

Self-Adjusting Packet Classification

MACIEJ PACUT, Faculty of Computer Science, University of Vienna, Austria

JUAN VANERIO, Faculty of Computer Science, University of Vienna, Austria

VAMSI ADDANKI, Faculty of Computer Science, University of Vienna, Austria

ARASH POURDAMGHANI, Faculty of Computer Science, University of Vienna, Austria

GABOR RETVARI, Budapest University of Technology and Economics, Hungary

STEFAN SCHMID, Faculty of Computer Science, University of Vienna, Austria

This paper is motivated by the vision of more efficient packet classification mechanisms that self-optimize in a demand-aware manner. At the heart of our approach lies a self-adjusting linear list data structure, where unlike in the classic data structure, there are dependencies and some items must be in front of the others; for example, to correctly classify packets by rules arranged in a linked list, each rule must be in front of lower priority rules that overlap with it. After each access we can rearrange the list, similarly to Move-To-Front, but dependencies need to be respected.

We present a 4-competitive online rearrangement algorithm, whose cost is at most four times worse than the optimal offline algorithm; no deterministic algorithm can be better than 3-competitive. The algorithm is simple and attractive especially for memory-limited systems, as it does not require any additional memory (e.g., neither timestamps nor frequency statistics). This drop-in replacement for static lists of rules may enhance existing networked systems.

We evaluate our self-adjusting list packet classifier on realistic ruleset and traffic instances. We find that our classifier performs similarly to a static list for low-locality traffic, but significantly outperforms Efficuts (by factor 7x), CutSplit (3.6x) and the static list (14x) for high locality and small rulesets. Memory consumption is 10x lower on average compared to Efficuts and CutSplit.

ACM Reference Format:

Maciej Pacut, Juan Vanerio, Vamsi Addanki, Arash Pourdamghani, Gabor Retvari, and Stefan Schmid. 2021. Self-Adjusting Packet Classification. In *ACM SIGMETRICS '22, July 06–10, 2022, Mumbai, India*. ACM, New York, NY, USA, 22 pages. <https://doi.org/10.1145/123456789>

1 INTRODUCTION

Packet classification [13] is the most fundamental task in communication networks, performed by switches, routers and middleboxes (e.g., firewalls). Existing packet classifiers are typically traffic-oblivious: their internal data structures do not depend on the specific traffic patterns they serve. For example, decision trees are typically designed for good performance under uniform traffic patterns.

This paper is motivated by the hypothesis that the performance of packet classification can be improved by more adaptive approaches. Indeed, in practice, workloads are often far from uniform, and the majority of traffic can be served with few rules [24]. If a packet classifier could identify and optimize toward the “heavy hitter rules”, its reaction time and throughput could potentially be improved.

Realizing the vision of such self-adjusting packet classifiers however is challenging. In particular, as shifts in the traffic patterns (and hence also in the heavy hitter rules) may not be predictable,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM SIGMETRICS '22, June 06–10, 2022, Mumbai, India

© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-9999-9/20/06...\$15.00

<https://doi.org/10.1145/123456789>

online algorithms need to be designed which strike a good tradeoff between the benefits and the costs of rearrangements. Ideally, these online algorithms achieve a good *competitive ratio*: intuitively, without knowing the future demand, their performance is almost as good as a clairvoyant optimal offline algorithm. Besides the challenge of continuously identifying what to optimize for, these algorithms should also be efficient in terms of memory usage and runtime, and not require collecting much statistical data, e.g., related to the hit frequencies, or have high convergence times.

Our approach in this paper builds upon the concept of self-adjusting data structures. Intuitively self-adjusting data structures provide the desired adaptability to workloads, and allow exploiting “locality of reference”, i.e., the tendency to repeatedly access the same set of items over short periods of time. Such data structures have been studied intensively for several decades already, including self-adjusting linear lists, which were one of the first data structures of this kind [26, 27].

Self-adjusting packet classifiers however also introduce novel requirements that do not exist in data structures. To correctly classify a packet according to a certain rule, we must not only check if it is a *match*, but also if it is the *best match*, by excluding matches to higher priority rules first. This check is unnecessary if the higher priority rule is independent: if the matching domains of the rules do not overlap, examining the rules in any order uniquely determines the action. We refer to this challenge as *inter-rule dependencies*, and note that usually we have few of them [17]. Consequently, any self-adjusting packet classifier must examine the matching conditions in a specified order. This introduces constraints in the internal structure of the classifier, specific to its design.

At the core of our approach to design self-adjusting packet classifiers lies a self-adjusting linear list data structure. We store rules in the nodes of the list, and to classify a packet, we match it to the rules in that order. The rule that matched the packet is then moved closer to the head of the list to speed up future matches. However, the inter-rule dependencies fix the order of some rules in the list: each rule must be in front of lower priority rules that overlap with it. This limits the effectiveness of self-adjustments, and significantly changes the structure of the problem.

1.1 Contributions

We present a *self-adjusting packet classifier* that automatically optimizes for the traffic it serves. Our approach is simple and efficient, and does not require maintaining timestamps or frequency statistics, as it only relies on simple pointer swaps and comparisons. It operates without any a priori assumptions about traffic distribution, and with near-instant convergence to (even local) shifts in traffic. The classification time improves with *locality of reference in traffic*: a tendency to repeatedly match the packet to a small set of rules over short periods. Its behavior is inspired by the Move-To-Front algorithm [26] that moves the item to the front each time it is accessed.

Competitive analysis. Our approach provides provable guarantees over time, and we present a competitive analysis [8] accordingly. Accounting for packet classification needs, we generalize the classic *online list access problem* [26] to the variant with precedence constraints among the nodes of the list. The designed algorithm, Move-Recursively-Forward is 4-competitive with dependencies, reaching the competitiveness of Move-To-Front¹ known from the classic setting. No deterministic online algorithm can be better than 3-competitive: a result carried from the classic list access problem [23].

Handling any inter-rule dependencies. The inter-rule dependencies may form any partial order, or equivalently any directed acyclic graph (see Figure 1). Handling arbitrary forms of dependencies saves memory and classification time by eliminating the need for rule preprocessing. If we restrict

¹Move-To-Front is 2-competitive in the *free exchange model*, and 4-competitive in the *paid exchange model* [23]. The latter model fits our study better, see the model section (Section 2.1).

the dependencies to e.g., trees, we may face an exponential growth of the ruleset size due to rule splitting during preprocessing [13].

Evaluation. We compare the system against packet classifiers that build decisions trees based on hierarchical cuts, and study whether self-adjusting list packet classifier can compete with existing packet classifiers in terms of performance.

Applications. Our self-adjusting list packet classifier is a drop-in replacement for these static lists and comes with no memory penalty, no significant increase in complexity, and performance improvements under high traffic locality. It may further enhance existing static lists maintained in the leaves of some hierarchical cut-based packet classifiers.

1.2 Related Work

Packet classification. Various data structures for this packet classification were proposed in the literature: lists, tries, hash tables, bit vectors or decision trees [10, 13, 28], as well as hardware solutions (TCAM). Packet classifiers are often accompanied by caching systems that provide some adjustability to traffic. Due to its simplicity, a linear lookup structure is commonly applied in practice; e.g., in the default firewall suite of the Linux operating system kernel called iptables [20], the OpenFlow reference switch [22], and in many QoS classifiers.

We are not the first to study self-adjusting list-based packet classifiers. Existing heuristic solutions typically rely on frequency statistics, and hence may adapt slowly to shifts in traffic.

- The Dynamic Rule Reordering algorithm [14] performs a batch reorganization to a configuration that is (heuristically) close to optimum in hindsight, using frequency and recency information to assign weights to the rules. This greedy solution builds the list by iteratively inserting the next heaviest rule with all its dependencies (weight of which is not considered).
- A local search algorithm [31] updates the list on-the-fly, possibly after each request. Neighboring nodes exchange positions if (1) are independent and (2) their weights are inverted. The procedure stops when further neighbor exchanges are impossible. Local search algorithms for some optimization problems may get stuck at a local minimum, and it is challenging to find a decent starting point for the search.

We note that packet classification with self-adjusting lists is NP-hard even without inter-rule dependencies, by a reduction from *offline list access* [4].

Online list access. Online list access problem was studied for decades [19], and remain an active field of research [2]. An algorithm Move-To-Front [26] that is 4-competitive in the *paid exchange model* [23] (used in this paper). Algorithms such as FREQUENCY-COUNT (sorting items according to use frequencies) or TRANSPOSE (on access exchange with the preceding node) are *not* constant-competitive: their competitive ratio grows with the length of the list [26]. No deterministic online algorithm can be better than 3-competitive [23]. Interestingly, obtaining an optimal 3-competitive algorithm is challenging: even in the setting without precedence constraints, no deterministic algorithm was proven to be better than 4-competitive yet (but a survey [16] suggests that an algorithm Move-To-Front-every-other-access may be 3-competitive). Randomization helps: RANDOM RESET [23] is $\sqrt{7} \approx 2.64$ -competitive against an oblivious adversary. An offline optimal solution is NP-hard to compute [4].

