Self-Adjusting Partially Ordered Lists

Vamsi Addanki*, Maciej Pacut*, Arash Pourdamghani*, Gabor Rétvári†, Stefan Schmid‡, Juan Vanerio§
*TU Berlin, Germany
†BME, Hungary and Ericsson Research, Hungary
‡TU Berlin, Germany and Fraunhofer SIT
§University of Vienna, Austria

***Abstract*—We introduce self-adjusting partially ordered lists, a generalization of self-adjusting lists where additionally there may be constraints for the relative order of some nodes in the list. The lists self-adjust to improve performance while serving input sequences exhibiting favorable properties, such as locality of reference, but the constraints must be respected.**

We design a deterministic adjusting algorithm that operates without any assumptions about the input distribution and without maintaining frequency statistics or timestamps. Despite the more general model, we show that our deterministic algorithm performs closely to optimum (it is 4-competitive). In addition, we design a family of randomized algorithms with improved competitive ratios, handling also a more general rearrangement cost model, scaled by an arbitrary constant $d \geq 1$. Moreover, we observe that different constraints influence the competitiveness of online algorithms, and we shed light on this aspect with a lower bound.

We investigate the applicability of our self-adjusting lists in the context of network packet classification. Our evaluations show 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.

I. Introduction

Self-adjusting data structures adapt their internal structure to the input sequence with the aim to reduce the request processing time. Self-adjusting data structures feature a reorganization algorithm that runs after each operation and adapts to any input without knowing it a priori. In comparison to static data structures, their self-adjusting counterparts experience overhead, which however is often eclipsed by the gains from adaptation to input. Popular data structures of this kind are self-adjusting lists [30] and splay trees [31].

Self-adjusting lists are traditionally analyzed in the context of online algorithms, where the problem is referred to as *online list access problem* [7, Ch. 1 and 2], one of the most fundamental problems in online algorithms. The problem was studied for decades [21, 27, 30, 34, 26, 2] and remains an active field of research [3]. Over years, the community studied the problem under various cost models [23, 18] and generalizations [24, 13]. In a nutshell, in the classic list access problem (without the partial order), we manage a linked list data structure in which accessing a node costs proportionally to its distance from the head of the list. An online algorithm may rearrange the list to decrease the access cost, but a rearrangement has its own cost for node transpositions. The goal of an online algorithm is to minimize the total cost of access and rearrangements.

This paper initiates the study of a natural generalization of self-adjusting lists, where additionally we are given a partial order among the nodes of the list. Each relation (u, v) in the partial order yields a constraint: u must appear before v in any configuration of the list. See Figure 1 for an example of a configuration of the list complying with a partial order.

To illustrate where such data structure could be useful, consider an information retrieval problem where we search a set of documents to find a piece of information. However, certain documents may be overridden by a new, related document, so to always find up-to-date information, the documents must always be accessed in a chronological order. A self-adjusting list respecting the chronological order is a suitable data structure for this application, and it can reduce the time for accessing popular documents, while always returning their newest version.

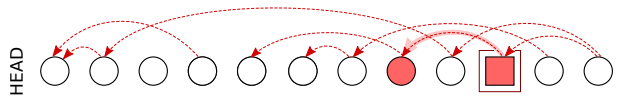


Fig. 1: An example of a list configuration with a partial order among the nodes. The requested node, depicted as a square, cannot move forward beyond the solid node due to a precedence constraint (highlighted with a bold arrow). If we choose to move the requested node close to the head of the list, we must first move the solid node, which however has its own constraints we must account for.

Another practical motivation arises in the context of *network packet classification* [16], where self-adjustments enable adaptation to traffic but rule priorities and overlaps introduce precedence constraints (we elaborate later in this section). More broadly, the problem models flexible processing pipelines in decision systems, where constraints realize part of the logic, and assembly lines where some stages must be finished before others.

The consideration of constraints poses algorithm design challenges not present in classic self-adjusting lists. For example, how to design an efficient procedure that moves the requested node closer to the front of the list? What if the node is blocked, as in Figure 1 — should we move the blocking node as well, and to what extent? Well-known online algorithms for the list access problem, such as Move-to-Front [30] and TIMESTAMP [2] can flexibly move nodes towards the head of the list upon access. In contrast, constraints may prevent such optimizations or at least make them costly.

This generalization of online list access raises fundamental questions about its competitiveness:

- Does the introduction of constraints actually make the problem harder or easier from the standpoint of competitive analysis? Which algorithmic techniques from classic list access can we use, and which become obsolete? Do classic lower bounds hold even with constraints?
- How does the structure of the partial order influence the competitive ratio? For example, if no nodes can move, the competitive ratio is 1. If the partial order consists of two disjoint chains of length $n/2$, then a simple static strategy that interleaves two chains is 2-competitive, and no deterministic algorithm can be better than 1.5-competitive. With no dependencies, we can directly carry lower bounds [3, 26] from classic list access in the P^d model. For example, if the directed acyclic graph describing the precedence constraints has depth Δ, then it is easy to see that a strategy that always moves the blocking nodes forward is at best $\Omega(\Delta)$-competitive.

In this work, we are interested in *competitive analysis* [7] of deterministic and randomized algorithms for online partially ordered list access. We design deterministic and randomized online algorithms, built around a simple recursive procedure that efficiently moves a carefully chosen set of nodes forward. Furthermore, we study how various partial orders influence the competitiveness of deterministic algorithms.

A. Practical Motivation: Self-Adjusting Packet Classification

Packet classification: In communication networks, packet classification [16] is one of the operations executed for each packet, at every node of the network: at switches, routers, middleboxes (e.g., firewalls), and end hosts. In a nutshell, packet classification assigns a label to each packet, determining, e.g., whether the sender of the packet can access the intended destination, or via which interface the packet should be forwarded to reach its destination. Consequently, it enables fundamental network functions such as access control, packet forwarding, quality of service, accounting and more. Packets are classified according to a set of *packet classification rules* (see Figure 2 for an example), and each rule consists of a rule priority, a filter expression on packet header fields, and an action that assigns a label. Classifying a packet in this setting requires finding the highest priority rule that matches the packet and applying the label determined by this rule.

