a single lookup (which can optimize lookup energy and time).

By the definition of an ideal $\mathcal{P}_{comb}$, the corresponding partition $\mathcal{P}^1, \mathcal{P}^2, \ldots, \mathcal{P}^m$ of $\mathcal{P}$ into m disjoint groups satisfies the following *sharing* condition: if P_i, P_j share a class then P_i and P_j belong to the same group. The following observation immediately follows from Theorems 3 and 4.

Observation 5. *If there exists an ideal representation of a given set of policies $\mathcal{P}$, then any partition of $\mathcal{P}$ satisfying the sharing condition defines an ideal $\mathcal{P}_{comb}$ emulating $\mathcal{P}$.*

Let $\mathbf{P}_{max}$ be a partition of $\mathcal{P}$ satisfying the sharing condition with a maximal number of groups.

Theorem 6. *If there exists an ideal representation of $\mathcal{P}$, then $\mathbf{P}_{max}$ defines an ideal $\mathcal{P}_{comb}$ with the minimum values of $plen(\mathcal{P}_{comb})$ and $lsize(\mathcal{P}_{comb})$ among all ideal representations.*

Proof. By definition of the sharing condition, $\mathbf{P}_{max}$ is unique and any partition of $\mathcal{P}$ satisfying the sharing condition can be obtained from $\mathbf{P}_{max}$ by merging the groups in $\mathbf{P}_{max}$. Since merging groups can only increase the values of $plen(\mathcal{P}_{comb})$ and $lsize(\mathcal{P}_{comb})$, the theorem immediately follows. □

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 $\mathbb{S}$ defining the order of class instances in P_{comb} 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 c_j, c_t in P_i, if $c_j \prec_{P_i} c_t$ then c_j appears before c_t in $\mathbb{S}'$. Only instances of classes from this subsequence participate in the classification by policy P_i, i.e., in the corresponding P_{comb} 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 7. *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 8. *PSP is NP-hard even for two policies, $|\mathcal{P}| = 2$.*

The proof can be found in the appendix. In the following, we denote by $|\mathbb{S}|$ a number of class instances in $\mathbb{S}$.

C. Optimal solution for PSP problem

If the partial orders of all policies in $\mathcal{P}$ are linear, the PSP problem is a weighted version of the classical Shortest Common Supersequence (SCS) problem [6]. For a set of strings, SCS finds a string with a minimal total length containing all these strings as subsequences. The algorithm in [6] finds an optimal solution for the weighted version of SCS in $O(|\mathcal{C}|^{|\mathcal{P}|})$ time, hence, this algorithm can find an optimal solution of the PSP problem but only if the number of policies is relatively small and the partial orders of all policies in $\mathcal{P}$ are linear.

D. Multiple combined policies again

Similar to ideal representations, $\mathcal{P}_{comb}$ consisting of multiple combined policies does not reduce the number of maintained filters compared to a representation consisting of a single P_{comb}. Note that the benefits from having multiple combined policies $\mathcal{P}_{comb}$ described in Section III-D remain the same for non-ideal representations. For instance, the lengths of policy prefixes can affect the memory requirements of a resulting $\mathcal{P}_{comb}$; to incorporate them into the final objective, we can minimize the total number of ternary bits in all maintained filters (including policy prefix bits). Also, representations consisting of multiple policies allow to reduce lookup complexity that can be incorporated by bounding the number of filters in each combined policy in $\mathcal{P}_{comb}$. The partition of $\mathcal{P}$ into multiple groups addresses two fundamental tradeoffs: (1) the total length of policy prefixes versus the number of filters in duplicated class instances; and (2) the maximum number of filters in each combined policy versus the number of filters in duplicated class instances.

Recall that any $\mathcal{P}_{comb}$ is obtained from a partition of $\mathcal{P}$ into multiple groups, where each group is represented in $\mathcal{P}_{comb}$ by a single combined policy. For a fixed partition of $\mathcal{P}$ into groups, the minimization of memory requirements (in bits) is obtained from the minimization of the number of entries in each combined policy since the lengths of the proposed policy prefixes depends only on the number of policies in every group. In the case of lookup complexity, the situation is similar. Hence, the construction of a combined policy for each group can be done by the proposed methods described below. We leave for the future study the development of methods partitioning $\mathcal{P}$ into multiple groups addressing both tradeoffs.

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 [7], 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 [8] proposes an algorithm for WFVS with approximation factor $O(\log|V|\log\log|V|)$, but there are other alternatives [9]. In what follows we denote by $\alpha(G)$ the approximation factor of an algorithm for the $WFVS$ problem on a graph G. Note that WFVS is not harder than PSP (see the proof of Theorem 8). This is why the algorithms proposed below exploit WFVS as a building block.

B. Algorithm AllOrOne

By Theorem 3 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 $c \in V^{\mathsf{wfvs}}$, 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

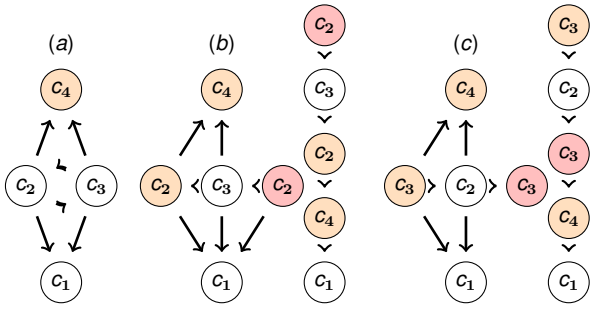


Fig. 5: (a) G^{jnt}; (b)-(c) two solutions depending on the values of $w(c_3)$ and $w(c_2)$.

Algorithm 1 AO($\mathcal{P}$)

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

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 9). Since G^* can be constructed in at most $T_{G^{\text{jnt}}}(\mathcal{P})$ time, the running time of AO is $T_{\text{FVS}}(G^{\text{jnt}}) + T_{G^{\text{jnt}}}(\mathcal{P})$, where $T_{\text{FVS}}(G)$ is the running time of the algorithm for the WFVS problem.

Theorem 9. 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.

