Self-Adjusting Partially Ordered Lists

Anonymous Author(s)

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, without maintaining frequency statistics or timestamps. Although the partial order limits the effectiveness of self-adjustments, the 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 the rearrangement cost 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.

Self-adjusting partially ordered lists enable the benefits of self-adjustments even in constrained scenarios. We investigate the applicability of our lists in the context of network packet classification.

CCS CONCEPTS

• **Theory of computation** → Online algorithms.

KEYWORDS

Online algorithms, competitive analysis, list access, partial order

ACM Reference Format:

Anonymous Author(s). 2022. Self-Adjusting Partially Ordered Lists. In *Proceedings of ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '22).* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 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 [35] and splay trees [36].

Self-adjusting lists are traditionally analyzed in the context of online algorithms, where the problem is referred to as *online list access problem* [10, Ch. 1 and 2], one of the most fundamental problems in online algorithms. The problem was studied for decades [2, 26, 31, 32, 35, 39] and remains an active field of research [3]. Over years, the community studied the problem under various cost models [22, 28] and generalizations [16, 29]. 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.

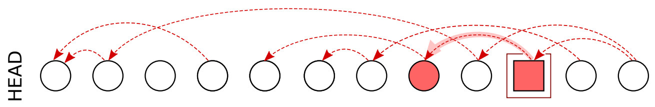


Figure 1: An example of a list configuration with a partial order among the nodes. The requested node, depicted as a square, cannot move forward beyond the solid node due to a precedence constraint (highlighted with a bold arrow). If we choose to move the requested node close to the head of the list, 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* [19], 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 [35] 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

SPAA '22, Philadelphia, PA. July 11-14, 2022,
2022. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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, 31] 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* [10] 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.

1.1 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 [11], and other classic online problems were extended with additional constraints with respect to a partial order given in addition. In scheduling with precedence constraints [7], a job can only be scheduled after all its predecessors are completed. In caching with dependencies [9], 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.)

Models for self-adjusting lists. Self-adjusting lists are traditionally considered in the context of online algorithms and competitive analysis [32, 35], where the problem is known under the name of *online list access* [10, 15]. The majority of the literature for the list access problem considers the *free exchange model* [35] where moving the accessed node to the front of the list is free, and any other transposition costs 1. In this model, Sleator and Tarjan showed that the algorithm Move-To-Front is 2-competitive [35], and Karp and Raghavan noted that this ratio is optimal for deterministic algorithms (reported in [31]). In the randomized setting, the best algorithm is 1.6-competitive COMB [6], and a lower bound of 1.5 [39] exists against the oblivious adversary. Another popular model is the *paid exchange P^d model* [31], where each transposition costs $d \geq 1$; we discuss this model in detail in the next paragraph. The above two models attracted the most attention, but some papers studied other rearrangement cost variants, e.g., with batch rearrangements with linear cost [25, 28], or with generalized costs where the access cost is any monotonic function of node's position [35]. Recently, two new models were studied: Fotakis et al. introduced *online min-sum set cover* [16], a variant where requests arrive in batches that can be served out-of-order; Olver et al. introduced *itinerant list update* [29], a variant where the pointer must not return to the front of the list after each request.

Online list access in the P^d model. First, we review deterministic algorithms. For $d = 1$, the deterministic algorithm Move-To-Front [35] can be shown to be 4-competitive, and for larger d, deterministic COUNTER algorithms achieve constant competitive ratios [15, 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. [31]. Second, we review randomized algorithms. Reingold et al. designed a family of RANDOM-RESET randomized algorithms [31], 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 [17], 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].

1.2 Contributions

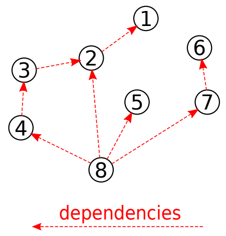
The main technical contribution of this paper is the design of constant-competitive deterministic and randomized algorithms for online partially ordered list access. The deterministic algorithm is simple, memoryless, and 4-competitive in $d = 1$ case (the case fitting the application of packet classification). The randomized algorithms successfully generalize a family of Markov algorithms from classic list access [17] (including Move-To-Front, BIT, COUNTER, RANDOM-RESET [31, 35]) 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. [31], 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* [19]. Our algorithms interpreted in this context give rise to *self-adjusting packet classifiers*, which adapt to the traffic they serve.

1.3 Practical Motivation: Self-Adjusting Packet Classification

Packet classification. In communication networks, packet classification [19] 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



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

Figure 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.

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 [14, 19, 37], as well as TCAM hardware solutions). We present initial experimental results in Appendix A 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 [33].

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 [35].

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 [23].

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, 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` [27], the OpenFlow reference switch [30] , and 5G packet detection rules (PDR) [1]. Our deterministic algorithm (introduced and analyzed in Section 3) is a drop-in replacement for static lists that comes with no memory penalty (no timestamps or frequency statistics) and no significant increase in complexity. The algorithm improves performance up to 14x times under high traffic locality for small rule sets (cf. Appendix A), with quick convergence to (even local) shifts in traffic, while operating without any a priori assumptions about traffic distribution.

1.4 Organization

The rest of this paper is organized as follows. In the next section, we formalize the model for online partially ordered list access. Next, we design and analyze online algorithms for this problem. We start with a deterministic algorithm in Section 3, where we introduce a recursive procedure, a building block of our algorithms, and state combinatorial facts about the procedure with respect to inversions. We design a family of randomized algorithms in Section 4. Then, in Section 5 we discuss the influence of particular partial orders on the competitiveness of online algorithms. We conclude in Section 6. We point to appendices that give additional details about our algorithms: in Appendix A, we perform an empirical case study to determine if our deterministic algorithm improves performance with locality of traffic. In Appendix B we discuss the best parameters for our randomized algorithms. Appendix C contains omitted proofs from the main body. In Appendix D, we discuss insertions and deletions. Appendix E overviews competitive analysis.

2 MODEL

We introduce the *online partially ordered list access* problem. In this problem, 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 given partial order. If the partial order is empty, the problem is equivalent to classic online list access [35].

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, traversing nodes until encountering the requested node. Accessing the node at position i in the list costs i (the first 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. For an overview of competitive analysis, we refer to Appendix E.

3 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 Section 4). The algorithm can be viewed as a generalization of Move-To-Front [35] 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* [12].

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 [13]. 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 $\text{pos}(z)$ 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). 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.

3.1 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.

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

```
Input: An access request to node $\sigma_t$
Access $\sigma_t$
Run the procedure MRF($\sigma_t$)
procedure MRF($y$):
    if $y$ has no ancestors in $\mathcal{P}$ then
        Move $y$ to the front of the list
    else
        Let $z$ be the blocking ancestor of $y$ in $\mathcal{P}$
        Move node $y$ to $\text{pos}(z) + 1$
        Run the procedure MRF($z$)
    end
```

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 for an example.

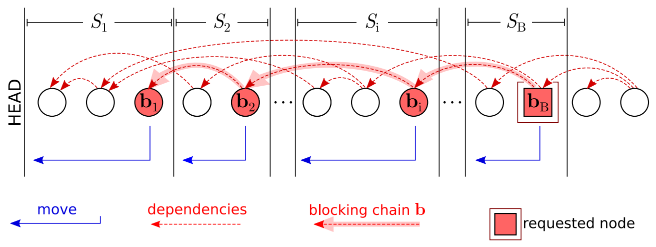


Figure 3: An example of handling a request by the algorithm MRF. The blocking chain is denoted by nodes $\mathbf{b}_i$, with the dependencies between them distinguished by bold arrows. Each node of the blocking chain moves right behind its blocking ancestor and this movement is depicted with a blue arrow 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 3.1). 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 $\text{pos}(y)$. For a graphical argument, we refer to Figure 3.