Packet classifiers that adapt to traffic: Existing packet classifiers typically have internal non-adaptive data structures designed for good performance under uniform traffic patterns (e.g., lists, tries, hash tables, bit vectors, or decision trees [16, 32, 11], as well as TCAM hardware solutions). We present initial experimental results in § VI suggesting that the performance of packet classification (e.g., reaction time and throughput) can be greatly improved by an adaptive solution, especially on non-uniform workloads, i.e., when the majority of traffic can be served with just a few rules [28].

Our approach to self-adjusting packet classifiers 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 [30].

Packet classifiers, however, additionally 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 determines the action uniquely. We refer to this challenge as *inter-rule dependencies*, and we note that usually, we have few of them [19].

Packet classification with self-adjusting partially ordered lists: We propose a self-adjusting list packet classifier, where we organize the classification rules in a linked list. To classify an incoming packet, we traverse the list of rules, searching for the first rule that the packet matches with (with this perspective, classifying a packet can be seen as the *access* operation for a node in a list). Overlaps between the rules together with rule priorities introduce precedence constraints among the rules, see Figure 2 for an example. The rule that matched the packet is then moved closer to the head of the list to speed up future matches, but dependencies need to be respected. Due to its simplicity, our algorithm can be a drop-in replacement for static list packet classifiers.

B. Contributions

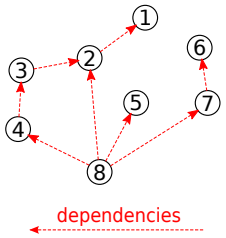
The main technical contribution of this paper is the design of constant-competitive deterministic and randomized algorithms for online partially ordered list access. The deterministic algorithm is simple, memoryless, and 4-competitive in $d = 1$ case (the case fitting the application of packet classification). The randomized algorithms successfully generalize a family of Markov algorithms from classic list access [14] (including Move-To-Front, BIT, COUNTER, RANDOM-RESET [30, 26]) without degrading the competitiveness. The randomized algorithms handle arbitrary $d \geq 1$, and for $d = 1$ we have a 2.64-competitive algorithm.

We shed light on how different partial orders influence competitiveness. By generalizing the argument of Reingold et al. [26], we show that the lower bound of 3 against deterministic algorithms applies to the setting with partial orders, even for pairwise independent nodes which have multiple dependencies themselves.

We identify applications of self-adjusting partially ordered lists in the context of *network packet classification* [16]. Our algorithms interpreted in this context give rise to *self-adjusting packet classifiers*, which adapt to the traffic they serve.

II. Model

We introduce the *online partially ordered list access* problem. In this problem, our task is to manage a self-adjusting



N	Proto	Src IP	Dst IP	Src Port	Dst Port	Action
1	TCP	10.12.12.0/24	20.0.0.1/32	ANY	80	DENY
2	TCP	0.0.0.0/0	20.0.0.1/32	ANY	80	ACCEPT
3	IP	0.0.0.0/0	20.0.0.1/32			DENY
4	UDP	0.0.0.0/0	0.0.0.0/0	1000-2000	1000-2000	ACCEPT
5	UDP	20.0.0.0/24	10.0.10.0/24	ANY	3306	ACCEPT
6	TCP	10.12.12.0/24	0.0.0.0/0	21	21	DENY
7	IP	10.0.0.0/16	20.0.0.0/20			ACCEPT
8	IP	0.0.0.0/0	0.0.0.0/0			DENY

Fig. 2: An example of a table of packet classification rules (right) and the corresponding partial order (left). In the table fields, N is the rule number and priority, fields Proto, Src IP, Dst IP, Src Port, and Dst Port are packet header fields, and Action determines the packet label to apply if the rule matches. A feasible method to classify a packet is to match it against the rules, starting from the top, and to apply the label of the first matching rule. This method is flexible: some rules can be reordered, and matching the packet to them one-by-one in the new order still applies the same labels to packets. The partial order hints if rearranging the rules results is correct. For example, rule 6 may move in front of rule 5, and each packet would be classified correctly, but moving rule 8 to the front of the list would result in dropping all packets.

linked list serving a sequence of requests, with minimal access and rearrangement costs and accounting for precedence constraints induced by a given partial order. If the partial order is empty, the problem is equivalent to classic online list access [30].

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 a request to a node in the list, an algorithm searches linearly through the list, starting from the head of the list. Accessing the node at position i in the list costs i (the 1st node is at position 1).

The partial order: In the beginning, the algorithm is given a partial order $\mathcal{P}$, which remains unchanged throughout the execution of the algorithm. The partial order induces precedence constraints among the nodes of the list. We say that a node v is *dependent on* a node u if there exists a relation (u, v) in the partial order $\mathcal{P}$, and then, v must be in front of u in every feasible configuration of the list. We assume that the given initial configuration of the nodes obeys the partial order.

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 the partial order. Each transposition incurs the cost 1.

The goal of the online algorithm is to minimize the total cost of accesses and node rearrangements. In this paper, we are interested in competitive online algorithms for this problem.

III. A Deterministic Algorithm

We propose a simple deterministic algorithm for online partially ordered list access. We argue that the algorithm is 4-competitive in the P^1 model (the general P^d model is considered in § IV). The algorithm can be viewed as a generalization of Move-To-Front [30] to partially ordered lists. If the partial order is empty, the algorithm is equivalent to Move-To-Front. Similarly to Move-To-Front, our algorithm satisfies the definition of a *memoryless online algorithm* [9].