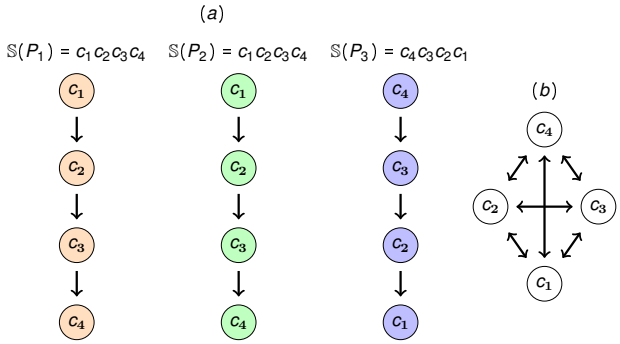


Fig. 6: PSP instance with $|\mathcal{P}| = 3$ policies showing the lower bound of AO: (a) the input $\mathcal{P} = P_1, P_2, P_3$; (b) the graph G^{jnt}.

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 $w(c_2) \leq w(c_3)$ 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 10. AO *has an approximation factor at most* $(|\mathcal{P}| - 1) \cdot \alpha(G^{\text{jnt}})$.

Proof. Recall that in AO the found V^{wfvs} defines classes appearing more than once in the resulting $\mathbb{S}$, therefore, $\text{W}^+(\mathbb{S}) \leq (|\mathcal{P}| - 1) \cdot W(V^{\text{wfvs}})$. Note that $W(V^{\text{wfvs}}) \leq \alpha(G^{\text{jnt}}) W(V^{\text{wfvs}}_{\text{OPT}})$, where $V^{\text{wfvs}}_{\text{OPT}}$ is FVS with the minimal total weight. Thus, $\text{W}^+(\mathbb{S}) \leq (|\mathcal{P}| - 1) \cdot \alpha(G^{\text{jnt}}) W(V^{\text{wfvs}}_{\text{OPT}})$.

To finish the proof it suffices to show that $W(V^{\text{wfvs}}_{\text{OPT}})$ is less than $\text{W}^+(\mathbb{S}_{opt})$ for an optimal solution $\mathbb{S}_{opt}$ of the PSP problem. In any solution of the PSP problem, classes appearing more than once form a FVS in the graph G^{jnt}. The value of $W(V^{\text{wfvs}}_{\text{OPT}})$ is less than $W(V_{\mathbb{S}_{\text{OPT}}})$, where $V_{\mathbb{S}_{\text{OPT}}}$ is a FVS corresponding to $\mathbb{S}_{opt}$. Thus, $W(V^{\text{wfvs}}_{\text{OPT}}) \leq W(V_{\mathbb{S}_{\text{OPT}}}) \leq \text{W}^+(\mathbb{S}_{\text{OPT}})$. □

Theorem 11. *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 $c_1c_2 \ldots c_n$; the partial order of the last policy is also linear but contains the same classes in the reversed order $c_nc_{n-1} \ldots c_1$ (see Fig. 6a). In the corresponding graph G^{jnt} each pair of vertices is connected by two edges with different directions (see Fig. 6b). Any feedback vertex set of G^{jnt} consists of $(n - 1)$ vertices, therefore, the total overhead $\text{W}^+(\mathbb{S})$ incurred by AO is equal to $(n - 1)(l - 1)$. For an optimal solution $\mathbb{S}_{\text{OPT}} = c_1c_2 \ldots c_n \ldots c_2c_1$, the overhead is equal to $\text{W}^+(\mathbb{S}_{\text{OPT}}) = n - 1$. □

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 limiting the optimization capabilities of the algorithm. In the proof of Theorem 11, 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-D. For the PSP instance in the proof of Theorem 11 these optimizations

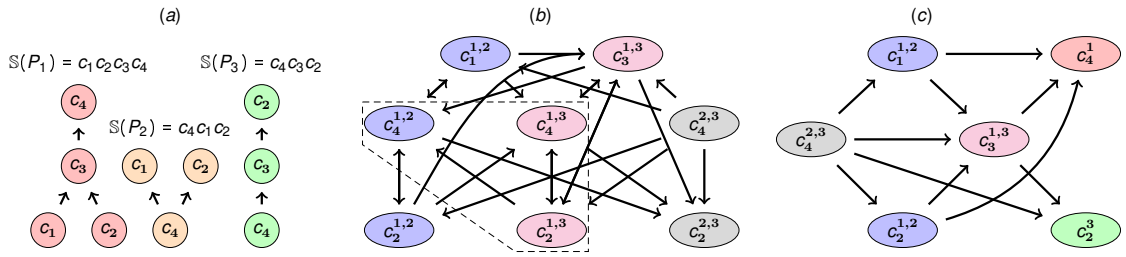


Fig. 7: Illustration of the CS algorithm: (a) the input $\mathcal{P} = \{P_1, P_2, P_3\}$; (b)$G^{\text{pair}}$; the dashed line encloses V^{wfvs}; (c) G^*.

Algorithm 2 CS($\mathcal{P}$)

1: construct the graph $G^{\text{pair}}(P_1, \ldots, P_l)$;
2: $V^{\text{wfvs}} \leftarrow WFVS(G^{\text{pair}})$, with vertex weights $w(c) = |c|$;
3: **for** every $c \in \mathcal{C}$ **do**
4: $\quad \mathbb{P}_c \leftarrow$ 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** $\mathbb{S}$.

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-C we will introduce algorithms based on alternative principles that do not require unnecessary constraints.

In the following theorem we show the inapproximability of PSP by reduction from the SCS problem.

Theorem 12. *Unless* P = NP*, there is no polynomial algorithm for the PSP problem with a constant approximation factor on* $\mathrm{W}^+(\mathbb{S})$.

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 [10]. 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 [10] 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 12, 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.

C. Algorithm CLIQUESHARE

The efficiency of an algorithm based on WFVS heavily depends on the information about $\mathbb{S}$ provided by FVS. In the AO algorithm this information is very limited: FVS only provides the set of classes appearing in $\mathbb{S}$ more than once. To overcome this limitation, we propose another algorithm CLIQUESHARE (CS): for each class c, FVS in CS provides the pairs of policies containing c that are not sharing the same instance of c in $\mathbb{S}$.