Algorithms for list access perform well in practice [6]. These algorithms were empirically evaluated under inputs exhibiting various forms of locality: such as generating them according to Zipf's law or Markov distributions [6]. A notable work in this area introduced the notion of *runs* that provably improves the competitive ratio [3].

Some papers considered variants of list access with unequal costs of access and rearrangement, but their focus was on increased cost of rearrangements — in contrast to our model, where the packet matching (access) is more costly to rearrangements. The model P^d [23] is a variant of the paid exchange model where the cost of each transposition is $d \geq 1$. In this model, the best known algorithm is COUNTER [23]. Curiously, our practical motivation to packet classification is the first to justify the cost of reconfiguration that is *smaller* than the cost of access, and can be viewed as a generalization of the P^d model for $d < 1$.

Online problems with precedence constraints. Various online problems were studied in settings with dependencies and precedence constraints. In scheduling with precedence constraints [5], a job may be scheduled only after all its predecessors are completed. In caching with dependencies [7] (another online problem motivated by packet classification), an element can be brought into the cache only if all its dependencies are in the cache.

1.3 Organization

The rest of this article is organized as follows. Section 2 introduces the optimization model for *online list access with precedence constraints* and presents the self-adjusting algorithm Move-Recursively-Forward. Sections 3 and 4 analyze the competitiveness of Move-Recursively-Forward; we dedicate a section for the basic cost model and one for the generalized model. Section 5 discusses different implementation strategies leading to different time-memory tradeoffs. In Section 6, we discuss insertions and deletions: building blocks for rule update mechanisms in the context of competitive analysis. We evaluate both implementations against hierarchical cut tree-based algorithms (under various ruleset sizes, inter-rule dependencies and traffic locality) in Section 7 and conclude and discuss future research directions in Section 8.

2 SELF-ADJUSTING LIST WITH DEPENDENCIES

We realize the vision of self-adjusting packet classifiers by organizing match-action rules in a linked list. A packet is first matched against the rule at the head of the list, and then, depending on the action, passed on to be classified further by the rules down in the list, according to the list order (respecting the precedence constraints). With this perspective, matching the packet to its rule in a list can be seen as the *find* operation for a node in a list (in this context often called *access*). The self-adjustments rearrange the nodes after each access. We speed up repeated matches to the same rule by moving it closer to the head of the list. This rearrangement comes at a cost of swapping pointers. As described so far, the model is equivalent to *online list access* [26]. However, there are special considerations in packet classification that introduce additional challenges.

To account for inter-rule dependencies keep certain rules in a fixed relative order, limiting the effectiveness of rearrangements, and making the algorithmic problem significantly more challenging. Besides moving the accessed node forward in the list, we additionally move forward a carefully chosen set of the dependencies. This assures the progress can be made even if the accessed node is blocked (a dependency is directly in front of it).

2.1 Optimization Model: Online List Access with Precedence Constraints

Our task is to manage a self-adjusting linked list serving a sequence of requests, with minimal access and rearrangement costs and accounting for precedence constraints induced by a directed acyclic graph G . If there are no constraints, the problem is equivalent to the classic online list access problem [26].

The list and the requests. Consider a set of n nodes arranged in a linked list. Over time, we receive a sequence σ of access requests to nodes of the list. Upon receiving an access request to a node in

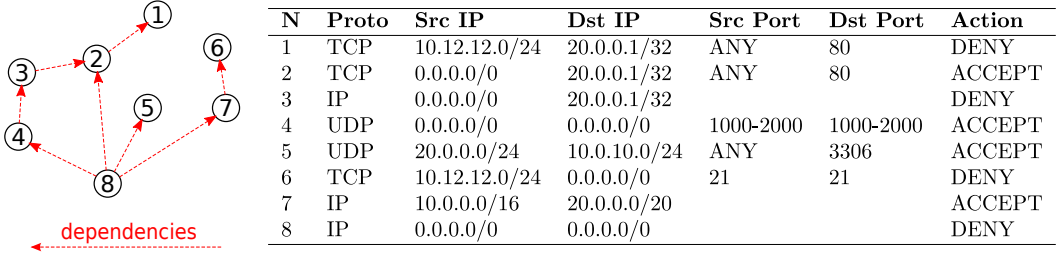


Fig. 1. An example of a table of packet classification rules (right) and the corresponding dependency DAG G among the nodes of the list (left).

the list, an algorithm searches linearly through the list, starting from the head of the list, traversing nodes until encountering the accessed node. Accessing the node at position i in the list costs i (the first node is at position 1). Upon receiving a deletion request, we must access the node first. Upon receiving an insertion request, we may insert a node in any feasible position (a list rearrangement may be required to deal with precedence constraints).

The precedence constraints (dependencies). We are given a directed acyclic graph G of dependency relations between nodes of our list. This dependency graph induces a partial order among the nodes (equivalent to the reachability relation in G) and the nodes must obey the partial order in the list in any configuration. We say that a node v is a *dependency* of a node u if there exists an edge (u, v) in G (we orient edges from the node u to the node v in front of it). Then, in every configuration of the list, v must be in front of u . We assume that the given initial configuration of the nodes obeys the precedence constraints induced by G .

Node rearrangement. After serving a request, an algorithm may choose to rearrange the nodes of the list. Precisely, the algorithm may perform any number of *feasible* transpositions of neighboring nodes, i.e., transpositions that respect the precedence constraints induced by G . We study the *paid exchange model* where all transpositions incur the cost 1; this is different from the *free exchange model* sometimes considered in the literature where moving the requested node closer to the front is free.

The model poses new algorithmic challenges. Using algorithms for online list access (such as Move-To-Front [26] or Timestamp [1]) could lead to infeasible solutions (violating the precedence constraints). The question arises, if provably good algorithms even exist for our setting, or the problem becomes much harder for online algorithms when faced dependencies?

Model generalizations. In Section 4, we generalize the model slightly by introducing a parameter $\alpha \geq 1$ related to the cost of access (a multiplier). In Section 6, we discuss insertion and deletion operations.

2.2 Algorithm Recursively-Move-Forward

How to navigate a DAG of dependencies? For a node y , we distinguish its *direct dependency*: the dependency node z that would block y 's movement forward in the list. Formally, a node y is a *direct dependency* of a node z if y is the dependency of z that is located at the furthest position in the list.

Our algorithm Move-Recursively-Forward moves forward *the sequence of direct dependencies*: the accessed node is swapped forward, until encountering one of its dependencies, at which point the encountered node runs forward recursively. This strikes a balance between the access and

rearrangement costs: the algorithm exchanges no more pairs than the position of the accessed node (Lemma 1). In Figure 2, we depict an example run of MRF after serving a request.

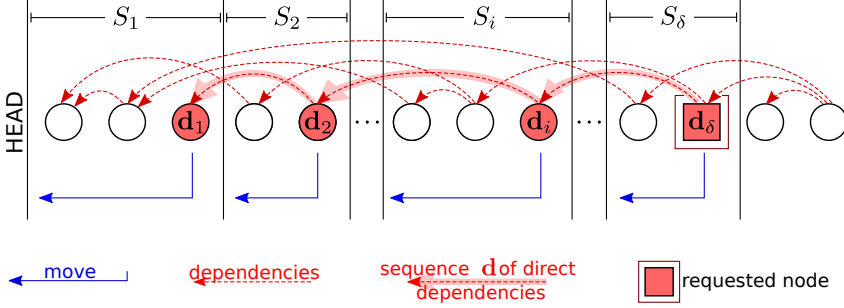


Fig. 2. An example of handling a request by the algorithm MRF. The red nodes d_i , formally introduced in Section 3, are the nodes that the algorithm moves forward (to the position denoted by a blue arrow). Among multiple dependencies (red dashed arrows), the algorithm MRF moves forward only the nodes that constitute the path of the direct dependencies (bold red arrow). Sets S_i are used in the analysis, cf. Section 3.

Algorithm MOVE-RECURSIVELY-FORWARD. The algorithm uses a recursive procedure MRF. The procedure $\text{MRF}(y)$ moves the node y forward (by transposing it with the preceding nodes) until it encounters any of its dependency nodes, say z , and recursively calls $\text{MRF}(z)$. Upon receiving an access request to a node σ_t , MRF locates σ_t in the list and invokes the procedure $\text{MRF}(\sigma_t)$.

We present the pseudocode of MOVE-RECURSIVELY-FORWARD in Algorithm 1. By $\text{pos}(z)$ we denote the position of node z in the list maintained by the algorithm, counting from the head of the list (recall that the position of the first node is 1).

Algorithm 1: The algorithm MOVE-RECURSIVELY-FORWARD.