The algorithm is designed around a recursive procedure that we run for the requested node. In a nutshell, we move the node forward until we encounter a node that is blocking the moving node, or the head of the list. If we encounter the blocking node, we recursively start moving the blocking node instead.

To define MRF, we introduce the concept of a blocking ancestor. An *ancestor* relation in a partial order is an extension of the parent-child relation to indirect relations, known as the transitive closure [10]. For each node with a non-empty set of ancestors in the partial order $\mathcal{P}$, we distinguish a node that is first to block the node's movement forward in the list: a node y is the *blocking ancestor* of a node z if z is dependent on y, and among all nodes z depends on, y is the furthest from the head.

We present the pseudocode of MRF in Algorithm 1. By $\mathsf{pos}(z)$ 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). When the algorithm moves the node to a certain position, it performs a sequence of swaps until the desired position is reached. In Figure 3, we depict an example run of MRF after serving a request.

Algorithm 1: The algorithm Move-Recursively-Forward for a partial order $\mathcal{P}$.

Input: An access request to node σ_t

```
Access σ_t
Run the procedure MRF(σ_t)
procedure MRF(y):
    if y has no ancestors in P then
        Move y to the front of the list
    else
        Let z be the blocking ancestor of y in P
        Move node y to pos(z) + 1
        Run the procedure MRF(z)
    end
```

A. The Blocking Chain

Next, we collectively reason about all the nodes that move after serving a request, referred to as the *blocking chain* of the requested node. The blocking chain is the central concept in the analysis of all algorithms in this paper.

Fix the configuration of MRF right before serving the request to a node σ_t. The *blocking chain* is the sequence of

nodes constructed by iteratively collecting the set of blocking ancestors of σ_t, constructed as follows. Initially, the chain contains σ_t, and in a single step, we determine the blocking ancestor of the head of the chain, and we insert the blocking ancestor to the front of the chain; we repeat until the head of the chain has no ancestors. We denote the blocking chain by $\mathbf{b}$, its length by B, and we emphasize that $\mathbf{b}$ contains the requested node at the last position, $\sigma_t = \mathbf{b}_B$ (see Figure 3).

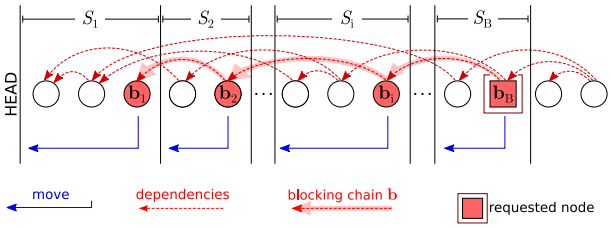


Fig. 3: An example of handling a request by the algorithm MRF. The blocking chain is denoted by nodes $\mathbf{b}_i$, with the dependencies between them distinguished by bold arrows. Each node of the blocking chain moves right behind its blocking ancestor and this movement is depicted with a blue arrow under the list. Furthermore, we illustrate sets of nodes S_i that we use in the analysis.

MOVE-RECURSIVELY-FORWARD strikes a balance between the access and rearrangement costs: the algorithm exchanges no more pairs than the position of the accessed node (Lemma 1). We claim that the reconfiguration cost of MRF is linear in terms of the access cost, and multiple recursive calls decrease the cost. Intuitively, each node from the blocking chain moves through a disjoint part of the list, in total at most $\mathsf{pos}(y)$. For a graphical argument, we refer to Figure 3.

Lemma 1. *Consider a single request to a node y at position $\mathsf{pos}(y)$ handled by the algorithm* MOVE-RECURSIVELY-FORWARD. *Then, the number of transpositions after serving the request is $\mathsf{pos}(y) - B$, where B is the length of the blocking chain of y.*

Proof. Let $\mathbf{b}_i$ for $1 \leq i \leq B$ be the blocking chain of the node y. Each node $\mathbf{b}_i$ moves to a position *one place behind* its blocking ancestor $\mathbf{b}_{i-1}$, except the node $\mathbf{b}_1$, which moves to the front of the list. The total number of transpositions is

$$\sum_{i=2}^{B}(\mathsf{pos}(\mathbf{b}_i)-(\mathsf{pos}(\mathbf{b}_{i-1})+1))+(\mathsf{pos}(\mathbf{b}_1)-1) = \mathsf{pos}(\mathbf{b}_B)-B.$$

As $y = \mathbf{b}_B$, we conclude that the lemma holds. □

B. Inversions and the Potential Function

Given a partial order, and two list configurations respecting the partial order, is it always possible to reach one from the other, and if so, at what cost? To answer this question, we revisit the concept of *inversions*, used in the analysis of Move-To-Front [30] to study their interaction with partial orders.

An *inversion* between two lists L_1 and L_2 is an ordered pair of nodes (u, v) such that u is located before v in L_1, and u is located after v in L_2. We claim that the distance (in the number of order-respecting transpositions) between partially ordered lists is equivalent to the number of inversions between them:

Lemma 2. *Consider two lists L_1, L_2 consisting of the same set of nodes and obeying a partial order $\mathcal{P}$. Then, the minimum number of transpositions respecting $\mathcal{P}$ required to transform L_1 to L_2 is equal to the number of inversions between L_1 and L_2.*

Proof. First, we show that transforming L_1 to L_2 is always possible without violating the partial order $\mathcal{P}$ at any transient configuration, and the number of transpositions required is at most the number of inversions.

Consider the following recursive strategy of transforming L_1 into L_2. Let v be the node at the front of L_2. Being the first node in L_2, the node v is not dependent on any other node from L_2. Since L_1 and L_2 share the same partial order $\mathcal{P}$, the node v can move to the front of L_1 as well, without violating any precedence constraints. For each node u in front of v in L_1, we have an inversion (u, v), and moving v to the front incurs the cost equal to the number of (u, v) inversions. Then, we remove v from both lists and recursively apply this procedure until the lists are empty. The minimum number of transpositions to transform L_1 to L_2 is no smaller than with the described procedure.