In the CS algorithm we construct another graph G^{pair} allowing to operate with a finer resolution. Denote by $\mathcal{P}_c$ the 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 $A, A' \subset \mathcal{P}$ containing P_i (e.g., Fig. 7b shows a G^{pair} graph for the input $\mathcal{P}$ shown on Fig. 7a).

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 $P_i, P_j \in A$. A 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. 7a). The weights of all classes are the same. At first CS constructs G^{pair} (see Fig. 7b), 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 $V^{\mathsf{wfvs}} = \{c_2^{1,3}, c_4^{1,2}, c_4^{1,3}\}$. 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. 7c). 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 11. In the following we prove that CS works correctly and estimate its approximation factor.

Theorem 13. CS *correctly solves the PSP problem.*

Proof. Similar to Theorem 9, 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 9. 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 14. 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, which is not possible for partitions constructed by CS. Therefore, $\mathrm{W}^+(\mathbb{S}) \le W(V^{\mathsf{wfvs}})$.

By any solution $\mathbb{S}'$ of the PSP problem we can construct an FVS in G^{pair} in the following way: if P_i and P_j do not share an instance of c in $\mathbb{S}'$, $c \in P_i, P_j$, then $c^{\{P_i, P_j\}}$ is in FVS.

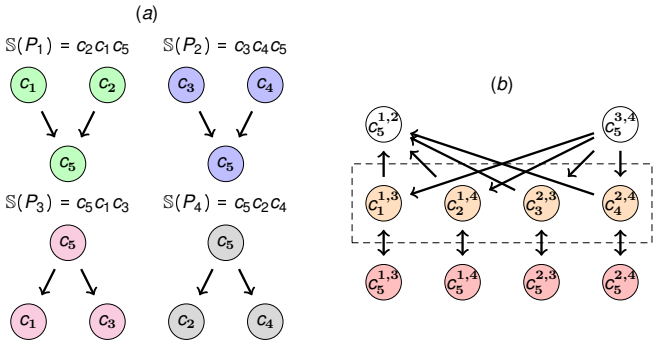


Fig. 8: The PSP instance with $|\mathcal{P}| = 4$ policies showing the lower bound of CS : (a) the input $\mathcal{P} = P_1, P_2, P_3, P_4$; (b) the graph G^{pair}; the dashed line encloses V^{wfvs}.

Hence, as in Theorem 10, $W(V^{\mathsf{wfvs}}) \le \alpha(G^{pair}) W(V^{\mathsf{wfvs}}_{\mathsf{OPT}}) \le \alpha(G^{pair}) W(V_{\mathbb{S}_{\mathsf{OPT}}})$, where $V^{\mathsf{wfvs}}_{\mathsf{OPT}}$ is FVS in G^{pair} with the minimal total weight and $V_{\mathbb{S}_{\mathsf{OPT}}}$ is FVS in G^{pair} 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 $N_c = \frac{1}{2}\left(l_c^2 - \sum_{i=1}^{t} s_{c,i}^2\right)$. It can be shown that for any partition on admissible subsets, $N_c \le \left\lfloor \frac{|\mathcal{P}|^2}{4} \right\rfloor \cdot (t-1)$ [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 $W(V_{\mathbb{S}_{\mathsf{OPT}}}) \le \lfloor \frac{|\mathcal{P}|^2}{4} \rfloor W^+(\mathbb{S})$. □

Theorem 15. *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 $c_k \succ_{P_j} c_n$. For instance, on Fig. 8a the class c_3 corresponding to the third pair of policies (P_2, P_3) precedes c_5 in P_2 and succeeds c_5 in P_3. The number of filters in classes is as follows: $|c_i| = x, i = 1 \dots n-1, |c_n| = x+1$.

A graph G^{pair} consists of three categories of vertices (see Fig. 8b): (1) the $n-1$ vertices corresponding to a class c_n in the enumerated pairs of policies (in Fig. 8b, these vertices are shown in purple); (2) the $n-1$ vertices for all other classes c_i, where $i < n$; (in Fig. 8b, these vertices are orange); (3) the vertices corresponding to c_n in non-enumerated pairs of policies which do not affect acyclicity of G^{pair}. (in Fig. 8b, these vertices are white). The G^{pair} graph contains $n-1$ bidirectional edges

[1] The proof of this inequality consists of only cumbersome calculations so we omit it.

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 $\mathbb{S}_{\mathsf{OPT}} = c_n c_1 c_2 \dots c_n$ 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 graph G^{pair} can be constructed in time $T_{G^{\mathsf{jnt}}}(\mathcal{P}) + O(|\mathcal{P}|^2 \cdot \sum_{P\in\mathcal{P}} |P|^2)$. Hence, the running time of CS is $T_{G^{\mathsf{jnt}}}(\mathcal{P}) + O(|\mathcal{P}|^2 \cdot \sum_{P\in\mathcal{P}} |P|^2) + T_{FVS}(G^{\mathsf{pair}}) + O(|\mathcal{C}| \cdot T_{\mathrm{part}}(|\mathcal{P}|))$, 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.

Note that AO and CS find optimal solutions in 'simplest' PSP instances. For example, AO and CS always find optimal solutions if there exists an ideal representation or if an optimal solution for two policies contains at most one duplicated class instance. Various heuristics that are not based on FVS cannot guarantee solution optimality even in these simplest PSP instances. Unfortunately, AO has its own constraints limiting optimization capabilities. The proposed CS overcomes these limitations preserving AO advantages.

D. Additional optimizations

Both AO and CS algorithms can be further improved by additional optimizations. First, we define GREEDYGLUING (GG) optimization that greedily shrink pairs of vertices of an acyclic graph G with the maximal possible sum of weights while G remains acyclic. GG can be added to both AO and CS as the penultimate step, to simplify G^* before taking its topological order. The time complexity of GG is $O(n^3)$, where n is the number of vertices in the corresponding graph.