Input: An access request to node σ_t

- 1 Access σ_t
- 2 Run the procedure $\text{MRF}(\sigma_t)$

Procedure $\text{MRF}(y)$:

- 3 **if** y has no dependencies **then**
 - 4 Move y to the front of the list
 - 5 **else**
 - 6 Let z be the direct dependency of y
 - 7 Move node y to $\text{pos}(z) + 1$
 - 8 Run the procedure $\text{MRF}(z)$
-

The algorithm can be viewed as a generalization of Move-To-Front [26] to the setting with dependencies. If the dependency graph is empty, the algorithm is equivalent to Move-To-Front.

In the next section, we show that Move-Recursively-Forward is 4-competitive, meaning it performs close to an offline optimal algorithm that dynamically updates its list while knowing the future requests. The algorithm balances the overhead of adjustments with its benefits: for each request, the access and the succeeding rearrangements cost roughly the same.

Lemma 1. *Consider a single request to a node y at position $\text{pos}(y)$ handled by the algorithm MRF. The rearrangements after serving the request to y costs at most $\text{pos}(y)$.*

PROOF. Intuitively, each of the nodes that was moved forward goes through a disjoint part of the list, in total at most $\text{pos}(y)$. For a graphical argument, see Figure 2.

Recall that y is moved to the position right before its furthest dependency, and then recursively the dependency moves forward until encountering the dependency of its own. The movements end when a moving node reaches the front of the list. Each node is moved right to a position *one place behind* its dependency.

Let \mathbf{d} be the sequence of the nodes that the algorithm moves forward (calls to MRF), ordered by increasing distance to the head. Let δ be the length of \mathbf{d} . Then, the node \mathbf{d}_i moves to the position $\text{pos}(\mathbf{d}_{i-1}) + 1$ (for formality of the argument, we assume an artificial dependency at the head of the list, $\text{pos}(\mathbf{d}_0) = 0$). Thus, in total the number of transpositions is

$$\sum_i^{\delta} (\text{pos}(\mathbf{d}_i) - (\text{pos}(\mathbf{d}_{i-1}) + 1)) = \text{pos}(\mathbf{d}_{\delta}) - \text{pos}(\mathbf{d}_0) - \delta \leq \text{pos}(\mathbf{d}_{\delta}).$$

As $y = \mathbf{d}_{\delta}$, we conclude that the lemma holds. \square

3 COMPETITIVE ANALYSIS OF MOVE-RECURSIVELY-FORWARD

The competitive ratio compares the online algorithm ALG with an optimal offline algorithm OPT: the goal is to minimize ALG/OPT . We overview the framework of competitive analysis in Appendix A.

In this section, we prove 4-competitiveness of Move-Recursively-Forward for the case where $\alpha = 1$ (access cost equal to reconfiguration cost). The competitive ratio is strict, and we note that the algorithm satisfies the definition of a *memoryless online algorithm* [9]. Later in Section 4 we generalize the analysis for arbitrary α and show that the algorithm is asymptotically optimal in this setting too. In both cases, we use a potential function argument, and we overview it below.

Analysis overview. The proof schema generalizes the argument of Sleator and Tarjan for Move-To-Front [26]. We fix an optimal offline algorithm OPT and its run on a given input σ , and we relate OPT's run with the online algorithm's run. A potential function measures the distance between the algorithms, measured in *inversions*: pairs of nodes in reverse order in Move-Recursively-Forward's and OPT's lists. Generalizing the proof required a careful analysis of inversions caused by movements of many nodes after each access.

Events overview. To analyze the competitiveness on σ , we sum an amortized cost of a sequence of events of type:

- (A) An *access event* $R^i(\sigma_t)$ for $i \in \{0, 1\}$. The algorithm serves the access to the node σ_t and runs the Move-Recursively-Forward procedure. We assume a fixed configuration of OPT throughout this event.
- (B) A *paid exchange event* of OPT, $P(\sigma_t)$, a single paid transposition performed by OPT, where it either creates or destroys a single inversion with respect to the node σ_t . We assume a fixed configuration of MRF throughout this event.

3.1 Preliminaries and Notation

Before analyzing the competitive ratio of the algorithm, we introduce the notation and the sets and sequences of nodes relevant to our analysis.

Inversions. An *inversion* is an ordered pair of nodes (u, v) such that u is located before v in MRF's list and u is located after v in OPT's list. The inversion is the central concept in the analysis of the presented algorithms in this paper.

The rearranged nodes \mathbf{d}_j . Consider an access to a node σ_t and the node rearrangements at t . Let \mathbf{d} be the sequence of the nodes that the algorithm moves forward (calls to MRF), ordered by increasing distance to the head. Let δ be the length of \mathbf{d} . We emphasize that \mathbf{d} contains the accessed node at the last position, $\sigma_t = \mathbf{d}_\delta$.

Values k and ℓ . To compare the costs of MRF and OPT, we define values k and ℓ related to the number of nodes in front of the accessed node σ_t in MRF's and OPT's list. Precisely, let k be the number of nodes before σ_t in both MRF's and OPT's lists, and let ℓ be the number of nodes before σ_t in MRF's list, but after σ_t in OPT's list.

Sets K_j and L_j . With the values k and ℓ it is possible to analyze the classic algorithm Move-To-Front, but they are not sufficient to express the complexity of Move-Recursively-Forward. Hence, we generalize the notion of k and ℓ to sets of elements related to positions of individual nodes \mathbf{d}_j in MRF's and OPT's lists. Precisely, let K_j be the set of elements before \mathbf{d}_j in both MRF's and OPT's lists for $j \in [1, \delta]$, and let L_j be the set of elements before \mathbf{d}_j in MRF's list but after \mathbf{d}_j in OPT's list. We note that these sets are generalizations of k and ℓ : for the accessed node \mathbf{d}_δ we have $k = |K_\delta|$ and $\ell = |L_\delta|$.

Sets S_j . The sets of nodes between the nodes \mathbf{d} in MRF's list are crucial to the analysis. Intuitively, the node \mathbf{d}_i moves in front of all the nodes from the set S_i . Let S_1 be the elements between the head of MRF's list and \mathbf{d}_1 (included). For $j \in [2, \delta]$, let S_j be the set of elements between \mathbf{d}_j and \mathbf{d}_{j-1} (with \mathbf{d}_{j-1} excluded) in MRF's list.

Figure 3 illustrates an example of possible composition of sets K_j , L_j and S_j for different values of j on a given access request.

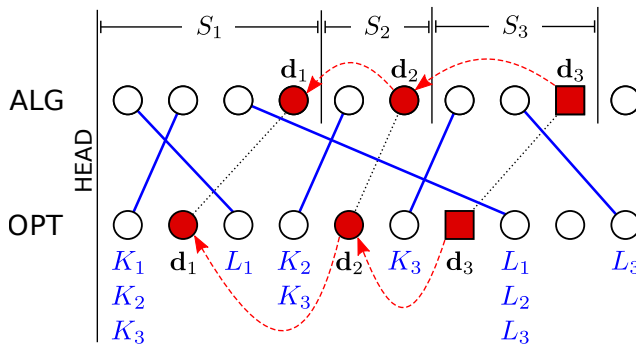


Fig. 3. The example illustrates central definitions of sets of nodes used in our analysis. We depict the positions of nodes in both MRF's and OPT's list (joined by solid blue lines). The dotted black lines between the nodes \mathbf{d}_i help in determining the assignment of nodes to sets: in K_i we have the nodes in front of the dotted black line between \mathbf{d}_i , and in L_i we roughly have the nodes that cross the dotted black lines between \mathbf{d}_i 's.

3.2 How Do Rearrangements Change Inversions?

Consider an access event, and the sequence of rearrangements of Move-Recursively-Forward. We study the influence of the rearrangements on the change in the potential function. To this end, we bound separately the number of introduced and destroyed inversions. The Figure 3 assists in illustrating the graphical arguments used in this section.

Theorem 2. *Consider access to the node σ_t , and fix a configuration of OPT at time t . Then, the change in the number of inversions due to MRF's node rearrangement after the access is at most $k - \ell$.*

To prove this claim, we consider the influence of the Move-Recursively-Forward operation on values k and ℓ (defined for the currently accessed node) by inspecting the sets K_j and L_j (defined for the nodes \mathbf{d}_j). We separately bound the number of inversions created (Lemma 4) and destroyed (Lemma 5). Before showing these claims, we inspect the basic relations between the sets K_j , L_j and S_j (Lemma 3).

Lemma 3. *The following relations hold:*

- (1) $\bigcup_{j=1}^{\delta} K_j = K_{\delta}$,
- (2) $\bigcup_{j=1}^{\delta} (S_j \cap L_j) = \bigcup_{j=1}^{\delta} L_j$.

PROOF. First, we prove the equality 1. We show inclusions both ways. Note that the order between nodes from \mathbf{d} is the same in both MRF's and OPT's lists. Hence, a node $y \in K_i$ is in front of \mathbf{d}_i and in front of all \mathbf{d}_j for $i \leq j$ in both MRF's and OPT's list. Consequently, each node from K_i belongs to all K_j for $i \leq j$, and we have $\bigcup_{j=1}^{\delta} K_j \subseteq K_{\delta}$. Conversely, $K_{\delta} \subseteq \bigcup_{j=1}^{\delta} K_j$ by basic properties of sets, and we conclude that the sets are equal, and the equality holds.