Lemma 3.1. *Consider a single request to a node y at position $\text{pos}(y)$ handled by the algorithm Move-Recursively-Forward. Then, the number of transpositions after serving the request is $\text{pos}(y) - B$, 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. Hence, the total number of transpositions is

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

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

3.2 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 [35] 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 3.2. *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.*

We defer the proof of this claim to Appendix C.

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 3.2 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.

3.3 How Do Rearrangements Affect Inversions?

In the analysis of Move-To-Front [35], 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.

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 3.3. Our argument requires careful analysis of the rearranged nodes, as many of them move after serving the request.

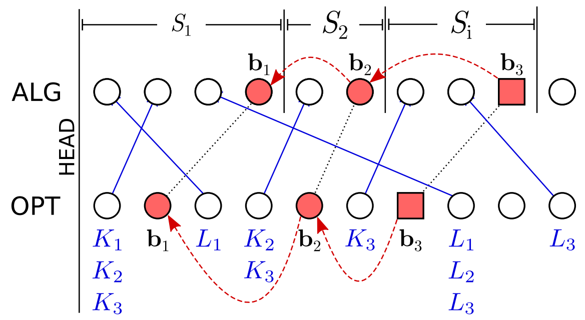


Figure 4: This example illustrates central definitions of sets of nodes used in our analysis. We depict the positions of nodes in both MRF's and OPT's list (joined by solid blue lines). The dotted black lines between the nodes $\mathbf{b}_i$ help in determining the assignment of nodes to sets: in K_i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

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 3.3. *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 due to MRF's node rearrangement after serving the request 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 3.6) and destroyed (Lemma 3.7). Before showing these claims, we inspect the basic relations between the sets K_j, L_j and S_j (Lemmas 3.4 and 3.5).

Lemma 3.4. *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 3.5. *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 3.4, equation 1 and the definition of k. Second, we prove the equality 2. We have the following chain of inequalities

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

where the first step holds as the sets S_j are disjoint, the second step follows by Lemma 3.4, equation 2, the inequality follows by basic properties of sets, and the last step follows by the definition of ℓ. □

Lemma 3.6. *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 3.5, equation 1. □

Lemma 3.7. *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 3.5, equation 2. □

Combining Lemmas 3.6 and 3.7 gives us the joint bound on the change in the number of inversions and proves the Theorem 3.3. We note that this bound is consistent with the bound on the changes in inversions for the algorithm Move-To-Front [35], where the inversions were considered with respect to the requested node only.

3.4 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 [35].

Theorem 3.8. *The algorithm MRF is strictly 4-competitive in the P^1 model.*

We defer the proof of this theorem to Appendix C. The proof repeats the arguments of Sleator and Tarjan for Move-To-Front, and internally we use a generalized argument concerning inversions (Theorem 3.3), which handles the generalized recursive rearrangement procedure.

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.

4 MARKOV FAMILY OF RANDOMIZED ALGORITHMS

We design a family of constant-competitive online algorithms that generalize Markov algorithms from classic list access [17] to the partially ordered list access (including Move-To-Front, BIT, COUNTER, RANDOM-RESET [31, 35]). 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 Section 3 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 Section 3) 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.

Markov chain. Let M be an irreducible Markov chain with a finite set of states $S_M = \{0, 1, ...s-1\}$, transition probabilities $P = (p_{i \to j})$ and has a stationary distribution $\pi = (\pi_0, \pi_1, ...\pi_{s-1})$ where $p_{i \to j}$ denotes the transition probability from state i to state j and π_i denotes the stationary probability of a state i. We denote by $h_{i \to j}$, the hitting time from state i to state j in M, where $i, j \in S_M$. Similar to [17], the hitting time to state 0 plays a crucial role in our analysis. For simplicity, we write h_i for the hitting time $h_{i \to 0}$. We denote by T the expected hitting time to state 0, given by $T = \sum_{i=0}^{s-1} \pi_i \cdot h_i$.

Algorithm MMRF definition. Let $y = \sigma_t$ be the requested node at time t. Before serving the access request, every node $x \in N$ in the list transitions to state j with probability $p_{\text{STATE}(x) \to j}$. The algorithm then executes the recursive procedure MMRF. Let $\text{pos}(z)$ denote the position of node z in the list maintained by the algorithm, starting from 1. MMRF(y) checks the state of y, and if it is 0, then it moves y forward (via transpositions) until it encounters the blocking ancestor node z. The procedure recursively calls MMRF(z) if $\text{pos}(z) \neq 0$. We present the pseudocode of MMRF in Algorithm 2.

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 $\pi$.
Input: An access request to node $\sigma_t$