Another optimization procedure 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 LOCALDESCENT (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 $O(|\mathbb{S}| + \sum_{P\in\mathcal{P}}(|P| + D(P)))$. 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) $\mathrm{delete}(P, c)$, remove a class c from a policy P; (2) $\mathrm{insert}(P, c, c_{succ})$, add a class c to a policy P such that c appears in $\mathbb{S}(P)$ just before c_{succ}. Each insert/delete operation modifies a sequence of classes $\mathbb{S}$ that represents the corresponding P_{comb}. Note that after each operation policy prefixes should be updated assuring that P_{comb} emulates $\mathcal{P}$.

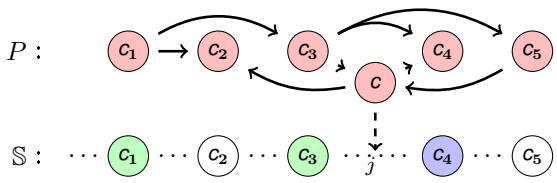


Fig. 9: Insert of an instance for a new class $c \in P$ into the j-th position of $\mathbb{S}$; the white instances are not in $\mathbb{L}_j$ or $\mathbb{R}_j$

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.

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 (DELETE() in Algorithm 3).

The case of an insert operation is more complicated. Let c be the class that is inserted into policy P and let $\mathcal{C}_{\mathrm{prec}}$ be the set of classes preceding c in $\prec_P$, and $\mathcal{C}_{\mathrm{succ}}$ be the set of classes succeeding c in $\prec_P$. If $\mathbb{S}$ is already compatible with the new P the insertion is done. Otherwise, to make $\mathbb{S}$ compatible with the new P, we insert to the jth position inside $\mathbb{S}$ an instance of c and instances of some classes in $\mathcal{C}_{\mathrm{prec}}$ and $\mathcal{C}_{\mathrm{succ}}$ (INSERT_AT() in Algorithm 3), where the jth position is selected to minimize the total number of filters in inserted instances.

Let $\mathbb{L}_j$ be the longest subsequence of $\mathbb{S}[0 \dots j-1]$ satisfying the following conditions:

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

Similarly, we define a subsequence $\mathbb{R}_j$ in the suffix of $\mathbb{S}$ starting from the j-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, $c_1 \prec_P c_2$, the instance of c_2 appears in $\mathbb{R}_j$ after c_1. We insert into the jth position an c instance, class instances in $\mathcal{C}_{\mathrm{prec}}$ that are not in $\mathbb{L}_j$ and class instances in $\mathcal{C}_{\mathrm{succ}}$ that are not in $\mathbb{R}_j$ in order satisfying $\prec_P$. Figure 9 illustrates this insertion procedure for a policy P: $\mathbb{L}_j = c_1 c_3$, $\mathbb{R}_j = c_4$, and class instances c_2, c_5 are inserted together with the instance of c.

Theorem 16. *All class instances inserted to the jth position of $\mathbb{S}$ make $\mathbb{S}$ compatible with a new P.*

Proof. Let $\mathbb{I}_j$ be an inserted sequence of class instances to

Algorithm 3 Dynamic Updates

1: **procedure** DELETE($\mathbb{S}, P, c$)
2: remove from $\mathbb{S}$ an instance of c corresponding only to P
3: remove c from P
4: $\mathbb{S} \leftarrow \mathrm{LD}(\mathbb{S})$
5: **end procedure**

6: **procedure** INSERT_AT($\mathbb{S}, P, c, j$)
7: $insert\_set \leftarrow (\mathcal{C}_{\mathrm{prec}} \setminus \mathbb{L}_j) \cup \{c\} \cup (\mathcal{C}_{\mathrm{succ}} \setminus \mathbb{R}_j)$
8: $\mathbb{I} \leftarrow$ sequence of classes from $insert\_set$ ordered by $\prec_P$
9: insert $\mathbb{I}$ into $\mathbb{S}$ at jth position
10: **if** $j > 0$ **and** $\mathbb{S}[j-1] = c$ **then**
11: remove $j-1$th element of $\mathbb{S}$
12: **end procedure**

13: **procedure** CALCULATE_$\mathcal{L}(P)$
14: $\mathcal{L}[0] = tempL = 0$
15: $\mathbb{L} = \{\}$
16: **for** each $c \in P$ **do**
17: $d(c) \leftarrow$ number of c_1 such that $c_1 \prec_P c$
18: **for** $j \in [0, |\mathbb{S}| - 1]$ **do**
19: **if** $d(\mathbb{S}[j]) = 0$ **and** $\mathbb{S}[j] \notin P$ **then**
20: $\mathbb{L} = \mathbb{L} \cup \{\mathbb{S}[j]\}$
21: **if** $c \in \mathcal{C}_{\mathrm{prec}}$ **then** $tempL = tempL + |\mathbb{S}[j]|$
22: **for** all c_1 such that $\mathbb{S}[j] \prec_P c_1$ **do**
23: $d(c_1) = d(c_1) - 1$
24: $\mathcal{L}[j+1] = tempL$
▷ After jth iteration $\mathbb{L}$ denotes $\mathbb{L}_j$
25: **return** $\mathcal{L}$
26: **end procedure**

27: **procedure** INSERT(P, c, c_{succ})
28: update $\mathbb{S}(P)$ by adding c immediately before c_{succ}
29: **if** $\mathbb{S}$ is compatible to P **then exit**
30: $\mathcal{L} \leftarrow$ CALCULATE_$\mathcal{L}(P)$
31: $\mathcal{R} \leftarrow$ CALCULATE_$\mathcal{R}(P)$
▷ CALCULATE_$\mathcal{R}$ is symmetric to CALCULATE_$\mathcal{L}$
32: $j \leftarrow \arg\max_{0 \le j \le n}(\mathcal{L}[j] + \mathcal{R}[j] + \delta(\mathbb{S}, j, c) \cdot |c|)$
33: INSERT_AT($\mathbb{S}, P, c, j$)
34: $\mathbb{S} \leftarrow \mathrm{LD}(\mathbb{S})$
35: **end procedure**

the j-th position in $\mathbb{S}$ (line 8 in Algorithm 3). 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 one instance of each class from the new P and satisfies $\prec_P$. □

Corollary 1. *An insert operation is correct.*

When $j > 0$ and $\mathbb{S}[j-1]$ is an instance of an inserted class c, we can remove the $(j-1)$th element of $\mathbb{S}$ since a new copy of c is inserted (line 11 in Algorithm 3).