Next, we prove the equality 2. We show inclusion both ways. Consider any element $y \in L_j$. The sets $\{S_i\}$ partition the nodes placed closer to the front of the list than σ_t (the requested node), thus y belongs to some S_i for $i \leq j$. Fix such i ; we claim that additionally $y \in L_i$:

- y belongs to S_i , and hence it is in front of \mathbf{d}_i in MRF's list,
- y is after \mathbf{d}_j in OPT's list (it belongs to L_j), and hence it is after \mathbf{d}_i in OPT's list (the order of \mathbf{d} is fixed due to dependencies).

Hence, any $y \in L_j$ belongs to $S_i \cap L_i$ for some i , and we conclude that the inclusion $\bigcup_{j=1}^{\delta} L_j \subseteq \bigcup_{j=1}^{\delta} (S_j \cap L_j)$ holds. Conversely, by properties of sets $\bigcup_{j=1}^{\delta} (S_j \cap L_j) \subseteq \bigcup_{j=1}^{\delta} L_j$, and we conclude that the sets are equal and the equality holds. \square

Lemma 4. *Consider an access to a node σ_t , and fix a configuration of OPT at time t . Due to rearrangements after the access, MRF creates at most k inversions.*

PROOF. Let I_j^+ be the number of inversions added by moving a single node \mathbf{d}_j by MRF, for $j \in [1, \delta]$. To bound I_j^+ , we inspect the set S_j of the nodes that \mathbf{d}_j overtakes, and we reason based on their positions in OPT's list. Moving \mathbf{d}_j forward creates inversions with nodes in (possibly a subset of) $S_j \cap K_j$. No other node changes its relation to the set S_j , hence the inversions for nodes in S_j are influenced only by the movement of \mathbf{d}_j . This gives us the bound $I_j^+ \leq |S_j \cap K_j|$.

We sum up the individual bounds on I_j^+ for all j to bound the total number of inversions created

$$\sum_{j=1}^{\delta} I_j^+ \leq \sum_{j=1}^{\delta} |S_j \cap K_j| = \left| \bigcup_{j=1}^{\delta} (S_j \cap K_j) \right| \leq \left| \bigcup_{j=1}^{\delta} K_j \right|,$$

where the second step holds as the sets S_j are disjoint, and the last step follows by basic properties of sets. By Lemma 3, equation 1, we have $|\bigcup_{j=1}^{\delta} K_j| = |K_{\delta}|$, thus by combining the above inequalities, we have $\sum_{j=1}^{\delta} I_j^+ \leq |K_{\delta}| = k$, and we conclude that the claim holds. \square

Lemma 5. *Consider an access to the node σ_t , and fix a configuration of OPT at time t . Due to rearrangements after the access, MRF destroys at least ℓ inversions.*

PROOF. Let I_j^- be the number of inversions destroyed by moving a single node \mathbf{d}_j by MRF, for $j \in [1, \delta]$. To bound I_j^- , we inspect the set S_j of the nodes that \mathbf{d}_j overtakes, and we reason based on their positions in OPT's list. Moving \mathbf{d}_j forward destroys all inversions with nodes in $S_j \cap L_j$. No other node changes its relation to the set S_j , hence the inversions for nodes in S_j are influenced only by the movement of \mathbf{d}_j . This gives us the bound $I_j^- \geq |S_j \cap L_j|$.

We sum up the individual bounds on I_j^- for all j to bound the total number of inversions destroyed.

$$\sum_{j=1}^{\delta} I_j^- \geq \sum_{j=1}^{\delta} |S_j \cap L_j| = \left| \bigcup_{j=1}^{\delta} (S_j \cap L_j) \right|,$$

where the second step holds as the sets S_j are disjoint. By Lemma 3, equation 2 we have $|\bigcup_{j=1}^{\delta} (S_j \cap L_j)| = |\bigcup_{j=1}^{\delta} L_j|$. Finally, by basic properties of sets, $|\bigcup_{j=1}^{\delta} L_j| \geq |L_{\delta}| = \ell$, and by combining all the above bounds we have $\sum_{j=1}^{\delta} I_j^- \geq \ell$, and we conclude that the lemma holds. \square

Combining Lemmas 4 and 5 gives us the joint bound on the change in the number of inversions, and proves the Theorem 2. We note that this bound is consistent with the bound on the changes in inversions for the algorithm Move-To-Front [26], where the inversions were considered with respect to the accessed node only.

3.3 Bounding the Competitive Ratio

Finally, we show the main result of this section: the competitive ratio of the algorithm MRF is 4 for $\alpha = 1$. First, we apply the bounds on the number of inversions (Theorem 2), and then we bound the ratio using a potential function argument (Theorem 6).

Potential function. We define the potential function Φ in terms of number of inversions in MRF's list with respect to OPT's list. Precisely, the potential function is defined as *twice the number of inversions*. The factor 2 accounts for the cost of an access and an exchange for each inversion.

Finally, we prove the main result of this section, that the algorithm MRF is 4-competitive. The proof uses a potential function argument similar to Move-To-Front [26], and it internally uses the Theorem 2 to reason about changes in inversion due to Move-Recursively-Forward runs.

Theorem 6. *The algorithm MRF is strictly 4-competitive.*

PROOF. Fix a sequence of access requests σ . We compare the costs of MRF and an optimal offline algorithm OPT on σ using a potential function Φ . Let $C_{\text{MRF}}(t)$ and $C_{\text{OPT}}(t)$ denote the cost incurred at time t by MRF and OPT respectively.

First, we bound the cost of MRF incurred while serving an access to a node σ_t at time t (an *access event*). This cost consists of the access cost and the rearrangement cost. To access the node σ_t , the algorithm incurs the cost $\text{pos}(\sigma_t)$, and by Lemma 1 the rearrangement cost is bounded by $\text{pos}(\sigma_t)$, hence $C_{\text{MRF}}(t) \leq 2 \cdot \text{pos}(\sigma_t)$.

Next, we bound the amortized cost for every access request served by MRF. The amortized cost is $C_{\text{MRF}}(t) + \Delta\Phi(t)$ for each time t . By Theorem 2, we bound the change in the number of inversions due to MRF's rearrangement after serving the request at time t by $\Delta I \leq k - \ell$. Thus, the change in the potential is $\Delta\Phi(t) \leq 2(k - \ell)$. As $\text{pos}(\sigma_t) = k + \ell + 1$, combining these bounds gives us

$$\begin{aligned} C_{\text{MRF}}(t) + \Delta\Phi(t) &\leq 2 \cdot \text{pos}(\sigma_t) + 2(k - \ell) \\ &\leq 2(k + \ell + 1) + 2(k - \ell) \leq 4 \cdot C_{\text{OPT}}(t), \end{aligned}$$

where the last inequality follows by $C_{\text{OPT}} \geq k + 1$.

Note that the bound on amortized cost accounts for possible *paid exchange events*, the rearrangement of OPT at time t . Each transposition of OPT increases the number of inversions by 1, which increases the LHS by 2; and for each transposition OPT pays 1, which increases the RHS by 4.

Finally, we sum up the amortized bounds for all requests of the sequence σ of length m , obtaining

$$C_{\text{MRF}}(\sigma) + \Phi(m) - \Phi(0) \leq 4 \cdot C_{\text{OPT}}(\sigma).$$

We assume that MRF and OPT started with the same list, thus the initial potential $\Phi(0) = 0$, and the potential is always non-negative, thus in particular $\Phi(m) \geq 0$, and we conclude that $C_{\text{MRF}}(\sigma) \leq 4 \cdot C_{\text{OPT}}(\sigma)$. \square

4 COMPETITIVE ANALYSIS IN THE GENERAL COST MODEL

Some algorithms may incur different cost at access than at rearrangement, for example due to additional computations needed before swapping pointers (as we will elaborate for the second algorithm from Section 5). Such algorithms operate under a different cost model, and our analysis must be strengthened to account for various ratios of the incurred cost.

Generalized model. We introduce a parameter $\alpha \geq 1$ to the model defined in Section 2.1. Then, accessing the node at position i in the list costs $\alpha \cdot i$ (instead of i). The first node is at position $i = 1$.

We repeat the competitive analysis of Move-Recursively-Forward in the generalized model, to avoid coming to possibly wrong conclusions of a different cost model. The results from Section 3 hold for $\alpha = 1$. The central question of this section is:

Should we redesign our algorithm, and if so, does it need to be aware of α ?

This generalization is an opposite of the P^d model [23], where the cost of rearrangement is higher than access: $d \geq 1$. Curiously, to the best of our knowledge, the cost model with $d < 1$ was not considered before. We use the parameter $\alpha = 1/d$ for simplicity. We note that for the converse scenario — the standard P^d model with $d > 1$, the algorithms must be modified in a way that depends on the cost of reconfiguration (e.g. COUNTER algorithm [23]) to remain competitive.