1  Access $\sigma_t$
2  for each node $x$ in the list do
3      Advance the Markov chain of $x$
4  end
5  Run the procedure MMRF($\sigma_t$)
6  procedure MMRF($y$):
7      if $y$ has no ancestors in $\mathcal{P}$ then
8          if STATE($y$) is 0 then
9              Move $y$ to the front of the list
10         end
11     else
12         Let $z$ be the blocking ancestor of $y$ in $\mathcal{P}$
13         if STATE($y$) is 0 then
14             Move node $y$ to pos($z$) + 1
15         end
16         Run the procedure MMRF($z$)
17     end
```

4.1 Hidden Inversions

We demonstrate the challenges in analyzing randomized algorithms for online partially ordered list access with an example related to the algorithm BIT [31], a 2.75-competitive randomized for classic list access in the P^1 model. In a nutshell, BIT maintains a bit for each node, initially set uniformly at random. Upon access to a node, the algorithm flips its bit, and it moves the node to the front only if the bit was turned to 0. The algorithm was analyzed using a potential function that distinguishes between types of inversions based on the bit of the back node, and weights these inversions differently. The crucial observation for the competitive analysis of BIT is that if the requested node does not move, the potential decreases due to the change of its bit, and the amortized cost for serving a request is low.

Consider a BIT-like randomized algorithm for online partially ordered list access that employs the recursive procedure MRF to handle partially ordered lists. Then, the requested node changes its bit, but it does not destroy all inversions: instead of moving all the way to the front, it moves forward only until encountering its blocking ancestor (only moves forward in front of nodes from S_B). The change in the accessed node's bit increases the potential of all inversions that were not destroyed (the nodes from $S_1, S_2, S_3, \ldots, S_{B-1}$). The increase in potential causes the amortized cost to increase, and this analysis approach is no better than 3-competitive.

To limit the effect of nodes flipping their bits (but not destroying all inversions), we introduce *hidden inversions*. Consider a pair of nodes (u, v) that OPT keeps in the reversed order (v, u). If in the online algorithm's list the blocking ancestor of v is in between u and v, then this inversion is hidden. Each hidden inversion adds a constant value to the potential function (independently of the bit of v). An inversion that is not hidden is called *visible*.

Definition 4.1 (Hidden regions H). For every node v in the list, we define a hidden region denoted by H_v as the set of nodes in front of the blocking ancestor of v in ALG's list.

Definition 4.2 (Hidden and visible inversions). An inversion (u, v) is hidden if u is in H_v, the hidden region of v. An inversion (u, v) is visible if u's position in the list is after the v's blocking ancestor and before v i.e., u is outside the hidden region of v. Visible inversions are further classified as type-i where i is the state of v.

When our algorithm moves a node, only the currently visible inversions are destroyed. Consider the example in Figure 5. Hidden inversions with respect to the node $\mathbf{b}_2$ are not destroyed; since $\mathbf{b}_2$ can only move forward to a position behind its blocking ancestor $\mathbf{b}_1$. The movement of $\mathbf{b}_2$ can however destroy all visible inversions, i.e., the inversions between $\mathbf{b}_2$ and $\mathbf{b}_1$.

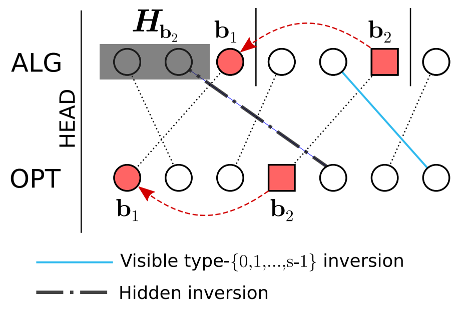


Figure 5: Consider the node b_2 and its blocking ancestor b_1. The region from the head of the list until b_1 is the *hidden* region H_{b_2} with respect to b_2. Any inversion of the form (u, b_2) is classified as *(i) hidden inversion* if u lies in the hidden region of b_2, otherwise *(ii) visible inversion* if u lies between b_1 and b_2. For intuition, notice that the movement of b_2 can only destroy visible inversions and cannot destroy any hidden inversions. This is due to the precedence constraints i.e., b_2 cannot be moved ahead of its blocking ancestor b_1.

4.2 Preliminaries to the Analysis of MMRF

First, we discuss the potential function used to relate the cost ALG and OPT, designed to account for hidden inversions, defined in Section 4.1. Second, we claim that the state of each node in ALG's list remains independent of other nodes in the list and OPT's configuration throughout the execution of the algorithm. Third, we bound the amortized cost of the algorithm on a single request; to this end, we inspect individual executions of the recursive procedure MMRF. Finally, we bound the competitive ratio of the algorithm.

Potential function. We compare the costs of ALG and an optimal offline algorithm OPT on σ using the potential function Φ defined as

$$\Phi = \sum_{i=0}^{s-1} (d + h_i) \cdot \Phi_i + (d + T) \cdot \Phi_h,$$

where Φ_i is the number of inversions of type-i (visible inversions) and i ranges from 0 to $s - 1$, corresponding to each state in M; Φ_h is the number of hidden inversions. Recall that T is the expected hitting time to state 0 and is given by $T = \sum_{i=0}^{s-1} \pi_i \cdot h_i$. Recall that d is the cost of one transposition of neighboring nodes in the list.

State independence. Our analysis uses an observation that the state of nodes in MMRF's list are initialized according to the stationary distribution and remain independent of each other at the stationary distribution as the states change over time.

Lemma 4.3. *The state of any node y in ALG's list is i with probability π_i at any time ($0 \le i < s$), independent of its position in OPT's list and other nodes' states.*

Proof. The initial states are chosen uniformly at random from s^n possible choices. Every node x in the list transitions to state j with probability $p_{\text{STATE}(x)\to j}$ upon every access request (to any node). As a result, after serving m requests, every node x in the list advances m times in its Markov chain. The state of a node x after any number of requests is j with probability π_j (the stationary distribution of the Markov chain). Thus, the state of each node depends only on its initial value and the number of requests observed so far. □

Lemma 4.4. *Consider a request to a node σ_j. Then the expected change in potential of any node that is not in the blocking chain of σ_j is zero.*

Proof. Fix any node x that is not in a blocking chain. By Lemma 4.3, the state of x in the list is j with probability π_j. As the position of x does not change, the expected change in potential by changing its state is due to the inversions that change type. Let ℓ be the number of inversions w.r.t the node x and j be the state of x before advancing its Markov chain. Then the expected change in potential due to inversions changing type is given by

$$E[\Delta\Phi] = \sum_{k=0}^{s-1} \ell \cdot p_{j\to k} \cdot ((d + h_k) - (d + h_j))$$
$$= \ell \cdot (\sum_{k=0}^{s-1} p_{j\to k} \cdot h_k - \sum_{k=0}^{s-1} p_{j\to k} \cdot h_j) = \ell \cdot (h_j - h_j) = 0.$$ □

4.3 How Do Rearrangements Affect Hidden Inversions?

The crucial part of our analysis is the change in the potential due to changes in hidden inversions. The result of this section is that it suffices to *only* consider visible inversions in the amortized cost analysis, contrary to considering *all* inversions. There are two cases for changes in hidden inversions: movement of a node in ALG's list or movement of a node in OPT's list.