Second, the minimum number of transpositions is at least the number of inversions, since a single transposition can decrease the number of inversions by at most one. We showed inequalities both ways, and hence we conclude that the claim holds. □

Potential function: Next, we define the potential function used throughout the analysis of MRF. Fix any optimal offline algorithm OPT. Let Φ be a function from a tuple of MRF' and OPT's lists to non-negative integers:

$$\Phi = 2 \cdot I,$$

where I is the number of inversions between MRF's list and OPT's list. Let $\Phi(t)$ denote the value of Φ right after serving t-th request. For succinctness, in the remainder of this paper we use the notion of inversions in this narrower meaning, for comparing the lists of MRF and OPT.

Lemma 2 supports our potential function choice. The potential function characterizes the distance between the online algorithm's list and a fixed optimal offline algorithm's list in terms of the number of order-respecting transpositions.

C. How Do Rearrangements Affect Inversions?

In the analysis of Move-To-Front [30], Sleator and Tarjan studied the influence of moving the requested node to the front of the list on inversions between Move-To-Front's list and OPT's list. To this end, they defined the values k and ℓ related to the number of nodes in front of the requested node σ_t in online algorithm's and OPT's list. We refer to these values in our analysis: 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.

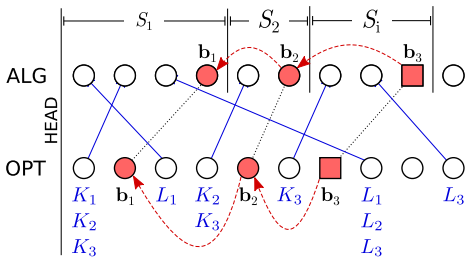


Fig. 4: This example illustrates central definitions of sets of nodes used in our analysis. We depict the positions of nodes in both MRF's and OPT's list (joined by solid blue lines). The dotted black lines between the nodes $\mathbf{b}_i$ help in determining the assignment of nodes to sets: in K_i we have the nodes in front of the dotted black line between $\mathbf{b}_i$, and in L_i we have the nodes that cross the dotted black lines between $\mathbf{b}_i$'s.

In Move-To-Front, the change in inversions after serving the request is $k - \ell$. The design goal of MOVE-RECURSIVELY-FORWARD is to arrive at the same conclusions for the change in inversions, see Theorem 3. Our argument requires careful analysis of the rearranged nodes, as many of them move after serving the request.

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. For the purpose of a fine-grained analysis of our rearrangement operation, we introduce the generalizations of the values of k and ℓ to sets K_j and L_j, defined for each node of the blocking chain.

Sets K_j and L_j: Precisely, let K_j be the set of elements before $\mathbf{b}_j$ in both MRF's and OPT's lists for $j \in [1, B]$, and let L_j be the set of elements before $\mathbf{b}_j$ in MRF's list but after $\mathbf{b}_j$ in OPT's list. We note that these sets are generalizations of k and ℓ: for the requested node $\mathbf{b}_B$ we have $k = |K_B|$ and $\ell = |L_B|$.

Sets S_j: The sets of nodes between the nodes $\mathbf{b}$ in MRF's list are crucial to the analysis. Intuitively, the node $\mathbf{b}_i$ 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 $\mathbf{b}_1$ (included). For $j \in [2, B]$, let S_j be the set of nodes between $\mathbf{b}_j$ and $\mathbf{b}_{j-1}$ (with $\mathbf{b}_{j-1}$ excluded) in MRF's list.

Figure 4 illustrates an example of possible composition of sets K_j, L_j and S_j for different values of j on a given request.

Consider a single request and the sequence of rearrangements of MOVE-RECURSIVELY-FORWARD that follows it. We study the influence of the rearrangements on the change in the potential function. To this end, we separately bound the number of introduced and destroyed inversions. The Figure 4 assists in illustrating the graphical arguments used in this section.

Theorem 3. *Consider a request to the node σ_t with a blocking chain of length B, and fix a configuration of* OPT *at time t. Then, the change in the number of inversions after serving the request by MRF is at most $k - \ell - B + 1$.*

To prove this claim, we consider the influence of the Move-Recursively-Forward operation on values k and ℓ (defined for the currently requested node) by inspecting the sets K_j and L_j (defined for the nodes $\mathbf{b}_j$). We separately bound the number of inversions created (Lemma 6) and destroyed (Lemma 7). Before showing these claims, we inspect the basic relations between the sets K_j, L_j and S_j (Lemmas 4 and 5).

Lemma 4. *Consider a request to a node and its blocking chain of length B. Then, the following relations hold:*

1) $\bigcup_{j=1}^{B} K_j = K_B$,
2) $\bigcup_{j=1}^{B} (S_j \cap L_j) = \bigcup_{j=1}^{B} L_j$.

Proof. First, we prove the equality 1. We show inclusions both ways. Note that the order between nodes from $\mathbf{b}$ is the same in both MRF's and OPT's lists. Hence, a node $y \in K_i$ is in front of $\mathbf{b}_i$ and in front of all $\mathbf{b}_j$ 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 $\bigcup_{j=1}^{B} K_j \subseteq K_B$.

Conversely, $K_B \subseteq \bigcup_{j=1}^{B} K_j$ 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 $\{S_j\}$ 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 $\mathbf{b}_i$ in MRF's list,
- y is after $\mathbf{b}_j$ in OPT's list (it belongs to L_j), and hence it is after $\mathbf{b}_i$ in OPT's list (the order of $\mathbf{b}$ is fixed due to precedence constraints).

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

Lemma 5. *Consider a request to a node and its blocking chain of length B. Then, the following relations hold:*

1) $\sum_{j=1}^{B} |K_j \cap S_j| \leq k - B + 1$,
2) $\sum_{j=1}^{B} |L_j \cap S_j| \geq \ell$.