In Theorem 8, we answer the question of this section from the standpoint of competitive analysis:

MRF is asymptotically optimal for any value of α , as long as $\alpha < n$,

where n is the number of nodes in the list. The value of α is the cost of multi-field packet matching, and is expected to be a small constant, far smaller than a typical list length.

4.1 Lower Bound

The competitive analysis framework finds online computational impossibility results for $\alpha > 1$: lower bounds linear in α for any online algorithm! The Theorem 8 reveals much more pessimistic scenario than a lower bound of 3 for $\alpha = 1$ [23]. We prove the result for the classic *online list access* without dependencies, since the result carries to our model.

Theorem 7. *If an algorithm ALG is a deterministic c -competitive algorithm for online list access in the general cost model with the visiting cost α and n nodes, then $c \geq \frac{n \cdot \alpha}{n + \alpha}$.*

PROOF. Fix any deterministic algorithm ALG, and consider an input sequence σ , where $\sigma(t)$ is the request to the last element of ALG's list at time t . The cost of traversing a node is α , thus the cost incurred by ALG for serving each request is $n \cdot \alpha$.

Consider the following offline strategy OPT: before serving a request to a node $\sigma(t)$, move it to the front of the list (at time $t - 1$). The cost of serving each request is then upper-bounded by $n + \alpha$. The competitive ratio is then at least $\frac{\text{ALG}}{\text{OPT}} \geq \frac{n \cdot \alpha}{n + \alpha}$. \square

We remark that *randomization* does not help more than a constant factor: the lower bound against randomized online algorithms also follows the linear trend in α . An oblivious adversary [8] cannot access the last node in the randomized algorithm's list, but instead it may access nodes with uniform probability. The expected cost of access for the algorithm is roughly $\alpha \cdot n/2$.

4.2 Upper Bound

Now we turn to the analysis of Move-Recursively-Forward in the generalized model.

Theorem 8. *The algorithm MRF is $\max\{4, 1 + \alpha\}$ -competitive for list update with dependencies in the general cost model.*

PROOF. The proof is a straightforward modification of 4-competitiveness proof from the previous section. To obtain the result, we use a potential function $\Phi = (1 + \alpha) \cdot I$, where I is the number of inversions. In case of a *request event*, the ratio is 4, and in case of a *paid exchange event* (by OPT), the ratio is $1 + \alpha$. \square

We note that the lower bound from Theorem 7 for $3 \leq \alpha \leq n$ is at most 3 times smaller than the upper bound from Theorem 8. Thus, the algorithm Move-Recursively-Forward is asymptotically optimal in this setting.

5 EFFICIENT IMPLEMENTATION

So far we abstracted from the implementation details of the algorithm Move-Recursively-Forward. The algorithm Move-Recursively-Forward can be implemented without any mutable memory besides the linked list: the dependency relations may be computed on the fly. The algorithm can be implemented in an array, reaching the lower bounds for (uncompressed) memory consumption. The recursive procedure is *tail-recursive*, eliminating the need for a call stack.

In this section we consider two approaches for representing the dependencies graph: *a simple memoryless algorithm* that computes dependencies on-the-fly, and *a more efficient algorithm* that precomputes and stores in-memory the dependency relations between nodes. This enables a time-memory tradeoff for the self-adjusting packet classifier.

Storing the dependencies in-memory speeds up the execution time of direct dependency operation. Instead of determining rule overlap for each of their fields, we may simply look it up in memory and compare pointers or checking if a bit is 1 (depending on implementation). However, this increases memory consumption: a time-memory tradeoff known from other systems (e.g. node replication factor in Efficuts [30]).

Storing the dependencies enables even further optimization options: to minimize the dependency graph while preserving the correctness of the algorithm. Notice that we may remove the dependencies that are indirect: in a chain of overlapping rules, we may store only the dependencies between neighbors rather than a full all-pairs graph. Consider two dependencies A (indirect) and B (direct) of a given node C. Consider A is a dependency of B too, hence their order is fixed. Thus,

the direct dependency of C can never be A. The indirect dependencies are never relevant in the direct dependency operation, and we may get rid of them, improving the running time and memory consumption.

5.1 First Algorithm: Memoryless

How to represent the dependency graph in memory? The simplest solution, the *memoryless* algorithm, avoids storing it altogether: to compute the dependencies on-the-fly by checking rules overlap and priority. In this implementation, matching a packet to a rule costs roughly as much as checking the dependency relation, fitting our model with $\alpha = 1$. This is our baseline implementation, and we evaluate this solution in Section 7. This implementation is simple: below 150 lines of C++ code, whereas some packet classifiers such as EffiCut have thousands of lines of code [30].

5.2 Second Algorithm: Storing the Dependency Graph in Memory

We propose an alternative implementation, designed to trade memory for execution time by storing the dependency graph in memory. By accessing the pre-computed dependency relations, we may expect faster execution of the rearrangement logic.

The gains of storing the dependency graph vary with the representation. An adjacency matrix guarantees constant-time lookup, but at the cost of a large memory footprint. An adjacency list guarantees lookup proportional to the number of neighbors, and saves a substantial amount of memory, given that common dependency graphs are sparse. A sparse matrix representation shows decent asymptotic runtime, employing binary search. In practice, solutions based on hashing may perform very well. We do not pursue the representation question further, but we assume its efficient memory footprint: proportional to the number of stored dependencies.

Storing all edges of the dependency graph is not necessary, as it can even be detrimental to the algorithm's performance. Consider a chain of m rules, where the lower priority rules overlap with higher priority rules (inducing the dependency relation). In the dependency graph, the last rule has $m - 1$ dependencies, and the total number of edges is in order of m^2 . The graph contains not only edges between neighbors, but also its *transitive closure*.

Instead of storing all edges, a *minimum equivalent DAG* [21] is sufficient. Removing indirect dependencies speeds up the direct dependency operation significantly. Moreover, reducing the graph affects the memory consumption, assuming memory-efficient representations.

At this point the cost model changes, and we generalize the competitive analysis to the updated setting, finding that the algorithm is still near-optimal. The constant α , defined in Section 2.1 is larger than 1, because the rearrangement cost (swapping) becomes cheaper than before: it was roughly the cost of matching the packet, as it required determining dependencies. Now, it consists only of a pointer comparison and swap, while matching a packet to a rule needs to examine multiple fields, yielding $\alpha > 1$.

5.3 Competitive Analysis Remarks

A question arises around various rearrangement strategies available: if the algorithm balances the rearrangement cost, should we rearrange the list more aggressively? For example by moving more (or all) dependencies forward, we may improve the position of the accessed node for future accesses even further. This would be ill-advised, as such an algorithm is less competitive: moving all to the front is $O(d)$ -competitive, where d is the maximum depth of the dependency graph.

What would be the effect of rearranging less often? For example, to rearrange the list after every 10th access, or with probability $1/10$. These more conservative algorithms were analyzed for the classic list access (without dependencies), and were designed for a different setting, where the cost of an exchange is *larger* than the access cost. In this model, the best known algorithm

is COUNTER [23]. However, such modifications are unnecessary: in Section 4, we claim that the algorithm Move-Recursively-Forward remains near-optimal for small α .

6 HANDLING INSERTIONS AND DELETIONS

In data structures, such as linked lists, the sets of nodes can change over time. Many data structures (including the ones for packet classification) hence support not only access operations, but also insertions and deletions.

To our best knowledge, we are the first to explore the problem of insertions and deletions in list data structures with dependencies. In this section, we discuss the feasibility of insertions: the algorithm may require reconfiguring other rules before inserting a new rule. We propose assumptions that allow for constant-competitiveness with insertions.

6.1 Model: Online List Update with Precedence Constraints

Consider the online list *update* problem with dependencies with three request types: *accesses* to existing nodes in the list, *insertions* of new nodes and *deletions* of existing nodes in the list. Upon receiving an insertion request, the node reveals its dependencies with the nodes that are already in the list. The revealed dependencies must be obeyed until a node is deleted. Note that a node may have dependencies with nodes that will be inserted later, but this information is unknown until then.

6.2 Packet Classification Challenges: Non-Transitivity

A desired property of possible dependency graphs is *transitivity*: the edges of the graph induce a relation that is a *transitive closure*. We report that the packet classification rule dependencies *may not* have the transitivity property. An inserted rule may reveal dependencies between the existing rules that become violated. For an example, see Table 1.