Movement of a node in ALG's list. Fix a single reconfiguration of the algorithm while serving the request σ_t, and consider a single node $\mathbf{b}_j$ that moves forward in a call of procedure MMRF. The movement of $\mathbf{b}_j$ may cause any of the following changes:

- A hidden inversion may become visible (moves outside the hidden set), and becomes type-i inversion.
- A visible type-i inversion may become hidden inversion.
- A new hidden inversion may be created.

Movement of a node in OPT's list. For each transposition OPT pays d and may create a new inversion. The new inversion is either hidden or visible type-i.

We claim that the change in potential due to inversions changing type from and to hidden inversions is zero.

Lemma 4.5. *The expected change in the potential due to inversion type changes from and to hidden inversions is zero.*

We defer the proof to Appendix C and provide a sketch of proof here. We first prove that the expected change in potential due to any inversions that change their type from hidden to a visible type is zero. We then prove the vice versa i.e., the expected change in potential due to inversions which change from any visible type to hidden is zero. Notice that the expected change in potential due to an inversion changing from visible type i to a hidden inversion is $\sum_{i=0}^{s-1} \pi_i \cdot ((d+T) - (d+h_i)) = 0$. Similarly, we analyze the change in potential due to an inversion changing its type from hidden to a visible type i and the proof follows.

We now claim that the change in potential due to a created hidden inversion equals the expected change in potential if the created inversion is a visible type-i inversion.

Theorem 4.6. *The expected change in potential $E[\Delta\Phi]$ due to a created hidden inversion is $d+T$ and equals the expected change in potential due to a created visible inversion.*

Proof. For each created hidden inversion, the change in potential is $\Delta\Phi = d+T$. If the created inversion is a visible inversion, the inversion is of type-i with probability π_i. The expected change in potential is then $E[\Delta\Phi] = \sum_{i=0}^{s-1} \pi_i \cdot (d+h_i) = d+T$. The last equality holds since $T = \sum_{i=0}^{s-1} \pi_i \cdot h_i$. □

4.4 Bounding the Competitive Ratio

We claim the following bound on the competitiveness of our algorithms:

Theorem 4.7. *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 $T = \sum_{i=0}^{s-1} \pi_i \cdot h_i$.*

To prove this claim, we first investigate the amortized cost of a single request handled by ALG, denoted $a(t)$, including the access cost, rearrangement cost and the change in potential.

First, we analyze the expected amortized reconfiguration cost due to the movement of a single node $\mathbf{b}_j$.

Lemma 4.8. *Consider a request σ_t served by ALG, and consider a single run of the procedure MMRF for some node $\mathbf{b}_j$ during the recursive call of MMRF procedure. The expected cost of transpositions that $\mathbf{b}_j$ participated in, and the potential change due to these transpositions is given by*

$$E[C_{re}^{j}(t) + \Delta\Phi^{j}] \le |K_j \cap S_j| \cdot (2d+T)\cdot\pi_0 - |L_j \cap S_j| \cdot h_0 \cdot \pi_0.$$

Second, we analyze the amortized cost of ALG for a single access request event i.e., access cost plus the total amortized reconfiguration cost over the recursive calls of MMRF procedure.

Lemma 4.9. *The amortized cost of serving a request σ_t by ALG is $(k+1)\cdot(1+\pi_0\cdot(2d+T))$.*

The proof uses properties of hidden inversions and analysis of the amortized cost of request is straightforward. We defer the proofs to Appendix C.

With the bounds of the amortized cost of a request, we can prove the Theorem 4.7; we sketch the proof below, a full version appears in Appendix C. Consider any sequence of access requests σ. In order to prove our claim, it suffices to show that the expected amortized cost of MMRF in serving a request at any time t is $E[a(t)] \le C \cdot C_{\text{OPT}}$, where $C = \max\{1+\pi_0\cdot(2d+T), 1+\frac{T}{d}\}$. We distinguish between the following types of events that occur throughout the algorithm's execution:

Event-1: An *access request event* where both ALG and OPT serve the request. This event includes any paid transpositions made by ALG. We assume a fixed configuration of OPT throughout this event. We analyze each node $\mathbf{b}_j$ separately and obtain the expected amortized cost due to the movement of node $\mathbf{b}_j$, and then we sum over all nodes $\mathbf{b}_j$ and add the access cost of $k+l+1$ to obtain the total amortized cost of the request. In each step, we analyze all the changes in inversions i.e., created, destroyed, and inversion type changes, and obtain

$$E[a(t)] \le k\cdot(1+\pi_0\cdot(2d+T)) + 1 \le C_{\text{OPT}}\cdot(1+\pi_0\cdot(2d+T)),$$

where the last inequality holds as OPT pays at least $k+1$ for serving the access request at time t.
Event-2: A *paid exchange event* of OPT, 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 ALG throughout this event. OPT pays d for any paid transposition and at most one new inversion is created. The created inversion is either a hidden or visible. From Theorem 4.6, irrespective of whether the created inversion is hidden or visible, the expected change in potential is $d+T = C_{OPT}\cdot(1+\frac{T}{d})$.

Analyzing the above two events concludes the proof sketch.

Which Markov chains are best? The Theorem 4.7 concerns arbitrary Markov chains, and to obtain concrete competitive ratios, we need to decide which Markov chain to select. For a discussion on the best choice for a particular transposition cost d, we refer to Appendix B.

Note on the running time. Our randomized algorithms incur a substantial computational cost $O(n)$ for advancing the Markov chains of all nodes upon each access. We point out that our algorithms generalize a COUNTER family of algorithms [31], and this subset of Markov algorithms can be efficiently implemented. The idea is to maintain a global counter, incremented on every access, and leave the counters of each node read-only after setting them uniformly at random. To decide if a node of a blocking chain should move forward, we sum the global counter with the counter of the node. This modification decreases the cost of maintaining the random counters to a constant per request.

5 INFLUENCE OF PARTIAL ORDERS ON COMPETITIVENESS

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 [31].

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 [31], 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 5.1. *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 $r \geq 0, q \geq 0$.*

Proof. Fix any deterministic algorithm ALG and consider an input sequence σ of length m, consisting of the requests to the currently last node from Q in ALG's list. This sequence incurs the cost at least $m \cdot (r+q)$ access cost in total to ALG, since in the given partial order the nodes of R must be in front of the nodes of Q. On top of the access cost, assume that ALG pays ρ for reconfiguration cost, hence $C_{\text{ALG}}(\sigma) \geq m \cdot (r+q) + \rho$. Depending on the reconfiguration cost ρ incurred by ALG, the adversary chooses one of the following offline algorithms to serve the sequence.

If $\rho > m \cdot (r+q)/2$, then the adversary chooses a *static strategy*: the offline algorithm OFF_S moves the sets Q and R to the front, and chooses a static list among all $q!$ list with the lowest overall access cost. For any node, there are $(q-1)!$ lists that place the node at position $i \in [r+1, r+q]$, therefore for any request, the sum of access costs for all lists is