Proof. First, we prove the equality 1. The nodes of the blocking chain $\mathbf{b}_1, \ldots, \mathbf{b}_{B-1}$ belong to K_B but not to $S_j \cap K_j$ for any $j \in [1, ..., B]$, thus

$$|\bigcup_{j=1}^{B} (S_j \cap K_j)| \leq |\bigcup_{j=1}^{B} K_j| - B + 1 = k - B + 1,$$

where the equality follows by Lemma 4, equation 1 and the definition of k. Second, we prove the equality 2. We have the following chain of inequalities

$$\sum_{j=1}^{B} |S_j \cap L_j| = |\bigcup_{j=1}^{B} (S_j \cap L_j)| = |\bigcup_{j=1}^{B} L_j| \geq |L_B| = \ell,$$

where the first step holds as the sets S_j are disjoint, the second step follows by Lemma 4, equation 2, the inequality follows

by basic properties of sets, and the last step follows by the definition of ℓ. □

Lemma 6. *Consider a request to a node σ_t with a blocking chain of length B, and fix a configuration of* OPT *at time t. Then, due to the rearrangements after serving the request,* MRF *creates at most $k - B + 1$ inversions.*

Proof. Let I_j^+ be the number of inversions added by moving a single node $\mathbf{b}_j$ by MRF, for $j \in [1, B]$. To bound I_j^+, we inspect the set S_j of the nodes that $\mathbf{b}_j$ overtakes, and we reason based on their positions in OPT's list. Moving $\mathbf{b}_j$ 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 $\mathbf{b}_j$. 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

$$\sum_{j=1}^{B} I_j^+ \leq \sum_{j=1}^{B} |S_j \cap K_j| = |\bigcup_{j=1}^{B} (S_j \cap K_j)| = k - B + 1,$$

where the first equality holds as the sets S_j are disjoint, and the last step follows by Lemma 5, equation 1. □

Lemma 7. *Consider a request to the node σ_t, and fix a configuration of* OPT *at time t. Due to rearrangements after serving the request,* MRF *destroys at least ℓ inversions.*

Proof. Let I_j^- be the number of inversions destroyed by moving a single node $\mathbf{b}_j$ by MRF, for $j \in [1, B]$. To bound I_j^-, we inspect the set S_j of the nodes that $\mathbf{b}_j$ overtakes, and we reason based on their positions in OPT's list. Moving $\mathbf{b}_j$ 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 $\mathbf{b}_j$. 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.

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

where the equality holds as the sets S_j are disjoint, and the last step follows by Lemma 5, equation 2. □

Combining Lemmas 6 and 7 gives us the joint bound on the change in the number of inversions and proves the Theorem 3. We note that this bound is consistent with the bound on the changes in inversions for the algorithm Move-To-Front [30], where the inversions were considered with respect to the requested node only.

D. Bounding the Competitive Ratio

The observations from previous subsections enable us to directly repeat the potential function argument of Sleator and Tarjan for Move-To-Front [30].

Theorem 8. *The algorithm* MRF *is strictly 4-competitive in the P^1 model.*

The proof repeats the arguments of Sleator and Tarjan for Move-To-Front, and internally we use a generalized argument concerning inversions (Theorem 3), which handles the generalized recursive rearrangement procedure.

Proof. 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.

We compare the costs of MRF and an optimal offline algorithm OPT on σ using the potential function Φ (cf. § III-B), distinguishing between two types of events. 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 request 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.

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 a request 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 $\mathsf{pos}(\sigma_t)$, and by Lemma 1 the rearrangement cost is bounded by $\mathsf{pos}(\sigma_t)$, hence $C_{\text{MRF}}(t) \leq 2 \cdot \mathsf{pos}(\sigma_t)$.

Next, we bound the amortized cost for every request served by MRF. The amortized cost is $C_{\text{MRF}}(t) + \Delta\Phi(t)$ for each time t. By Theorem 3, we bound the change in the number of inversions due to MRF's rearrangement after serving the request at time t by $\Delta I \leq k - \ell - B + 1 \leq k - \ell$. Thus, the change in the potential is $\Delta\Phi(t) \leq 2(k - \ell)$. As $\mathsf{pos}(\sigma_t) = k + \ell + 1$, combining these bounds gives us

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

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 at most 1, which increases the LHS by at most 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 $C_{\text{MRF}}(\sigma) \leq 4 \cdot C_{\text{OPT}}(\sigma)$. □

We note that the deterministic algorithm was analyzed in the P^1 model, where each transposition costs 1. In the

next section, we consider randomized algorithms, where we account for arbitrary d.

IV. Randomized Algorithms

We sketch a family of constant-competitive online algorithms that generalize Markov algorithms from classic list access [14] to the partially ordered list access (including Move-To-Front, BIT, COUNTER, RANDOM-RESET [30, 26]). Although the best algorithm from this family is RANDOM-RESET, we generalize a broader family of Markov algorithms. The competitive ratios of our algorithms generalized to partial orders remain equal to Markov algorithms from classic list access, and an enabler for this result is the concept of hidden inversions, overviewed in the next subsection. In contrast to the deterministic algorithm from § III that treated about $d = 1$, now we handle arbitrary exchange cost $d \geq 1$.

The generalized algorithms mix the concepts from Markov algorithms and deterministic Move-Recursively-Forward algorithm. Each node maintains a state, represented as a Markov chain; the node moves only if the Markov chain reaches a distinguished state. To guarantee independence of states of the nodes, we advance states of all nodes in the list, but we may move only the nodes included in the blocking chain (defined in § III) of the requested node. Our randomized algorithms employ a randomized version of the recursive procedure handling the blocking chain, where the movement of each node in the chain is independent of other nodes. MMRF is a family of randomized algorithms, with each algorithm characterized by an irreducible Markov chain maintained for every node in the list.

Algorithm 2: The algorithm Markov-Move-Recursively-Forward for a partial order $\mathcal{P}$.