N	Proto	Src IP	Dst IP	Src Port	Dst Port	Action
1	TCP	10.1.1.1	20.1.1.1	ANY	80	ACCEPT
2	TCP	10.1.1.2	20.1.1.1	ANY	80	ACCEPT
3	TCP	10.1.1.3	20.1.1.1	ANY	80	ACCEPT
x	TCP	10.1.1.0/24	20.1.1.1	ANY	ANY	DENY
4	TCP	0.0.0.0/0	0.0.0.0/0	ANY	445	ACCEPT
5	TCP	0.0.0.0/0	0.0.0.0/0	ANY	17	ACCEPT
6	TCP	0.0.0.0/0	0.0.0.0/0	ANY	18	ACCEPT

Table 1. Consider a set of n nodes with no dependencies between them. Then, consider an insertion of a rule x , which reveals the dependencies with all existing rules. The insertion of rule x enforces the order between all existing rules. The dependencies imposed by the rules are not transitive. With the transitivity assumption, the rules (4-6) would have dependencies to rules (1-3), and the rule x would be inserted between them without rearrangements. The situation is illustrated in Figure 4.

Lack of transitivity may lead to costly rearrangements required to insert a single node. Figure 4 depicts an example.

How to deal with insertions without preprocessing rules? Faced with the need to rearrange before insertion, we recommend finding the minimum rearrangement that satisfies new constraints. Examples such as Figure 4 cannot happen too often: each time they add dependencies, and the adversary would need to delete nodes to deceive the algorithm again. An amortized analysis may lead to constant-competitiveness, and we leave it to future work.

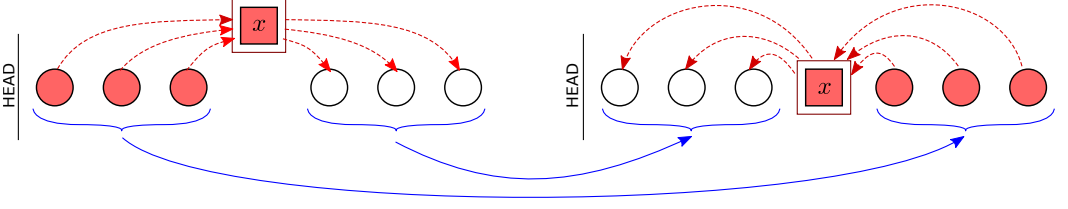


Fig. 4. Consider a set of n nodes in a list, with no dependencies between them. Then, consider insertion of a node x , which reveals the depicted dependencies (arrow points to the node that must precede in the list). Here, x has $\frac{n}{2}$ dependencies from the first half list (the red nodes) to x , and $\frac{n}{2}$ dependencies from x to the nodes of the second half of the list (the white nodes). To insert x without violating the dependencies, the nodes on the first half of the list must move ahead of the nodes on the second half, thus the cost of rearrangement is $(\frac{n}{2})^2$.

6.3 Constant-competitiveness With Transitivity Assumption

We build on assumptions present in the online list update problem [26], where insertions cost n and nodes are accessed before deletion [26]. Additionally, we require *transitivity*.

Under these assumptions, any algorithm inserts the node σ_t at any position of its choice that respects the dependency DAG G , *without incurring further cost*. A position that satisfies all constraints already exists in every configuration due to transitivity assumption. Consider the universe of all nodes, including the ones that may be inserted in the future. Assume the transitivity of their precedence constraints, meaning that if a node v_1 must be in front of v_2 , and v_2 must be in front of v_3 , then v_1 must be in front of v_3 , *even if v_2 is currently not present in the list*.

Move-Recursively-Forward with insertions and deletions. The algorithm's logic for access does not change. Upon receiving an insertion request, insert the new node into an arbitrary feasible position (respecting the dependencies). Upon receiving a deletion request, the algorithm deletes the node.

Theorem 9. *The algorithm Move-Recursively-Forward with insertions and deletions is strictly 4-competitive for $\alpha = 1$, with transitive G , with insertions costing n and accesses before deletes.*

PROOF. Fix a sequence of requests σ of length m . We compare costs of MRF and an optimal offline algorithm OPT on σ using the potential function Φ , defined *twice the number of inversions* (as before). Let $C_{\text{MRF}}(t)$ and $C_{\text{OPT}}(t)$ denote the cost incurred at time t by MRF and OPT respectively.

The transitivity assumption enables insertion without node rearrangements: a feasible position for insertion always exists. Thus, for each insertion, both MRF and OPT pay exactly $C_{\text{OPT}}(t) = C_{\text{MRF}}(t) = n$. Note that MRF and OPT might place the node at different positions, thus we account for at most n inversions created, and consequently for the change in potential we have $\Delta\Phi(t) \leq 2n$. This gives us $C_{\text{MRF}}(t) + \Delta\Phi(t) \leq 3n \leq 3 \cdot C_{\text{OPT}}(t)$.

In case of a deletion request, exactly ℓ inversions are removed, and since $C_{\text{OPT}}(t) = k$ and $C_{\text{MRF}}(t) = k + \ell$, we have $C_{\text{MRF}}(t) + \Delta\Phi(t) \leq (k + \ell) - 2\ell \leq k \leq 1 \cdot C_{\text{OPT}}(t)$.

Similarly to the proof of Theorem 6, the amortized cost for an access request is $C_{\text{MRF}}(t) + \Delta\Phi(t) \leq 4 \cdot C_{\text{OPT}}(t)$. We sum up the amortized bounds for all requests of the sequence σ , of all types (access, insertion, deletion), and we conclude that MRF is 4-competitive. \square

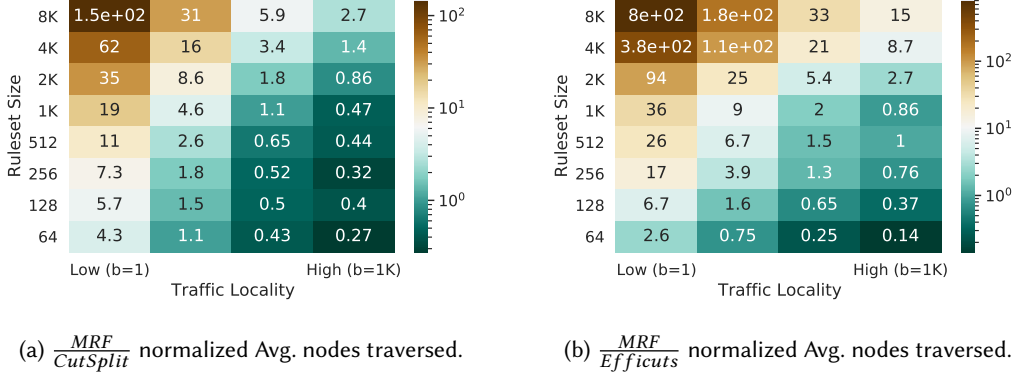


Fig. 5. MRF out-performs static decision tree based packet classifiers for small ruleset sizes and at high traffic locality (towards the bottom right region of the ruleset size vs locality dimensions).

7 EMPIRICAL EVALUATION

We now empirically evaluate the benefits and limitations of our self-adjusting packet classifier by comparing it to existing packet classifiers. We evaluate two implementation variants: MRF (Section 5.1) and MRF-Fast (Section 5.2). We investigate two questions:

Do self-adjustments improve the classification time? We show that compared to a static list, the self-adjusting list (MRF) improves the classification time by at least 2x on average and at least 10x better under high locality in traffic. Compared to hierarchical cut classifiers, under high locality and for small ruleset sizes, MRF improves classification time. Specifically, our results show that MRF’s classification time is 7.01x better than Efficuts [30] and 3.64x better than CutSplit [18] for small ruleset sizes and high traffic locality.

What is the benefit in memory consumption? Our evaluation shows that the simple data structure of the self-adjusting list classifier dramatically improves memory consumption: on average 10x lower than hierarchical cut classifiers. The implementation MRF-Fast (the second algorithm from Section 5) uses slightly more memory — but still lower than any decision tree based classifiers. This enables the time-memory tradeoff between MRF and MRF-Fast.

Figures 5a and 5b show the key findings of our evaluation, presenting the average nodes traversed (representing classification time) normalized to CutSplit and Efficuts. We observe that MRF’s classification time improves faster than CutSplit and Efficuts in the region of smaller rulesets and high traffic locality. However, MRF’s classification time degrades below the competition outside these regions.

7.1 Methodology

Rulesets and Packet traces: We use Classbench [29] to generate rulesets and traffic resembling real-world scenarios. We examine a wide range of rulesets sizes and traffic locality. To control traffic locality, we use Classbench’s parameter Pb : the Pareto distribution *scale* parameter [15].

Comparison with existing packet classifiers: We compare MRF and MRF-Fast against list-based and decision tree based packet classifiers. In our evaluations, a static-list serves as a baseline for list-based approach. Among wide range of decision tree based packet classifiers, our baselines include Hicuts [12], Hypercuts [25], Efficuts [30] and the more recent packet classifier CutSplit [18]

which combines cutting and splitting techniques. We use the default parameter settings for all our baselines.