$$\sum_{i=r+1}^{r+q} i \cdot (q-1)! = \sum_{i=1}^{q} (i+r) \cdot (q-1)! = \frac{(q+1)!}{2} + r \cdot q!.$$

Averaging over all $q!$ lists implies that a static list exists with access cost at most $\frac{(q+1)}{2} + r$ per request, and OFF_S moves to this list configuration. In addition to the access cost, the offline algorithm incurs the reconfiguration cost at most $d \cdot (n-q-r) \cdot (q+r)$ for moving the sets Q and R to the front (recall that n is the length of the list). The initial reconfiguration cost of the offline algorithm is however bounded, hence by considering long enough sequence σ, the competitive ratio tends to

$$\frac{C_{\text{ALG}}(\sigma)}{C_{\text{OFF}_S}(\sigma)} \geq \frac{m \cdot (r+q) + \rho}{m \cdot (\frac{q+1}{2} + r)} > \frac{(r+q) + (r+q)/2}{\frac{q+1}{2} + r} = \frac{3r+3q}{q+2r+1},$$

where the inequality follows by the case assumption $\rho > m \cdot \frac{(r+q)}{2}$.

If $\rho \leq m \cdot (r+q)/2$, then the adversary chooses a *dynamic strategy*: initially, an offline algorithm OFF_D brings the set R to the front and the set Q right behind R, and keeps the list Q in the reversed order of ALG. The transpositions to maintain the reversed order are feasible possible since the nodes from Q are pairwise independent, and since the set R of all ancestors of nodes from Q is already in the front. The offline algorithm maintains the reversed order: for any transposition of ALG concerning two nodes from Q, the offline algorithm performs the reverse transposition; such transposition is possible without moving other nodes since the set R is already in the front. Hence, for the rearrangements the offline algorithm incurs the cost $\frac{d \cdot q \cdot (q-1)}{2}$ for the initial list reversal and $d \cdot (n-q-r) \cdot (q+r)$ for bringing sets Q and R to the front and at most ρ for reconfiguration. Furthermore, the offline algorithm pays the cost $m \cdot (r+1)$ for access, since the requested node is always the first node among the set Q, and the set Q is only preceded by the set R. The total cost of the offline algorithm is at most $\rho + m \cdot (r+1) + d \cdot (n-q-r) \cdot (q+r) + \frac{d \cdot q \cdot (q-1)}{2}$. The cost of initial rearrangements of the offline algorithm is bounded, hence by considering long enough sequence σ, the competitive ratio tends to

$$\frac{C_{\text{ALG}}(\sigma)}{C_{\text{OFF}_D}(\sigma)} \geq \frac{m \cdot (r+q) + \rho}{m \cdot (r+1) + \rho} \geq \frac{(r+q) + (r+q)/2}{(r+1) + (r+q)/2}$$
$$= \frac{\frac{3}{2} \cdot (r+q)}{\frac{1}{2} \cdot (r+q) + r + 1} = \frac{3r+3q}{q+3r+2},$$

where the second inequality holds by the case assumption $\rho \leq m \cdot (r+q)/2$.

Finally, we combine the bounds from both cases. The ratio obtained for the static strategy is no smaller than for the dynamic strategy: we have $\frac{3r+3q}{q+2r+1} \geq \frac{3r+3q}{q+3r+2}$, for $q \geq 0, r \geq 0$. The optimal offline algorithm incurs the cost no larger than OFF_S and OFF_D, hence the competitive ratio of ALG is least $\frac{3r+3q}{q+3r+2} = 3 - \frac{6r+6}{q+3r+2}$, which concludes our claim. □

Our lower bound generalizes the lower bound of $3 - 6/(n+2)$ given by Reingold et al. [31], where n is the length of the list. For $r = 0$ and $q = n$ the lower bounds match.

6 CONCLUSIONS AND FUTURE DIRECTIONS

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 raises about the relation of the competitiveness of list access and partially ordered list access. Currently, the algorithms for classic list access are more competitive. There are two major families of algorithms for classic list access problem: Markov [17, 31] and TIMESTAMP [2, 3]. The currently best is TIMESTAMP (by 0.04 for large d), shown in a recent paper [3]. In this work, we generalized the Markov family (including Move-To-Front, BIT, COUNTER, and RANDOM-RESET) to partially ordered list access; generalizing TIMESTAMP would equate to the model's known competitive ratios. However, Transferring the TIMESTAMP family to partially ordered list access is challenging. The analysis technique of *list factoring*, used to analyze TIMESTAMP, is not straightforward to generalize to the setting with precedence constraints. List factoring may open avenues for parameterized analysis of algorithms in terms of *locality of reference* [4] (a phenomenon confirmed empirically in Appendix A).

A fundamental question of how various partial orders influence competitiveness was merely scratched in this work. In this work, we demonstrated a substructure of partial orders that allow us to construct a lower bound approaching 3 against deterministic algorithms. A systematic study of which constraints make the problem easier, and which pose a challenge is left to future work.

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] Susanne Albers and Sonja Lauer. On list update with locality of reference. *J. Comput. Syst. Sci.*, 82(5):627–653, 2016.

[5] Susanne Albers and Michael Mitzenmacher. Revisiting the counter algorithms for list update. *Information processing letters*, 64(3):155–160, 1997.

[6] Susanne Albers, Bernhard von Stengel, and Ralph Werchner. A combined BIT and TIMESTAMP algorithm for the list update problem. *Inf. Process. Lett.*, 56(3):135–139, 1995.

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

[8] 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.

[9] 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.

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

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

[12] 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.

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

[14] 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.

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

[16] 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.

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

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

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

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

[21] Mark Kac. On the notion of recurrence in discrete stochastic processes. *Bulletin of the American Mathematical Society*, 53(10):1002–1010, 1947.

[22] 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.

[23] 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.

[24] 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.

[25] Conrado Martinez and Salvador Roura. On the competitiveness of the move-to-front rule. *Theor. Comput. Sci.*, 242(1-2):313–325, 2000.

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

[27] 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.

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

[29] 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.

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

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

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

[33] 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.

[34] 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.

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

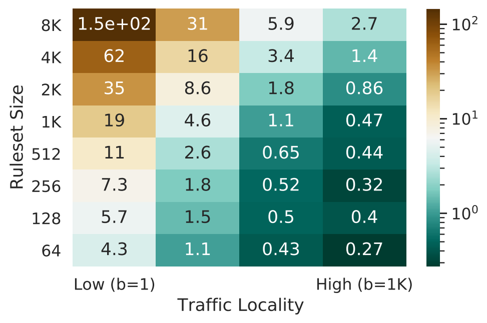
[36] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

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

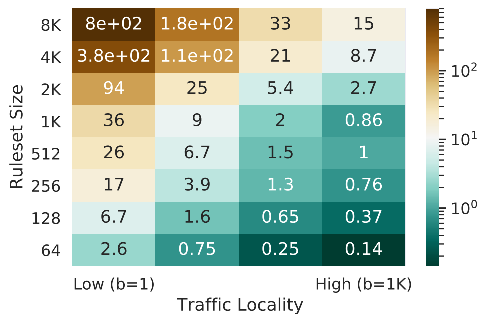