Initialization : The Markov chain for each node in N is initialized according to the stationary distribution π.

Input: An access request to node σ_t

```
Access σ_t
for each node x in the list do
    Advance the Markov chain of x
end
Run the procedure MMRF(σ_t)
procedure MMRF(y):
    if y has no ancestors in P then
        if STATE(y) is 0 then
            Move y to the front of the list
        end
    else
        Let z be the blocking ancestor of y in P
        if STATE(y) is 0 then
            Move node y to pos(z) + 1
        end
        Run the procedure MMRF(z)
    end
```

Theorem 9. *Let M be an irreducible Markov chain. The* MMRF *algorithm that operates on M has a competitive ratio that is upper bounded by $\max\{1+\pi_0 \cdot (2d+T), 1+\frac{T}{d}\}$ against the oblivious adversary, where T denotes the expected hitting time to state 0, given by $T = \sum_{i=0}^{s-1} \pi_i \cdot h_i$.*

We defer the proof to the full version of the paper. The Theorem 9 matches the best known competitive bounds for self-adjusting lists without dependencies [26, 14].

V. A lower bound

The competitiveness of the problem varies with the given partial order. We examine this trend in the deterministic setting. The trivial case, where the partial order is complete, no nodes can move, for both the online algorithm and the offline algorithm, results in the competitive ratio of 1. This stands in contrast to the setting inherited from classic list access (equivalent to the empty partial order case) and we have a lower bound that approaches 3 as the number of nodes grow [26].

Besides the above corner cases, the scenarios with the competitiveness in-between exist. Consider a partial order that consists of two disjoint chains of length $n/2$ each. Then a static strategy that interleaves the nodes of the chains and does not move them further is 2-competitive. For this partial order, a lower bound of 1.5 exists, as a consequence of the mentioned lower bound [26], applied to the two independent nodes (heads of the chain).

A question arises, does adding constraints always lead to lower competitive ratios? It is easy to see that for partial orders that contain enough nodes with no constraints whatsoever: the lower bound still approaches 3 as the number of such nodes grows. However, such a case can be viewed as degenerated, hence we look into the competitiveness of a less-trivial setting, where pairwise independent nodes are dependent on other nodes. In the following, we demonstrate that if a certain substructure appears in the partial order, the lower bound approaches 3.

Theorem 10. *Consider a partial order $\mathcal{P}$ with subsets of nodes Q and R such that (1) nodes of Q are pairwise independent, and (2) the set of ancestors of each node of Q is R. Then, if a deterministic online algorithm* ALG *is c-competitive for online list access with the partial order $\mathcal{P}$, then $c \geq 3 - \frac{6r+6}{q+3r+2}$, for $q = |Q|, r = |R|$ and $r \geq 0, q \geq 0$.*

We defer the proof to the full version of the paper.

This lower bound generalizes the lower bound of $3-6/(n+2)$ given by Reingold et al. [26], where n is the length of the list. For $r = 0$ and $q = n$ the lower bounds match.

VI. Empirical Evaluation of Self-Adjusting Packet Classifier

We now empirically evaluate the benefits and limitations of our deterministic algorithm from § III in the practical use-case of self-adjusting packet classifier (see § I-A). Similar evaluations for classic list access algorithms, also concerning locality were performed by Bachrach et al. [5]. In our evaluations, the traffic locality refers to the skewness of the distribution of the rules (nodes of the list) hit with the packet match.

Our evaluation compares with packet classifiers beyond lists: we compare our algorithm MOVE-RECURSIVELY-FORWARD (MRF) to existing packet classifiers, including hierarchical cut classifiers [35, 29, 15]. We investigate the following question:

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[1] sizes, MRF improves classification time. Specifically, our results show that MRF's classification time is 7.01x better than Efficuts [35] and 3.64x better than CutSplit [20] for small ruleset sizes and high traffic locality.

The Figure 7a show the key findings of our evaluation, presenting the average nodes traversed (representing classification time) normalized to CutSplit [20]. We observe that MRF significantly improves the classification time compared to CutSplit in the region of smaller rulesets and high traffic locality. However, MRF's classification time degrades below the competition outside these regions.

A. Methodology

Rulesets and Packet traces: We use Classbench [33] 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 [17].

Comparison with existing packet classifiers: We compare MRF with list-based and decision tree-based packet classifiers. In our evaluations, a static-list serves as a baseline for a list-based approach. Among wide range of decision tree-based packet classifiers, our baselines include Hicuts [15], Hypercuts [29], Efficuts [35] and the more recent packet classifier CutSplit [20] 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. 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 Efficuts[2].

Metrics: We report the main metrics of interest: average classification time measured as the average number of traversed nodes. To measure the number of traversed nodes, we count the number of nodes accessed during lookup (access cost) and the number of swapped nodes during rearrangement (reconfiguration cost). For all our baselines, we only count the number of nodes accessed during lookup since they do not perform any rearrangements. For MRF, which additionally incurs the list reconfiguration cost due to self-adjustments, we 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).

[1]Ruleset is a set of rules to be maintained in a data structure which is then accessed for the purpose of packet classification in a network device. See Figure 2 for an example and § I-A for more details.

[2]The 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.

B. 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 5, 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 the average number of ancestors. All three metrics of the dependency graph structure influence the classification time.

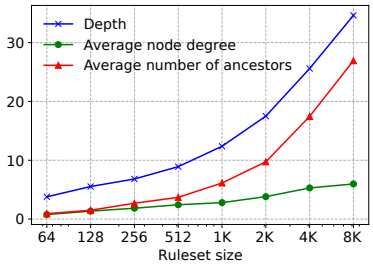


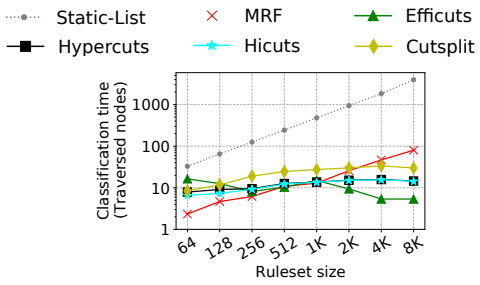
Fig. 5: Statistics concerning our synthetic dataset. Increase of maximum node depth, average node degree, and the average number of ancestors in the dependency graph increase with the ruleset size.