Simulations: We built a custom simulator written in C++ and implemented all the baselines including MRF and MRF-Fast. We have faithfully merged the online available source code of our baselines into our simulator for a common ground of comparison. We additionally implemented the packet lookup function for Hicuts, Hypercuts and Efficuts². In the implementation of MRF-Fast we store the dependencies as a linked-list at each node. We did not apply the dependency structure minimization (thus the actual memory consumption by MRF-Fast may be lower). We ran our simulations on a server with 40 CPU cores (Intel(R) Xeon(R) Gold 6209U CPU @ 2.10GHz) and 192GB RAM. Specifically each pointer consumes 4 bytes of memory.

Metrics: We report two main metrics of interest (i) average classification time measured as average number of traversed nodes and (ii) memory consumed measured in Kilo Bytes (KB).

In measuring the nodes traversed, we count the number of nodes accessed during lookup, and the number of swapped nodes during rearrangement. For all our baselines, we only count the number of nodes accessed during lookup since they do not perform any rearrangements. For MRF, we simply add the nodes traversed during lookup and rearrangement: the cost of a packet match is comparable to the cost of operations that accompany the swap (checking a rule overlap). For MRF-Fast, we divide the rearrangement cost by a factor $\alpha = 5$ (see also Section 4): a packet match requires one comparison with each of the 5-tuple header fields, whereas the operations that accompany the swap requires only one comparison. All the values reported in our evaluation are averaged over 32 runs.

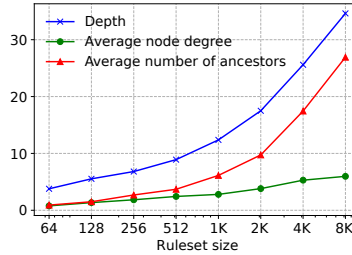


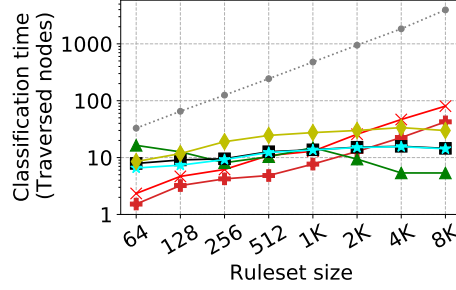
Fig. 6. Maximum node depth, average node degree, and average number of ancestors in the dependency graph increases with the ruleset size.

7.2 Results

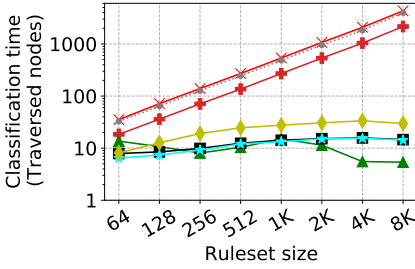
Before presenting our results, we analyze the characteristics of the rulesets used in our evaluations. Specifically, we are interested in the diversity of the dependency graph's structure across ruleset sizes. In Figure 6, we fix `acl1_seed` provided by Classbench and generate rulesets of sizes in the range 64 to 8192. We observe that some parameters increase with the ruleset size: maximum depth of nodes, average node degree and average number of ancestors. All three metrics of the dependency graph structure influence the classification time. The larger the average number of ancestors, the larger the classification time of Move-Recursively-Forward, at least for the memoryless implementation: the *direct dependency* operation traverses the entire list of ancestors of a moving node.

²The original source code of Hicuts, Hypercuts and Efficuts does not implement a packet lookup function, and instead estimates by the worst case: the maximum depth of the tree.

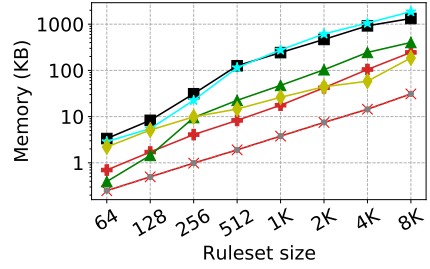
..... Static-List × MRF + MRF-Fast ▲ Effcuts ■ Hypercuts ◆ Hicuts ◆ Cutsplit



(a) Classification time under high-locality in traffic



(b) Classification time under low-locality in traffic



(c) Memory usage: 10x lower memory consumption by MRF

Fig. 7. The self-adjusting list-based packet classifier MRF outperforms decision tree-based algorithms under high locality in traffic for ruleset sizes up to 1K and significantly improves the memory requirements (10x on average). MRF-Fast trades memory for classification time, showing an improvement over MRF in classification time with slightly more memory requirements. Note the log scale in the figures.

MRF under high traffic locality. In Figure 7a, we show the average number of nodes traversed with a high locality in traffic for various ruleset sizes. For small ruleset sizes in the range 64 up to 1K, MRF outperforms in classification time compared to all our baselines. For ruleset size of 64: MRF performs 7.01x better than Effcuts, 3.64x better than CutSplit and 14.06x better than a static-list. For ruleset size of 1K: MRF performs 1.15x better than Effcuts, 2.12x better than CutSplit and 37.04x better than a static-list. For larger ruleset sizes ($> 1K$): MRF performs 15x worse compared to Effcuts and 2.7x worse compared to CutSplit, yet 49x better than a static-list.

For a small ruleset size (e.g., 64), we can see from Figure 6 that the average number of ancestors is much lower. This allows MRF to move the frequently matched rules closer to the head of the list which significantly improves classification time under high locality. For large ruleset sizes ($> 1K$) the average node ancestors grows up to 30 which does not allow moving the frequently matched rules closer to the head (a rule cannot be moved ahead of dependencies).

MRF under low traffic locality. We evaluate the performance of MRF under low locality in traffic even though MRF's design is not targeted for this case. A trace with low traffic locality in our evaluations consists of unique packets i.e., a packet arrives only once in a trace and each packet matches a rule in the ruleset uniformly at random. As a result, the number of nodes that MRF traverses on average is nearly half the ruleset size. In Figure 7b, we observe that the classification

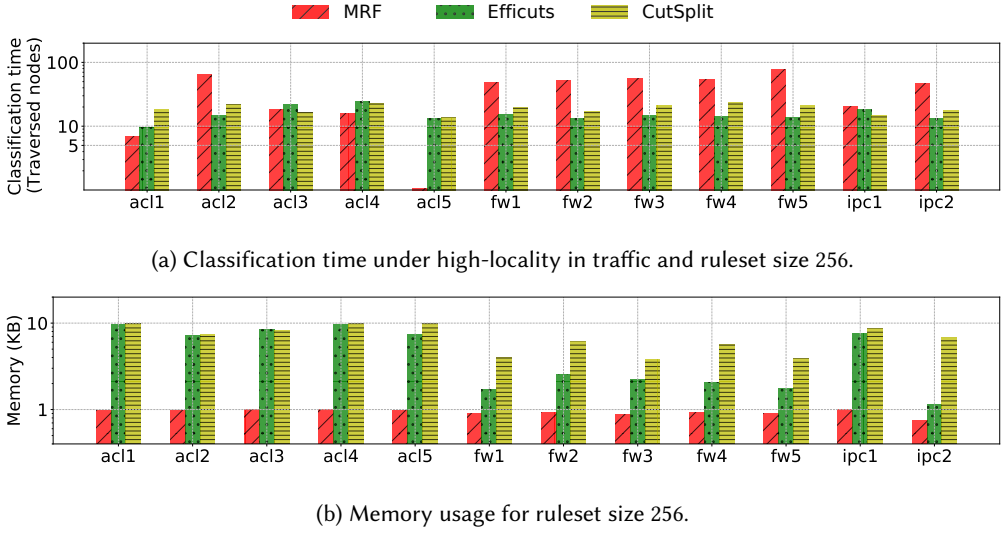


Fig. 8. For small rulesets, MRF’s classification time is on-par compared to Efficuts but with a significantly lower memory usage. Note the log scale in the figures.

time of MRF is comparable to a static list for all ruleset sizes. On the other hand, MRF-Fast slightly improves the classification time compared to MRF but still performs worse compared to decision tree based classifiers.

MRF memory consumption. MRF significantly reduces memory requirements for packet classification. The memory requirement of MRF reaches the lower bound for uncompressed data structures (see Section 5). In Figure 7c we show the memory requirements of different packet classifiers across various ruleset sizes generated using `acl1_seed`. We observe that MRF requires on average across various ruleset sizes 10.24x lower memory compared to Efficuts, 43.95x lower memory compared to Hypercuts, and 7.58x lower memory compared to CutSplit. In Figure 8b we further evaluate the memory requirements across all the ruleset seeds available in Classbench. We observe significant improvement in memory consumption while the classification time of MRF is on-par compared to Efficuts and CutSplit as shown in Figure 8a.

8 CONCLUSIONS AND FUTURE DIRECTIONS

We proposed a packet classifier design that adjusts to traffic and performs provably well. The viability of the solution improves as locality of traffic increases or the ruleset size decreases, as also demonstrated empirically.