[38] David E. Taylor and Jonathan S. Turner. Classbench: A packet classification benchmark. *IEEE/ACM Transactions on Networking*, 15(3):499–511, 2007.

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

[40] 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.



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



(b) $\frac{MRF}{Efficuts}$ **normalized Avg. nodes traversed.**

Figure 6: 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).

A EMPIRICAL EVALUATION OF SELF-ADJUSTING PACKET CLASSIFIER

We now empirically evaluate the benefits and limitations of our deterministic algorithm from Section 3 in the practical use-case of self-adjusting packet classifier (see Section 1.3). Similar evaluations for classic list access algorithms, also concerning locality were performed by Bachrach et al. [8]. 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 aims to compare to packet classifiers beyond lists: we compare our algorithm Move-Recursively-Forward (MRF) to existing packet classifiers, including hierarchical cut classifiers [18, 34, 40]. 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[1] sizes, MRF improves classification time. Specifically, our results show that MRF's classification time is 7.01x better than Efficuts [40] and 3.64x better than CutSplit [24] 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.

Figures 6a and 6b show the key findings of our evaluation, presenting the average nodes traversed (representing classification time) normalized to CutSplit [24] and Efficuts [40]. We observe that MRF significantly improves the classification time compared to 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.

[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 Section 1.3 for more details.

A.1 Methodology

Rulesets and Packet traces: We use Classbench [38] 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 [20].

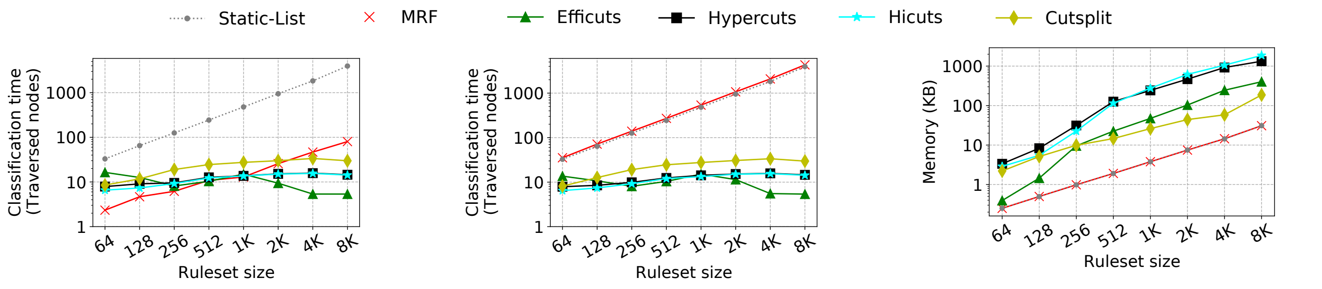
Comparison with existing packet classifiers: We compare MRF with list-based and decision tree-based packet classifiers. In our evaluations, a static-list serves as a baseline for a list-based approach. Among wide range of decision tree-based packet classifiers, our baselines include Hicuts [18], Hypercuts [34], Efficuts [40] and the more recent packet classifier CutSplit [24] 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]. 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. As a contribution to the research community and to facilitate future work, we will make our simulator and results publicly available as open-source code together with the final version of this paper.

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

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

[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.



(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**

Figure 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). Note the log scale in the figures.

rearrangement: the cost of a packet match is comparable to the cost of operations that accompany the swap (checking a rule overlap).

A.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 8, 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.

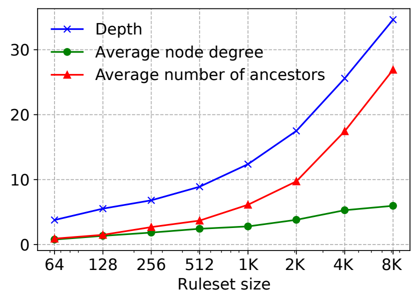


Figure 8: Increase of maximum node depth, average node degree, and the average number of ancestors in the dependency graph increase with the ruleset size.

MRF under high traffic locality. In Figure 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 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.

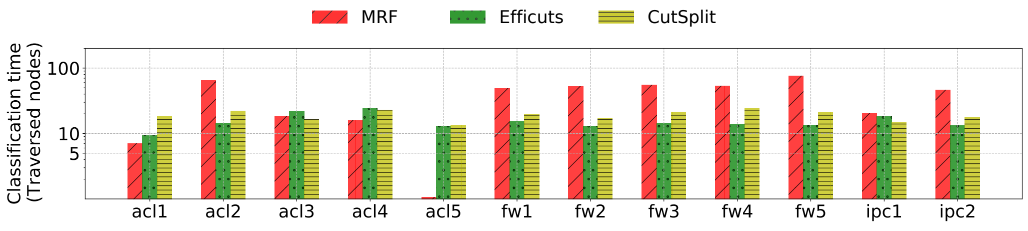
For a small ruleset size (e.g., 64), we can see from Figure 8 that the average number of ancestors is much lower. This allows MRF to move the frequently matched rules closer to the head of the list, which significantly improves classification time under high locality. For large ruleset sizes (> 1K) the 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 7b, we observe that the classification time of MRF is comparable to a static list for all ruleset sizes.

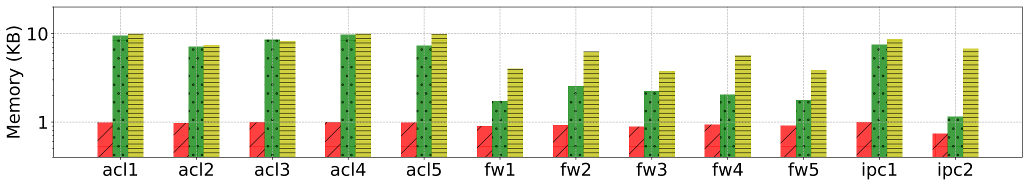
MRF memory consumption. MRF significantly reduces memory requirements for packet classification. The memory requirement of MRF reaches the lower bound for uncompressed data structures. In Figure 7c we show the memory requirements of different packet classifiers across various ruleset sizes generated 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 9b 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 9a.

B WHICH MARKOV CHAINS ARE BEST? LIFTING CLASSIC ALGORITHMS

We consider the COUNTER and RANDOM-RESET classes of algorithms, originally proposed by Reingold et al. [31] for the classic list access problem. Both COUNTER and RANDOM-RESET can be represented as a Markov chain algorithm by a transformation given by Garefalakis [17] that we recall below. Given a Markov chain M that corresponds to COUNTER and RANDOM-RESET class, with s states, Table 1 summarizes the competitive ratio of MMRF algorithms for partially ordered list access. The competitive ratios in our model match the ratios in the classic model. Notably,



(a) Classification time under high-locality in traffic and ruleset size 256.



(b) Memory usage for ruleset size 256.

Figure 9: 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.

RANDOM-RESET remains the best even for $d = 1$ and has a ratio of 2.64.

The two classic algorithms for online list access that we discuss were first proposed by Reingold et al. [31], and extended to many other randomized algorithms [3, 5].

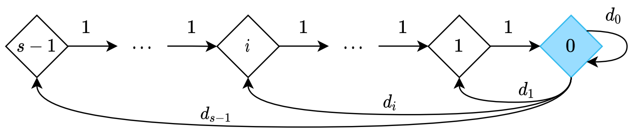


Figure 10: An example of the Markov chain representation of RANDOM-RESET(s, d). The diamonds represent states of the Markov chain, and the arrows are transitions between two states, indexed with the transition probability. The state 0 that initiates the movement in the list is shown in blue.

COUNTER for partially ordered list access. In a COUNTER(s) algorithm, each node has a *counter* internalized with an integer value chosen uniformly at random from the range $[0, s)$. The algorithm decreases the counter of each node after it was chosen, moves the node if its counter was 0, and resets the counter back to $s - 1$. As an example of algorithms in the COUNTER family, consider the extension of the known BIT algorithm, which has two states which are flipped after every access, and movement only happens if states are equal to 0. BIT algorithm can be expressed as COUNTER(2). Using the using transition probability below, we can express the COUNTER algorithm in terms of MMRF:

$$\forall\ 0 < i < s,\ p_{i\to(i-1)} = 1, \quad p_{0\to s-1} = 1.$$

RANDOM-RESET for partially ordered list access. In the RANDOM-RESET(s, P) algorithm, we assign counters to nodes similar to RANDOM-RESET. During the execution of MMRF on a node, the counter reduces by one (modulo s), and the node moves if the counters equal to 0, then goes to another state chosen based on the probability distribution P. Formally, RANDOM-RESET(s, P) is MMRF algorithm with the following transition probability (depicted in Figure 10):

$$\forall\ 0 < i < s,\ p_{i\to(i-1)} = 1, \quad \forall\ 0 \le i < s, p_{0\to i} = d_i.$$

	MMRF-COUNTER		MMRF-RANDOM-RESET	
d	best s	competitive ratio	best s	competitive ratio
1	2	2.75	3	2.64
2	5	2.50	5	2.45
3	7	2.43	8	2.39
4	10	2.38	10	2.36
5	12	2.38	13	2.34
6	15	2.33	15	2.33

Table 1: Competitive ratio of special cases of Markov algorithms, with increasing value of d in the P^d model.