MRF under high traffic locality: In Figure 6a, 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 Efficuts, 3.64x better than CutSplit, and 14.06x better than a static-list. For ruleset size of 1K: MRF performs 1.15x better than Efficuts, 2.12x better than CutSplit, and 37.04x better than a static-list. For larger ruleset sizes ($> 1K$): MRF performs 15x worse compared to Efficuts and 2.7x worse compared to CutSplit, yet 49x better that a static-list.

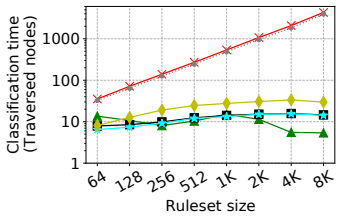
For a small ruleset size (e.g., 64), we can see from Figure 5 that the average number of ancestors is much lower. This allows MRF to move the frequently matched rules closer to the head of the list, which significantly improves classification time under high locality. For large ruleset sizes ($>$ 1K) the average node ancestors grow 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 6b, we observe that the classification time of MRF is comparable to a static list for all ruleset sizes.

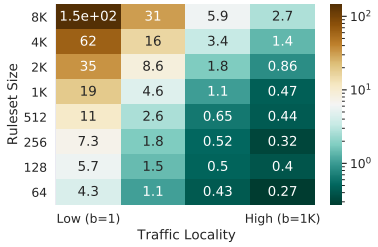


(a) Classification time under high-locality in traffic



(b) Classification time under low-locality in traffic

Fig. 6: 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). Note the log scale in the figures.



(a) $\frac{MRF}{CutSplit}$ normalized Avg. nodes traversed.

Fig. 7: MRF out-performs static decision tree-based packet classifier CutSplit for small ruleset sizes and at high traffic locality (towards the bottom right region of the ruleset size vs. locality dimensions).

VII. Related Work

Online problems with partial orders: Prior work considers various online problems with partial orders (often referred to as *dependencies* or *precedence constraints*). Some online problems directly concern partial orders, e.g., poset partitioning [8], and other classic online problems were extended with additional constraints with respect to a partial order given in addition. In scheduling with precedence constraints [4], a job can only be scheduled after all its predecessors are completed. In caching with dependencies [6], an element can be brought into the cache only if all its dependencies are present in the cache. (Similarly to our problem, caching with dependencies was also motivated by network packet classification.)

Online self-adjusting lists: Self-adjusting lists are traditionally considered in the context of online algorithms and competitive analysis [27, 30], where the problem is known under the name of *online list access* [7, 12]. This problem was studied under many models, and one of the most popular is the *paid exchange* P^d *model* [26], where each transposition costs $d \geq 1$. For $d = 1$, the deterministic algorithm Move-To-Front [30] can be shown to be 4-competitive, and for larger d, deterministic COUNTER algorithms achieve constant competitive ratios [12, Ch. 1], converging to the competitive ratio $(5 + \sqrt{17})/2 \approx 4.56$ as d grows. The lower bound of 3 for deterministic algorithms was given by Reingold et al. [26]. Randomized algorithm achieve better competitive ratios: Reingold et al. designed a family of RANDOM-RESET randomized algorithms [26], which includes a $\sqrt{7} \approx 2.64$-competitive algorithm against the oblivious adversary for $d = 1$, and converges to the competitive ratio $(5+\sqrt{17})/2 \approx 2.28$ as d grows. The analysis of RANDOM-RESET algorithms was later extended to the Markov family of algorithms [14], but no improved competitive ratios were provided. The best algorithm belongs to a TIMESTAMP family, with competitive ratio approaching 2.24 as d grows [3], given by Albers and Janke, who also designed the best known lower bound of 1.8654 against the oblivious adversary [3]. Recently, two new models were studied: Fotakis et al. introduced *online min-sum set cover* [13], a variant where a request can be matched by more than one item from the list; Olver et al. introduced *itinerant list update* [24], a variant where the pointer must not return to the front of the list after each request.

Packet classification: A wide range of data structures for packet classification were proposed in the literature: lists, tries, hash tables, bit vectors or decision trees [16, 32, 11], 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 static linear list packet classifier is commonly applied in practice, e.g., in the default firewall suite of the Linux operating system kernel `iptables` [22], the OpenFlow reference switch [25], and 5G packet detection rules (PDR) [1].

VIII. Conclusions

We introduced a model for online partially ordered list access, a well-motivated and natural extension of online list access. Despite the additional constraints, our algorithms match the competitiveness of their counterparts from classic list access. A question about the relation of the competitiveness of list access and partially ordered list access remains open.

Acknowledgements. This work was supported by the European Union's Horizon 2020 European Research Council (ERC) 2020-2025, grant agreement No. 864228 (AdjustNet), by the Austrian Science Fund (FWF) project I 5025-N (DELTA) and by the National Research, Development and Innovation Fund of Hungary OTKA/ANN-135606, OTKA/FK-135074 and OTKA/FK-134604 grants.

REFERENCES

[1] 3GPP. Interface between the Control Plane and the UserPlane Nodes; Stage 3. Technical Specification (TS) 29.244, 3rd Generation Partnership Project (3GPP), 2021. Version 17.1.0.

[2] Susanne Albers. Improved randomized on-line algorithms for the list update problem. *SIAM J. Comput.*, 27(3):682–693, 1998.