Denote by $\mathcal{I}$ an array of $|\mathbb{S}| + 1$ integeres where $\mathcal{I}[j]$ is a total number of filters in all inserted class instances in case when c is inserted into the jth position of $\mathbb{S}$. Let $\delta(\mathbb{S}, j, c)$ be a function such that $\delta(\mathbb{S}, j, c) = 1$ if $j > 0$ and $\mathbb{S}[j-1] = c$, otherwise, $\delta(\mathbb{S}, j, c) = 0$. Then:

$$\mathcal{I}_j = (1 - \delta(\mathbb{S}, j, c)) \cdot |c| + \sum_{c' \in \mathcal{C}_{\mathrm{prec}} \setminus \mathbb{L}_j} |c'| + \sum_{c' \in \mathcal{C}_{\mathrm{succ}} \setminus \mathbb{R}_j} |c'|$$

Among all potential positions to insert a class c, we choose one minimizing $\mathcal{I}[j]$. Let $\mathcal{L}$ and $\mathcal{R}$ be arrays such that $\mathcal{L}[j] = \sum_{c' \in \mathcal{C}_{\mathrm{prec}} \cap \mathbb{L}_j} |c'|$ and $\mathcal{R}[j] = \sum_{c' \in \mathcal{C}_{\mathrm{succ}} \cap \mathbb{R}_j} |c'|$; then the expression for $\mathcal{I}[j]$ can be rewritten as

$$\mathcal{I}_j = |c| + \sum_{c' \in \mathcal{C}_{\mathrm{prec}} \cup \mathcal{C}_{\mathrm{succ}}} |c'| - (\mathcal{L}_j + \mathcal{R}_j + \delta(\mathbb{S}, j, c) \cdot |c|)$$

We compute elements of $\mathcal{L}, \mathcal{R}$ and then select insertion position j maximizing value of $\mathcal{L}[j] + \mathcal{R}[j] + \delta(\mathbb{S}, j, c) \cdot |c|$ (INSERT() in Algorithm 3).

We calculate elements of $\mathcal{L}$ in the order of increasing j (CALCULATE_$\mathcal{L}$() in Algorithm 3). During this computation, we maintain the sets $\mathbb{L}_j$ and $\mathbb{L}_j \cap \mathcal{C}_{\mathrm{prec}}$. If $\mathbb{S}[j] \notin \mathbb{L}_j$ and all classes preceding $\mathbb{S}[j]$ in P belong to $\mathbb{L}_j$ then $\mathbb{L}_{j+1} = \mathbb{L}_j \cup \{\mathbb{S}[j]\}$, otherwise $\mathbb{L}_{j+1} = \mathbb{L}_j$. Therefore, we can compute $\mathbb{L}_{j+1}$ (and $\mathbb{L}_{j+1} \cap \mathcal{C}_{\mathrm{prec}}$ with $\mathcal{L}[j+1]$ respectively) from $\mathbb{L}_j$ in time proportional to the number of classes preceding $\mathbb{S}[j]$ in P. To speed up this process, when adding $\mathbb{S}[j]$ to $\mathbb{L}_{j+1}$ we look at all classes that succeed $\mathbb{S}[j]$ and mark those for which all classes preceding them in P are in $\mathbb{L}_{j+1}$ (lines 22-23 in Algorithm 3, class c_1 becomes marked if $d(c_1) = 0$). For a subsequent position j', we add $\mathbb{S}[j']$ into $\mathbb{L}_{j'+1}$ if and only if $\mathbb{S}[j']$ is not in $\mathbb{L}_{j'}$ and $\mathbb{S}[j']$ corresponds to a marked class (line 19 in Algorithm 3). This implementation allows to compute all elements of $\mathcal{L}$ in time $O(|\mathbb{S}| + |P| + D(P))$.

The values of $\mathcal{R}$ elements can be computed in the reverse order of j in a similar way. Therefore, the total time complexity of the insert operation equals $O(|\mathbb{S}| + |P| + D(P))$.

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

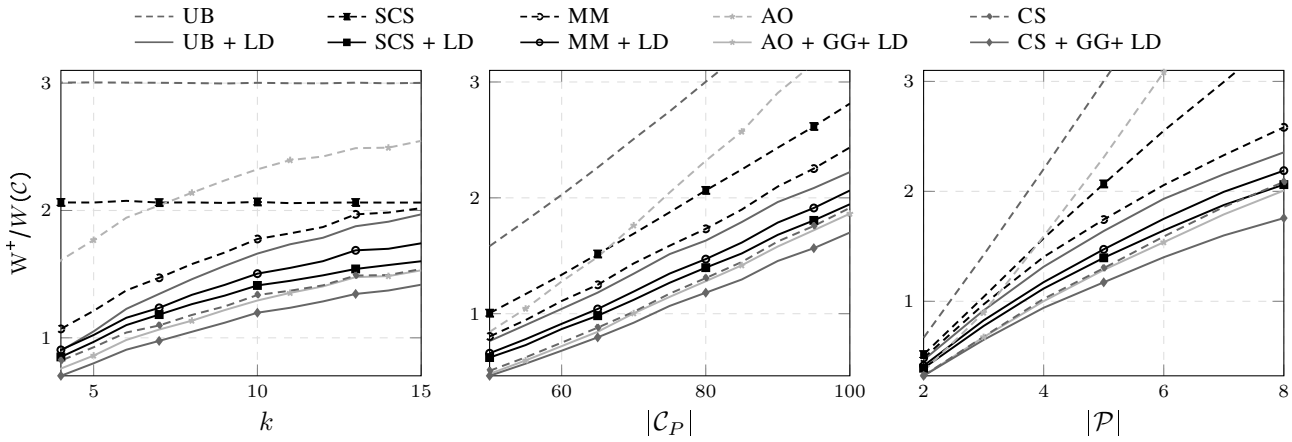
VII. Experimental evaluation

A. Combined representations

Algorithms. We compare the algorithms AO, CS, weighted SCS, UpperBound (UB) that simply concatenate all $\mathbb{S}(P_i)$, $P_i \in \mathcal{P}$, into a single $\mathbb{S}$, and MajorityMerge (MM) proposed in [6] with $(3,1)$-look-ahead extensions [11], [12] (see Algorithm 4). We have also extended the algorithms with the GG heuristic and LD. For each considered algorithm we also evaluate its version extended by additional optimizations. In particular, we extend all algorithms by LD, and additionally extend graph-based algorithms AO and CS by GG.