The optimal number of states. Table 1 summarizes optimal competitive ratios for the two mentioned algorithms, showing the number of states required to achieve the optimal ratio in each case. This table has equivalent values as derived by Reingold et al. [31], which shows the same ratios can be achieved even for partially ordered list access.

C OMITTED PROOFS

C.1 Deterministic Algorithm

LEMMA 3.2. *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. □

Theorem 3.8. *The algorithm MRF is strictly 4-competitive in the P^1 model.*

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. Section 3.2), 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 $\text{pos}(\sigma_t)$, and by Lemma 3.1 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 request served by MRF. The amortized cost is $C_{\text{MRF}}(t) + \Delta\Phi(t)$ for each time t. By Theorem 3.3, 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 $\text{pos}(\sigma_t) = k + \ell + 1$, 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 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 $C_{\text{MRF}}(\sigma) \leq 4 \cdot C_{\text{OPT}}(\sigma)$. □

C.2 Hidden Inversions

Lemma C.1. *Consider a node $\mathbf{b}_j$ that moves forward after access. The expected change in potential due to $\mathbf{b}_j$'s movement for inversion type changes from hidden to visible type is zero.*

Proof. Consider any inversion (u, v) that changes type from hidden to any visible type-i. Let $(u, \mathbf{b}_j, v)$ be the order of nodes in the list, where $\mathbf{b}_j$ is the blocking ancestor of v. Note that the presence of $\mathbf{b}_j$ between u and v is required such that u lies in the hidden region of v. For such an inversion to change from hidden to visible, $\mathbf{b}_j$ must move ahead of u, thereby leaving u in the visible region of v. The visible inversion type is then based on the state of v. This change in inversion type is caused by the movement of $\mathbf{b}_j$ and is related to neither u nor v. By Observation 4.3, the probability that the state of v is i is given by the stationary probability π_i, and consequently, the probability that the inversion is of visible type-i is π_i. Thus, the potential value for this inversion changes from $d + T$ to $d + h_i$, with probability π_i. Hence the expected change in the potential is zero i.e., $\sum_{i=0}^{s-1} \pi_i \cdot ((d + h_i) - (d + T)) = 0$, since $T = \sum_{i=0}^{s-1} \pi_i \cdot h_i$ and $\sum_{i=0}^{s-1} \pi_i = 1$. □

Lemma C.2. *When the algorithm moves a node $\mathbf{b}_j$ forward in the list, the expected change in potential due to inversion type changes from any visible type to hidden is zero.*

Proof. Consider any inversion $(\mathbf{b}_j, v)$ that changes type from a visible type to hidden. Let u be the blocking ancestor of v such that $(u, \mathbf{b}_j, v)$ is the order of nodes in the list. Note that the presence of $\mathbf{b}_j$ between u and v is required such that $\mathbf{b}_j$ lies in the visible region of v. For such an inversion to change from visible to hidden, $\mathbf{b}_j$ must move ahead of u so that $\mathbf{b}_j$ lies in the hidden region of v. The original visible inversion type is based on the state of v, which is independent of the state of $\mathbf{b}_j$ and u. By Observation 4.3, the probability that the state of v is i is given by the stationary probability π_i, and consequently, the probability that the visible inversion is of type i is π_i. Thus, the potential value for this inversion changes from $d + h_i$ (with probability π_i) to $d + T$. Hence the expected change in the potential is zero, i.e., $\sum_{i=0}^{s-1} \pi_i \cdot ((d + T) - (d + h_i)) = 0$, since $T = \sum_{i=0}^{s-1} \pi_i \cdot h_i$ and $\sum_{i=0}^{s-1} \pi_i = 1$. □

Lemma 4.5. *The expected change in the potential due to inversion type changes from and to hidden inversions is zero.*

Proof. The claim follows from Lemma C.1 and Lemma C.2, by summing over all inversions that change their type to hidden and all inversions that change their type from hidden. □

C.3 Amortized Cost of a Request

Let $a(t)$ be the amortized cost of ALG in serving a request t and the cost incurred by OPT be $C_{\text{OPT}}(t)$. We split the amortized cost of ALG as $a(t) = C_{acc}(t) + C_{re}(t) + \Delta\Phi$, where $\Delta\Phi = A + B + F + H$ is the change in potential, $C_{acc}(t)$ is the access cost, $C_{re}(t)$ is the cost for paid transpositions i.e., reconfiguration cost, A is the change in potential due to created inversions, B is the change in potential due to destroyed inversions, F is the change in potential due to inversions that change type and H is the inversions that change type to and from hidden inversions.

Let $\mathbf{b}_\delta$ be the requested node. Since the movement of the node $\mathbf{b}_j$ (see Algorithm 2) is independent for all $1 \le j \le \delta$, we represent the amortized cost $a(t)$ as shown in Equation 1, where the superscript j indicates the changes in potential due the movement of $\mathbf{b}_j$. This accounts for the access cost, paid transpositions and all the changes in potential.

$$a(t) = C_{acc}(t) + C_{re}(t) + \Delta\Phi = \overbrace{C_{acc}(t)}^{\text{access cost}} + \overbrace{\sum_{j=1}^{\delta} C^j_{re}(t) + \Delta\Phi^j}^{\text{reconfiguration cost}}. \tag{1}$$

We analyze the expected amortized reconfiguration cost due the movement of a single relay node $\mathbf{b}_j$.

Lemma 4.8. *Consider a request σ_t served by ALG, and consider a single run of the procedure MMRF for some node $\mathbf{b}_j$ during the recursive call of MMRF procedure. The expected cost of transpositions that $\mathbf{b}_j$ participated in, and the potential change due to these transpositions is given by*

$$E[C^j_{re}(t) + \Delta\Phi^j] \le |K_j \cap S_j| \cdot (2d + T) \cdot \pi_0 - |L_j \cap S_j| \cdot h_0 \cdot \pi_0.$$

Proof. The proof repeats the arguments of Garefalakis [17], but additionally we must account for hidden inversions. We split the change in potential due to the movement of the node $\mathbf{b}_j$ into $\Delta\Phi^j = A^j + B^j + F^j + H^j$, where A^j is the change in potential due to created inversions, B^j is the change in potential due to destroyed inversions, F^j is the change in potential due to visible inversions that change their type and H^j is the change in potential due to inversions that change their type from and to hidden inversions. We sum over all the nodes $\mathbf{b}_j$ encountered in the recursive procedure. This accounts for the access cost, paid transpositions and all the changes in potential.

We first fix the state $\text{STATE}(\mathbf{b}_j)$ of the node $\mathbf{b}_j$ at the time of access as $\text{STATE}(\mathbf{b}_j) = i$ and later take expectation over all possible states $[0, s)$. Given the state of the node $\mathbf{b}_j$, we split our analysis into two cases.

Case-1: The state of $\mathbf{b}_j$ is $\text{STATE}(\mathbf{b}_j) = i$ where $i \neq 0$ and hence $\mathbf{b}_j$ is not moved forward in the list.

(1) Reconfiguration cost is zero i.e., $C^j_{re}(t) = 0$, since there are no paid transpositions.
(2) Change in potential due to destroyed inversions is zero i.e., $B^j = 0$.
(3) No inversions change their type since the node remains in the state j. The change in potential due to inversions changing type is then $F^j = 0$.
(4) Since the node does not move forward, no inversions change from hidden to visible type i and vice versa i.e., $H^j = 0$.
(5) No inversions are created i.e., $A^j = 0$.

The expectation of $C^j_{re}(t) + \Delta\Phi^j$ in this case is given by,