This paper leaves open several research avenues. In particular, it would be interesting to evaluate our drop-in replacement for static lists of rules in existing systems (e.g., 5G packet detection rules), even in existing packet classifiers (e.g., in leaves of decision trees). It would also be interesting to refine our implementation, especially around the *direct dependency* operation, and study hybrid system designs, i.e., self-adjusting lists which may offload existing packet classifiers upon detecting that traffic exhibits high locality. Design opportunities may also arise for self-adjusting *trees* in packet classification.

To strengthen the theoretical analysis, we point beyond the pure competitive analysis of Move-Recursively-Forward. In the classic list access setting, the algorithms perform provably better under

locality of input (measured in *runs*, a metric specific to list access) [3], and empirical evidence collected in our work similarly suggests that our algorithm may improve too. In the *average-case analysis* for the stochastic setting, the competitive ratio of Move-To-Front improves from 4 to π [11] (in the paid exchange model), a behavior that the Move-Recursively-Forward algorithm may share.

REFERENCES

- [1] Susanne Albers. 1998. Improved Randomized On-Line Algorithms for the List Update Problem. *SIAM J. Comput.* 27, 3 (1998), 682–693.
- [2] Susanne Albers and Maximilian Janke. 2020. New Bounds for Randomized List Update in the Paid Exchange Model. In *Proc. of the International Symposium on Theoretical Aspects of Computer Science, STACS*, Vol. 154. Schloss Dagstuhl, 1–17.
- [3] Susanne Albers and Sonja Lauer. 2016. On list update with locality of reference. *J. Comput. Syst. Sci.* 82, 5 (2016), 627–653.
- [4] Christoph Ambühl. 2000. Offline List Update is NP-Hard. In *8th Annual European Symposium on Algorithms, ESA*, Vol. 1879. Springer, 42–51.
- [5] Yossi Azar and Leah Epstein. 2002. On-line scheduling with precedence constraints. *Discret. Appl. Math.* 119, 1-2 (2002), 169–180.
- [6] Ran Bachrach, Ran El-Yaniv, and M. Reinstadtler. 2002. On the Competitive Theory and Practice of Online List Accessing Algorithms. *Algorithmica* 32, 2 (2002), 201–245.
- [7] Marcin Bienkowski, Jan Marcinkowski, Maciej Pacut, Stefan Schmid, and Aleksandra Spyra. 2017. Online Tree Caching. In *Proc. of the ACM Symposium on Parallelism in Algorithms and Architectures, SPAA*. ACM, 329–338.
- [8] Allan Borodin and Ran El-Yaniv. 1998. *Online Computation and Competitive Analysis*. Cambridge University Press.
- [9] Marek Chrobak and Lawrence L. Larmore. 1991. The Server Problem and On-Line Games. In *On-Line Algorithms, Proc. of a DIMACS Workshop (DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 7)*. DIMACS/AMS, 11–64.
- [10] David Eppstein and Shan Muthukrishnan. 2001. Internet packet filter management and rectangle geometry. In *Proc. of the Twelfth Annual Symposium on Discrete Algorithms, SODA*. ACM/SIAM, 827–835.
- [11] Gaston H. Gonnet, J. Ian Munro, and Hendra Suwanda. 1979. Toward Self-Organizing Linear Search. In *Proc. of the 20th Annual Symposium on Foundations of Computer Science, FOCS*. IEEE Computer Society, 169–174.
- [12] Pankaj Gupta and Nick McKeown. 2000. Classifying packets with hierarchical intelligent cuttings. *IEEE Micro* 20, 1 (2000), 34–41.
- [13] Pankaj Gupta and Nick McKeown. 2001. Algorithms for packet classification. *IEEE Network* 15, 2 (2001), 24–32.
- [14] Hazem H. Hamed and Ehab Al-Shaar. 2006. Dynamic rule-ordering optimization for high-speed firewall filtering. In *Proc. of the ACM Symposium on Information, Computer and Communications Security, ASIACCS*. ACM, 332–342.
- [15] Jonathan R.M. Hosking and James R. Wallis. 1987. Parameter and Quantile Estimation for the Generalized Pareto Distribution. *Technometrics* 29, 3 (1987), 339–349.
- [16] Shahin Kamali and Alejandro López-Ortiz. 2013. A Survey of Algorithms and Models for List Update. In *Space-Efficient Data Structures, Streams, and Algorithms*, Vol. 8066. Springer, 251–266.
- [17] Kirill Kogan, Sergey I. Nikolenko, Ori Rottenstreich, William Culhane, and Patrick Eugster. 2014. SAX-PAC (Scalable And eXpressive Packet Classification). In *ACM SIGCOMM*. ACM, 15–26.
- [18] Wenjun Li, Xianfeng Li, Hui Li, and Gaogang Xie. 2018. CutSplit: A Decision-Tree Combining Cutting and Splitting for Scalable Packet Classification. In *IEEE Conference on Computer Communications, INFOCOM*. 2645–2653.
- [19] John McCabe. 1965. On serial files with relocatable records. *Operations Research* 13, 4 (1965), 609–618.
- [20] Sebastiano Miano, Matteo Bertrone, Fulvio Rizzo, Mauricio Vásquez Bernal, Yunsong Lu, and Jianwen Pi. 2019. Securing Linux with a faster and scalable iptables. *ACM SIGCOMM Computer Communication Review, CCR* 49, 3 (2019), 2–17.
- [21] Dennis M. Moyles and Gerald L. Thompson. 1969. An Algorithm for Finding a Minimum Equivalent Graph of a Digraph. *J. ACM* 16, 3 (1969), 455–460.
- [22] ONF. 2013. OpenFlow Reference Release. <https://github.com/mininet/openflow>.
- [23] Nick Reingold, Jeffery R. Westbrook, and Daniel Dominic Sleator. 1994. Randomized Competitive Algorithms for the List Update Problem. *Algorithmica* 11, 1 (1994), 15–32.
- [24] Nadi Sarrar, Steve Uhlig, Anja Feldmann, Rob Sherwood, and Xin Huang. 2012. Leveraging Zipf’s law for traffic offloading. *ACM SIGCOMM Computer Communication Review, CCR* 42, 1 (2012), 16–22.
- [25] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. 2003. Packet classification using multidimensional cutting. In *Proc. of the Conference on Applications, technologies, architectures, and protocols for computer communications, ACM SIGCOMM*. ACM, 213–224.

- [26] Daniel D. Sleator and Robert E. Tarjan. 1985. Amortized Efficiency of List Update and Paging Rules. *Commun. ACM* 28, 2 (1985), 202–208.
- [27] Daniel D. Sleator and Robert E. Tarjan. 1985. Self-Adjusting Binary Search Trees. *J. ACM* 32, 3 (1985), 652–686.
- [28] Venkatachary Srinivasan, Subhash Suri, and George Varghese. 1999. Packet Classification Using Tuple Space Search. In *Proc. of the ACM SIGCOMM*. ACM, 135–146.
- [29] David E. Taylor and Jonathan S. Turner. 2007. ClassBench: A Packet Classification Benchmark. *IEEE/ACM Transactions on Networking* 15, 3 (2007), 499–511.
- [30] Balajee Vamanan, Gwendolyn Voskuilen, and T. N. Vijaykumar. 2010. EffiCuts: optimizing packet classification for memory and throughput. In *Proc. of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, ACM SIGCOMM*. ACM, 207–218.
- [31] Liang Zhao, Yuji Inoueand, and Hideo Yamamoto. 2004. Delay Reduction for Linear-Search Based Packet Filters. In *Proc. of the International Technical Conference on Circuits/Systems,Computers and Communication, ITC-CSCC*. 160–163.

A ONLINE ALGORITHMS AND COMPETITIVE ANALYSIS

An adversary reveals a sequence of requests σ one-by-one, in an online fashion. Upon seeing a request, the algorithm must serve it without the knowledge of future requests. The decisions of the algorithm are permanent: it must be consistent with decisions from the past. For an overview of the competitive analysis framework, we refer to books [8].

Competitive ratio. We measure the performance of an online algorithm by comparing its cost with the cost of an optimal offline algorithm OPT . Formally, let $\text{ALG}(\sigma)$, resp. $\text{OPT}(\sigma)$, be the cost incurred by a deterministic online algorithm ALG , resp. by an optimal offline algorithm, for a given sequence of requests σ . In contrast to ALG , which learns the requests one-by-one as it serves them, OPT has complete knowledge of the entire request sequence σ *ahead of time*. The goal is to design online algorithms with worst-case guarantees for the ratio. In particular, ALG is said to be *c-competitive* if there is a constant b , such that for any input sequence σ it holds that

$$\text{ALG}(\sigma) \leq c \cdot \text{OPT}(\sigma) + b.$$

Note that b cannot depend on input σ but can depend on other parameters of the problem, such as the number of nodes. The minimum c for which ALG is *c-competitive* is called the *competitive ratio* of ALG . We say that ALG is *strictly c-competitive* if additionally $b = 0$. All algorithms in this paper are strictly competitive.

The paper treats mostly about deterministic online algorithms, but in some aspects remarks about *randomized* online algorithms; for an introduction we refer to [8].