Methodology. Unfortunately, de-facto standard frameworks to generate filter-based classifiers such as ClassBench [13] do not have class-based level of abstraction and, hence, cannot be used for the declaration of multiple policies based on the same set of classes $\mathcal{C}$. 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



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

Fig. 10: 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 $|\mathcal{P}|$

Algorithm 4 MM($\mathcal{P}$)

- $\mathcal{F}(c, \mathcal{P})$ – set of policies containing every policy P s.t. $c \in P$ and there is no class c' preceeding c in $\prec_P$
- $\mathcal{P}/c$ – instance of the PSP problem with c removed from policies in $\mathcal{F}(c, \mathcal{P})$

1: find (c_1, c_2, c_3) maximizing
$$(|\mathcal{F}(c_1, \mathcal{P})| - 1) \cdot w(\mathcal{P}, c_1) + (|\mathcal{F}(c_2, \mathcal{P} \setminus c_1)| - 1) \cdot w(c_2) + (|\mathcal{F}(c_3, (\mathcal{P} \setminus c_1) \setminus c_2| - 1) \cdot w(c_3),$$
breaking ties by $(|\mathcal{F}(c_1, \mathcal{P})| - 1) \cdot w(c_1)$;
2: **if** $\mathcal{P} \setminus c_1 = \varnothing$ **then return** c_1 ▷ return c_1 if $\mathcal{P}/c_1$ empty
3: $\mathbb{S} = c_1 \mathrm{MM}(\mathcal{P} \setminus c_1)$ ▷ concat c_1 with the recursive result
4: **return** $\mathbb{S}$

For simplicity, we assume that each class in every policy is associated with a different action.

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 [14], and the results are summarized on Fig. 10. 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.

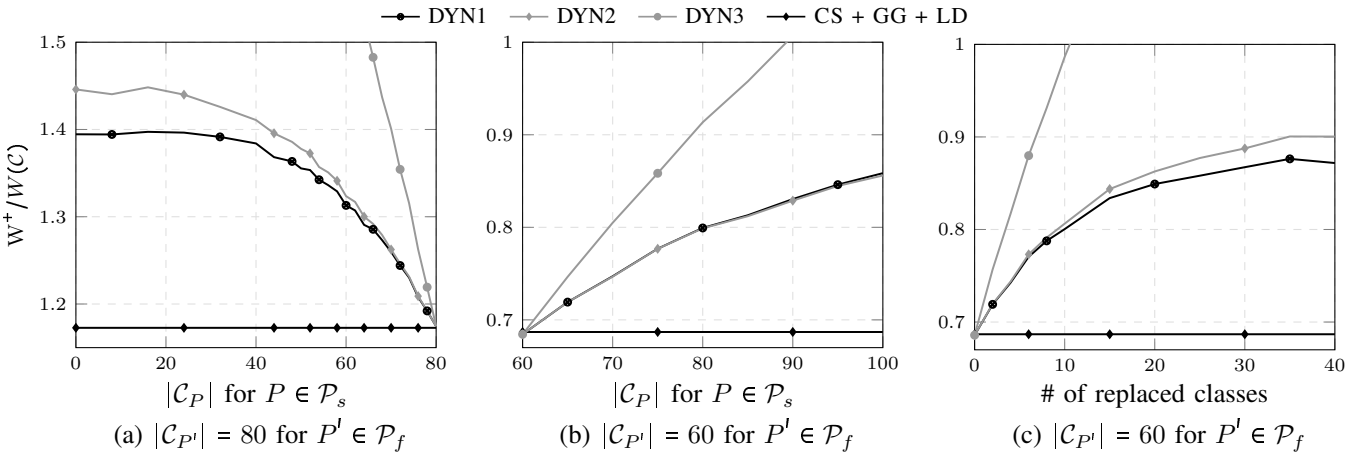
The number of filters in a combined representation significantly depends on the structure of given policies. There are three main characteristics of the input structure: (1) number of intersecting classes, (2) average number of policies that contain a class, (3) total number of policies. We generate inputs with different values of these characteristics. Fig 10a shows how the relative overhead grows as the average number of classes k intersecting with each $c \in \mathcal{C}$ increases. In Fig. 10b 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. The inputs in Fig. 10c consist of various numbers of policies.

The algorithm CS with GG and 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 GG and LD constructs a representation with only 20-50% of the filters 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.

Additional optimizations: The evaluations show that class sharing introduces substantial savings, and changes the linear behavior to nearly logarithmic in Fig. 10.c; even UB with LD reduces the overhead for additional instances of classes (e.g., by 23-63% in Fig. 10a); still it is worse than the other considered algorithms. Additional optimizations are especially effective for UB, SCS, and AO algorithms saving up to 52%, 38%, and 47% respectively in the second set of experiments (see Fig. 10b); CS can be also improved by additional optimizations, but in this case its effect is not substantial (at most 17% in all experiments) since produced results are close to local minimum. Comparing SCS with and without LD in Fig. 10a 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. 10a); this is due to optimality of SCS in the case of linear orders.

AO: In the third set of experiments (see Fig. 10.c), AO without optimizations outperforms UB without optimizations by up to 45% (see Fig. 10.b) but performs up to 1.7 times worse than CS without optimizations. In the first and second set of experiments, AO without optimization always performs worse than MM without optimizations (see Fig. 10.a-b). 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, CS without optimizations constructs



(a) $|\mathcal{C}_{P'}| = 80$ for $P' \in \mathcal{P}_f$ (b) $|\mathcal{C}_{P'}| = 60$ for $P' \in \mathcal{P}_f$ (c) $|\mathcal{C}_{P'}| = 60$ for $P' \in \mathcal{P}_f$

Fig. 11: Experiments with dynamic updates: (a) insertion scenario; (b) removal scenario; (c) replacement scenario.

class sequences with almost the same total number of filters in duplicated class instances as AO with GG and LD. For inputs generated with parameters ($k = 10$, $|\mathcal{C}_P| = 80$, $|\mathcal{C}| = 100$, $|\mathcal{P}| = 5$), CS with GG and LD requires only 40% of the filters in the duplicated instances of classes versus UB on average (see Fig. 10c).