[3] Susanne Albers and Maximilian Janke. New bounds for randomized list update in the paid exchange model. In *Proc. of the International Symposium on Theoretical Aspects of Computer Science, STACS*, volume 154, pages 1–17. Schloss Dagstuhl, 2020.

[4] Yossi Azar and Leah Epstein. On-line scheduling with precedence constraints. *Discret. Appl. Math.*, 119(1-2):169–180, 2002.

[5] Ran Bachrach, Ran El-Yaniv, and M. Reinstadtler. On the competitive theory and practice of online list accessing algorithms. *Algorithmica*, 32(2):201–245, 2002.

[6] Marcin Bienkowski, Jan Marcinkowski, Maciej Pacut, Stefan Schmid, and Aleksandra Spyra. Online tree caching. In *Proc. of the ACM Symposium on Parallelism in Algorithms and Architectures, SPAA*, pages 329–338. ACM, 2017.

[7] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[8] Bartlomiej Bosek and Tomasz Krawczyk. On-line partitioning of width w posets into wo(loglogw) chains. *Eur. J. Comb.*, 91, 2021.

[9] Marek Chrobak and Lawrence L. Larmore. The server problem and on-line games. In *On-Line Algorithms, Proc. of a DIMACS Workshop*, volume 7 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 11–64. DIMACS/AMS, 1991.

[10] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge mathematical textbooks. Cambridge University Press, 2002.

[11] David Eppstein and Shan Muthukrishnan. Internet packet filter management and rectangle geometry. In *Proc. of the Twelfth Annual Symposium on Discrete Algorithms, SODA*, pages 827–835. ACM/SIAM, 2001.

[12] Amos Fiat and Gerhard J. Woeginger, editors. *Online Algorithms, The State of the Art*, volume 1442 of *Lecture Notes in Computer Science*. Springer, 1998.

[13] Dimitris Fotakis, Loukas Kavouras, Grigorios Koumoutsos, Stratis Skoulakis, and Manolis Vardas. The online min-sum set cover problem. In *47th International Colloquium on Automata, Languages, and Programming, ICALP*, volume 168, pages 51:1–51:16, 2020.

[14] Theodoulos Garefalakis. A new family of randomized algorithms for list accessing. In *European Symposium on Algorithms*, pages 200–216. Springer, 1997.

[15] Pankaj Gupta and Nick McKeown. Classifying packets with hierarchical intelligent cuttings. *IEEE Micro*, 20(1):34–41, 2000.

[16] Pankaj Gupta and Nick McKeown. Algorithms for packet classification. *IEEE Network*, 15(2):24–32, 2001.

[17] Jonathan R.M. Hosking and James R. Wallis. Parameter and quantile estimation for the generalized pareto distribution. *Technometrics*, 29(3):339–349, 1987.

[18] Shahin Kamali and Alejandro López-Ortiz. A survey of algorithms and models for list update. In *Space-Efficient Data Structures, Streams, and Algorithms*, volume 8066, pages 251–266. Springer, 2013.

[19] Kirill Kogan, Sergey I. Nikolenko, Ori Rottenstreich, William Culhane, and Patrick Eugster. SAX-PAC (scalable and expressive packet classification). In *ACM SIGCOMM*, pages 15–26. ACM, 2014.

[20] Wenjun Li, Xianfeng Li, Hui Li, and Gaogang Xie. Cutsplit: A decision-tree combining cutting and splitting for scalable packet classification. In *IEEE Conference on Computer Communications, INFOCOM*, pages 2645–2653, 2018.

[21] John McCabe. On serial files with relocatable records. *Operations Research*, 13(4):609–618, 1965.

[22] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Mauricio Vásquez Bernal, Yunsong Lu, and Jianwen Pi. Securing linux with a faster and scalable iptables. *ACM SIGCOMM Computer Communication Review, CCR*, 49(3):2–17, 2019.

[23] J. Ian Munro. On the competitiveness of linear search. In *Proc. of the European Symposium, ESA*, volume 1879, pages 338–345, 2000.

[24] Neil Olver, Kirk Pruhs, Kevin Schewior, René Sitters, and Leen Stougie. The itinerant list update problem. In *Approximation and Online Algorithms - International Workshop, WAOA*, volume 11312, pages 310–326, 2018.

[25] ONF. Openflow reference release. https://github.com/mininet/openflow, 2013.

[26] Nick Reingold, Jeffery R. Westbrook, and Daniel Dominic Sleator. Randomized competitive algorithms for the list update problem. *Algorithmica*, 11(1):15–32, 1994.

[27] Ronald Rivest. On self-organizing sequential search heuristics. *Commun. ACM*, 19(2):63–67, 1976.

[28] Nadi Sarrar, Steve Uhlig, Anja Feldmann, Rob Sherwood, and Xin Huang. Leveraging zipf's law for traffic offloading. *ACM SIGCOMM Computer Communication Review, CCR*, 42(1):16–22, 2012.

[29] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. Packet classification using multidimensional cutting. In *Proc. of the Conference on Applications, technologies, architectures, and protocols for computer communications, ACM SIGCOMM*, pages 213–224. ACM, 2003.

[30] Daniel Dominic Sleator and Robert Endre Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, 1985.

[31] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[32] Venkatachary Srinivasan, Subhash Suri, and George Varghese. Packet classification using tuple space search. In *Proc. of the ACM SIGCOMM*, pages 135–146. ACM, 1999.

[33] David E. Taylor and Jonathan S. Turner. Classbench: A packet classification benchmark. *IEEE/ACM Transactions on Networking*, 15(3):499–511, 2007.

[34] Boris Teia. A lower bound for randomized list update algorithms. *Inf. Process. Lett.*, 47(1):5–9, 1993.

[35] Balajee Vamanan, Gwendolyn Voskuilen, and T. N. Vijaykumar. Efficuts: optimizing packet classification for memory and throughput. In *Proc. of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, ACM SIGCOMM*, pages 207–218. ACM, 2010.