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.

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 GG and LD calculated 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 GG and LD, 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. 11a): 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. 11b): 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. 11c): 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 Fig. 11, the usage of LD is extremely important after modification operations. In the case of class removals (Fig. 11b), one can apply LD after all operations, and the result will stay the same. In the insertion scenario (Fig. 11a), 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. 11a); the algorithm that never uses LD (DYN3) looses in all scenarios, e.g., by 17% in the removal scenario in Fig. 11b, $|\mathcal{C}_P| = 85$, and much more as we increase $|\mathcal{C}_P|$.

In the scenario with 8 replaced classes in each policy (see Fig. 11c) the number of filters in duplicated class instances in the output is at most 14% larger than in the case of CS with GG and LD. Finally, the algorithm DS constructing P_f with dynamic inserts from scratch ($|\mathcal{C}_P| = 0$ in Fig. 11a) builds a combined representation that has 19% more filters in duplicated class instances then CS with GG and LD. On the other hand, the time complexity of DS is significantly smaller than the time complexity of CS with GG and LD; hence, DS can be used as the offline algorithm constructing $\mathbb{S}$ in cases when time complexity of constructing the representation is a bottleneck. Note that DS does not provide any guarantees on the resulting $\mathbb{S}$ even for $\mathcal{P}$ that does have an ideal representation.

Our evaluation study confirms the usefulness of class-based abstractions in optimization of policy classifiers both based on sequences and partial orders of classes, but the former perform much better. In the offline case, CS with GG and LD significantly outperform all other evaluated algorithms. The proposed algorithm for dynamic updates implements the fundamental trade-off between the number of filters in duplicated class instances and the time spent on the construction of resulting representations.

VIII. Previous Work

Finding efficient representations 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. These solutions mainly span three techniques: decision trees, hashing, and coding-based compression [15]–[20]. In decision trees, finding a matching filter is based on tracing a path in a decision tree (e.g., [15]); however, there is an inherent tradeoff between space and time

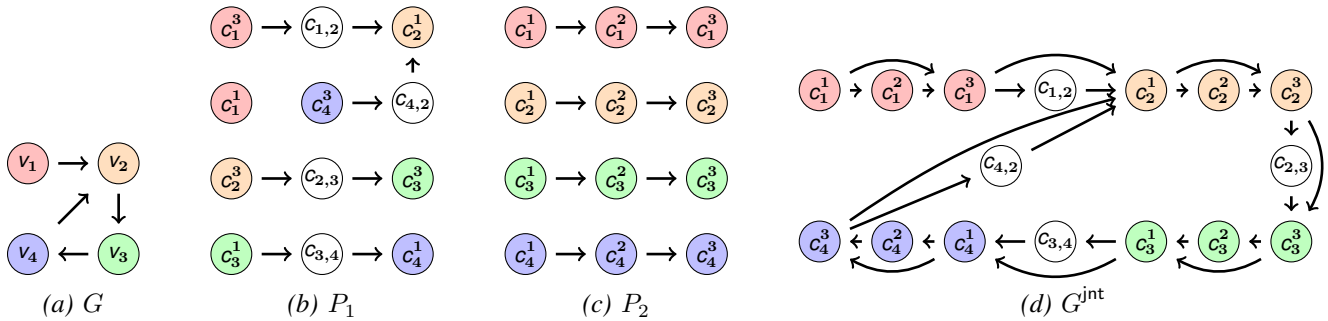


Fig. 12: Illustration for the proof of Theorem 8: (a) a graph G; (b) - (c) P_1, P_2 constructed from G; (d) G^{jnt} for $\mathcal{P} = \{P_1, P_2\}$.

complexity in these approaches. Hash-based solutions that match a packet to its possible matching filters have also been considered [17]. 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 [18]–[22]. Unfortunately, in most cases these methods apply only to filters with a limited number of fields or perform poorly as it increases. The works [23]–[28] 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 [29]. In [30], 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 [31], [32] and found to be an intractable optimization problem. The main contribution of [33] 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. The work [34] introduces the notion of classification with a controlled error allowing to trade the classification accuracy for the additional efficiency of classifier representations. Note that all these proposals are orthogonal to our combined policy representations and can be used alongside with it.

The FVS and especially the SCS problems are among classical NP-complete problems, with plenty of research devoted to them. In particular, the work [35] presents general ideas of "splitting" vertices that we extend here, [36] proves hardness results for SCS and similar problems, though [37] studies parameterized complexity and shows that SCS problem is already $W[1]$-hard. The work [38] presents approximation results for $\mathrm{SCS}(2,k)$ and $\mathrm{SCS}(2,3)$ that also employ a reduction to the FVS problem. The work [39] presents `Reduce-Expand` techinuqe that is effective only for SCS instances in which the same characters often appear in the same strings. [12] surveys practical SCS heuristics.

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 the structural properties of represented policies on a single network element.

Appendix

Proof of Theorem 8. The proof is by reduction from the WFVS problem [7]. 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 $v_i \in V$ we create three classes: an *input* class c_i^1, a *middle* class c_i^2, and an *output* class c_i^3. For each edge $(v_i, v_j) \in E$ we create a class $c_{i,j}$. 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 $v_i \in V$, the class c_i^2 intersects with both c_i^1 and c_i^3. For each $(v_i, v_j) \in E$, the class $c_{i,j}$ intersects with both c_i^3 and c_j^1. All other classes are pairwise disjoint. Each class in every policy is associated with a different action. The policy P_1 contains the classes c_i^1, c_i^3 for each $v_i \in V$ and classes $c_{j,k}$ for each $(v_j, v_k) \in E$. The partial order of P_1 is defined as follows: for each edge $(v_i, v_j) \in E : c_i^3 \prec_{P_1} c_{i,j} \prec_{P_1} c_j^1$. The policy P_2 consists of all classes c_i^1, c_i^2, c_i^3 for each $v_i \in V$. The partial order of P_2 is defined as follows: for each $v_i \in V : c_i^1 \prec_{P_2} c_i^2 \prec_{P_2} c_i^3$.s Figure 12 illustrates the reduction from FVS to PSP for a graph G consisting of four vertices.

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: $c_i^1 \in V_{G^{\mathsf{jnt}}} \equiv v_i \in V_G$. 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 G^{jnt} and G. □

References

[1] V. Demianiuk, S. Nikolenko, P. Chuprikov, and K. Kogan, "New alternatives to optimize policy classifiers," in *ICNP*, Sept 2018, pp. 121–131.

[2] "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.

[3] "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.

[4] "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.

[5] "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.

[6] 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.

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

[8] 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.

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

[10] 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.

[11] S. Kasif, Z. Weng, A. Derti, R. Beigel, and C. DeLisi1, "A computational framework for optimal masking in the synthesis of oligonucleotide microarrays," *Nucleic Acids Research*, vol. 30, no. 20, p. e106, 2002.

[12] K. Ning and H. Leong, "Towards a better solution to the shortest common supersequence problem: the deposition and reduction algorithm," *BMC Bioinformatics*, vol. 7, no. Suppl 4, p. S12, 2006.

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

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

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

[16] 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.

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

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

[19] 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.

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

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

[22] V. Demianiuk and K. Kogan, "How to deal with range-based packet classifiers," in *SOSR*, 2019, pp. 29–35.

[23] K. Kogan, S. Nikolenko, W. Culhane, P. Eugster, and E. Ruan, "Towards Efficient Implementation of Packet Classifiers in SDN/OpenFlow," in *HotSDN*, 2013.

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

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

[26] S. I. Nikolenko, K. Kogan, G. Rétvári, E. R. Bérczi-Kovács, and A. Shalimov, "How to represent ipv6 forwarding tables on ipv4 or MPLS dataplanes," in *INFOCOM Workshops*, 2016, pp. 521–526.

[27] P. Chuprikov, K. Kogan, and S. I. Nikolenko, "General ternary bit strings on commodity longest-prefix-match infrastructures," in *ICNP*. IEEE Computer Society, 2017, pp. 1–10.

[28] A. Kesselman, K. Kogan, S. Nemzer, and M. Segal, "Space and speed tradeoffs in TCAM hierarchical packet classification," *J. Comput. Syst. Sci.*, vol. 79, no. 1, pp. 111–121, 2013.

[29] C. Umans, "The minimum equivalent DNF problem and shortest implicants," *J. Comput. Syst. Sci.*, vol. 63, no. 4, pp. 597–611, 2001.

[30] K. Kogan, S. Nikolenko, P. Eugster, and E. Ruan, "Strategies for Mitigating TCAM Space Bottlenecks," in *HOTI*, 2014.

[31] Y. Kanizo, D. Hay, and I. Keslassy, "Palette: Distributing tables in software-defined networks," in *INFOCOM*, 2013, pp. 545–549.

[32] 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.

[33] 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.

[34] V. Demianiuk, K. Kogan, and S. I. Nikolenko, "Approximate classifiers with controlled accuracy," in *INFOCOM*, April 2019, pp. 2044–2052.

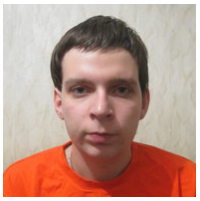
[35] V. G. Timkovsky, "Ten etudes on shortest common nonsubsequence and supersequence approximations," Unpublished manuscript, 1993.

[36] V. G. Timkovskii, "Complexity of common subsequence and supersequence problems and related problems," *Cybernetics*, vol. 25, no. 5, pp. 565–580, Sep 1989.

[37] M. Fellows, M. Hallett, and U. Stege, "Analogs & duals of the mast problem for sequences & trees," *Journal of Algorithms*, vol. 49, no. 1, pp. 192 – 216, 2003.

[38] Z. Gotthilf and M. Lewenstein, "Improved approximation results on the shortest common supersequence problem," in *SPIRE*, 2009, pp. 277–284.

[39] P. Barone, P. Bonizzoni, G. Della Vedova, and G. Mauri, "An approximation algorithm for the shortest common supersequence problem: An experimental analysis," in *SAC1 2001*, 01 2001, pp. 56–60.



Vitalii Demianiuk Vitalii Demianiuk is a postdoctoral fellow at the Ariel University. He completed his Ph.D. studies at the Steklov Institute of Mathematics at St.Petersburg in 2019. While pursuing his PhD he worked as a Research Assistant at IMDEA Networks Institute, Madrid. He obtained his M.Sc. from the ITMO University, St. Petersburg in 2016. His research interests include packet classification, software defined networks, network function virtualization, and combinatorial optimization.

Sergey Nikolenko Sergey Nikolenko is an Associate Professor at the Higher School of Economics at St. Petersburg, Russia, and a Laboratory Head at the Steklov Institute of Mathematics at St. Petersburg, Russia. He is doing research in machine learning (deep learning, Bayesian methods, natural language processing, and more), analysis of algorithms (algorithms for networking, competitive analysis, theoretical computer science), and mathematics. He obtained his Ph.D. degree from the Steklov Institute of Mathematics in 2009 and his M.Sc. from the St. Petersburg State University in 2005.

Pavel Chuprikov Pavel Chuprikov received his Ph.D. degree from the Higher School of Economics, Moscow after completing Ph.D. studies at the Steklov Institute of Mathematics at St.Petersburg. While pursuing his PhD he worked as a Research Assistant at IMDEA Networks Institute, Madrid. Currently, Pavel is a Postdoctoral researcher at the Università della Svizzera Italiana. His research interests include software defined networking, online algorithm design, and dependent types.

Kirill Kogan Kirill Kogan is a Senior Lecturer at Ariel University. He received his PhD from Ben-Gurion University (Israel) at 2012. He is a former Technical Leader at Cisco Systems, where he worked during 2000-2012. He was a Postdoctoral Fellow at University of Waterloo and Purdue University during 2012-2014. His current research interests are in design, analysis, and implementation of networked systems, broadly defined.