$$E[C^j_{re}(t) + \Delta\Phi^j \mid (\text{STATE}(\mathbf{b}_j) = i)] = 0.$$

Case-2: The state of $\mathbf{b}_j$ is $\text{STATE}(\mathbf{b}_j) = 0$ and hence $\mathbf{b}_j$ is moved forward in the list by paid transpositions. The node $\mathbf{b}_j$ is moved to the position just after its blocking ancestor.

(1) Reconfiguration cost is $C^j_{re}(t) = |S_j| \cdot d = (|K_j \cap S_j| + |L_j \cap S_j|) \cdot d$.
(2) $|L_j \cap S_j|$ inversions which are initially of type 0 at the time of access, get destroyed since $\mathbf{b}_j$ moves forward i.e., $B^j = -(d + h_0) \cdot |L_j \cap S_j|$.
(3) There are no visible inversions which change their type and hence $F^j = 0$.
(4) From Lemma 4.5, the expected change in potential due to any inversion that changes from visible to hidden and from hidden to visible is zero i.e., $E[H^j] = 0$.
(5) At most $|K_j \cap S_j|$ new inversions are created. Each new inversion is either a hidden inversion or a visible inversion of type i. If the created inversion is hidden, then the increase in potential is $d + T$. If the created inversion is visible, then the inversion is of type i with probability π_i due to state independence of nodes (Observation 4.3) i.e., the expected change in potential is $\sum_{i=0}^{s} \pi_i \cdot (d + h_i) = (d + T)$. In total, $E[A^j] \le |K_j \cap S_j| \cdot (d + T)$.

The expectation of $C^j_{re}(t) + \Delta\Phi^j$ in this case is given by,

$$\begin{aligned} E[C^j_{re}(t) + \Delta\Phi^j \mid (\text{STATE}(\mathbf{b}_j) = 0)] \le {} & - |L_j \cap S_j| \cdot h_0 \\ & + |K_j \cap S_j| \cdot (2d + T). \end{aligned}$$

The expected amortized reconfiguration cost for the node $\mathbf{b}_j$ is obtained as follows:

$$\begin{aligned} E[C^j_{re}(t) + \Delta\Phi^j] &= \sum_{i=1}^{s-1} \pi_i \cdot E[C^j_{re}(t) + \Delta\Phi^j \mid (\text{STATE}(\mathbf{b}_j) = i)] \\ &\quad + \pi_0 \cdot E[C^j_{re}(t) + \Delta\Phi^j \mid (\text{STATE}(\mathbf{b}_j) = 0)] \\ &= \pi_0 \cdot (2d + T) \cdot |K_j \cap S_j| - h_0 \cdot \pi_0 \cdot |L_j \cap S_j|. \end{aligned}$$

□

We analyze the amortized cost of ALG for a single access request event i.e., access cost plus the total amortized reconfiguration cost over the recursive calls

Lemma 4.9. *The amortized cost of serving a request σ_t by ALG is $(k + 1) \cdot (1 + \pi_0 \cdot (2d + T))$.*

Proof. The amortized cost of ALG in serving a request is the sum of access cost plus the amortized reconfiguration cost. By Lemma 4.4, the change in potential due to the changes of states of the nodes which are not part of a blocking chain is zero.

It remains to consider the nodes in the blocking chain of the requested node. From Lemma 4.8, for each node $\mathbf{b}_j$, the cost of transpositions it participates in and the potential change due to its movements is at most $|K_j \cap S_j| \cdot (2d + T) \cdot \pi_0 - |L_j \cap S_j| \cdot h_0 \cdot \pi_0$. In total, transpositions of nodes $\mathbf{b}_j$ for $1 \le j \le B$ account for all transpositions at time t, thus we sum over all the nodes $\mathbf{b}_j$ to obtain the amortized reconfiguration cost of ALG.

$$\begin{aligned} E[C_{re}(t) + \Delta\Phi] &= \sum_{j=1}^{B} E[C^j_{re}(t) + \Delta\Phi^j] \\ &\leq \sum_{j=1}^{B} \left(\pi_0 \cdot (2d+T) \cdot |K_j \cap S_j| - h_0 \cdot \pi_0 \cdot |L_j \cap S_j|\right) \\ &\leq (k-B+1) \cdot (2d+T) \cdot \pi_0 - l \cdot h_0 \cdot \pi_0, \end{aligned}$$

where last inequality follows by Lemma 3.5, using both equation 1 and equation 2.

We bound the access cost of ALG by $C_{acc}(t) \leq k + \ell + 1$. Finally, using the result from Lemma 4.8, we obtain the amortized cost of ALG in serving a request,

$$\begin{aligned} E[a(t)] &\leq \overbrace{(k+l+1)}^{\text{access cost}} + \overbrace{((k-B+1) \cdot (2d+T) \cdot \pi_0 - l \cdot h_0 \cdot \pi_0)}^{\text{amortized reconfiguration cost}} \\ &\leq (k+1) \cdot (1 + \pi_0 \cdot (2d+T)) - B \cdot (2d+T) \cdot \pi_0 \\ &\leq (k+1) \cdot (1 + \pi_0 \cdot (2d+T)). \end{aligned}$$

The last inequality holds since $\pi_0 \cdot h_0 = 1$ from Kac's Lemma [17, 21] and $B \cdot \pi_0 \cdot (2d+T) \geq 0$.

□

C.4 Competitive Ratio of MMRF

Theorem 4.7. *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 $T = \sum_{i=0}^{s-1} \pi_i \cdot h_i$.*

Proof of Theorem 4.7. Consider any sequence of access requests σ. In order to prove our claim, it suffices to show that the expected amortized cost of MMRF in serving a request at any time t is $E[a(t)] \leq C \cdot C_{\text{OPT}}$, where $C = \max\{1 + \pi_0 \cdot (2d+T), 1 + \frac{T}{d}\}$.

- **Event-1:** We use the result from Lemma 4.9 to bound the amortized cost

$$\begin{aligned} E[C_{acc}(t) + C_{re}(t) + \Delta\Phi] &\leq (k+1) \cdot (1 + \pi_0 \cdot (2d+T)) \\ &\leq C_{\text{OPT}} \cdot (1 + \pi_0 \cdot (2d+T)), \end{aligned}$$

  where the last inequality holds as OPT pays at least $k+1$ for serving the access request at time t.
- **Event-2:** OPT pays d for any paid transposition and at most one new inversion is created. The created inversion is either a hidden or visible. If the created inversion is a hidden inversion then the change in potential is $d + T = d \cdot (1 + \frac{T}{d})$. If the created inversion is visible, using the state independence from Observation 4.3, the created visible inversion is of type i with probability π_i. Hence, the expected change in potential is $\Delta\Phi \leq \sum_{i=0}^{s-1} (d + h_i) \cdot \pi_i \leq d + T \leq d \cdot (1 + \frac{T}{d})$.

From Event-1 and Event-2, the expected amortized cost of MMRF is bounded by $\max\{1 + \pi_0 \cdot (2d+T), 1 + \frac{T}{d}\} \cdot C_{OPT}$ which concludes the proof.

□

D HANDLING INSERTIONS AND DELETIONS

In data structures, such as linked lists, the sets of nodes can change over time. Hence, many data structures (including the ones for packet classification) also support insertions and deletions in addition to access operations. This section discusses the feasibility of insertions. The algorithm's list may need node rearrangement before inserting a node. We propose assumptions that allow for constant competitiveness with insertions.

D.1 Model: Online Partially Ordered List Update

Consider the online partially ordered list *update* 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 the partial order edges with the nodes that are already in the list. The revealed constraints must be obeyed until the node is deleted. Note that a node may have constraints with nodes that will be inserted later, but this information is unknown until then.

D.2 Constant-competitiveness With Transitivity Assumption

In many practical applications, the partial order is transitive. In the introduction to this paper, we considered an information retrieval problem where we search a set of documents to find a piece of information. With documents overridden by new related documents, the documents must always be accessed in a chronological order. Here, we witness two relations: being overridden, and the chronological order, and both of them are transitive, and the partial order to maintain in the list (the intersection of the two) is transitive itself.

We build on assumptions present in the online list update problem [35], where insertions cost n and nodes are accessed before deletion. Additionally, we require *transitivity*.

Under these assumptions, any algorithm inserts the node σ_t at any position of its choice that respects the partial order *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 precedence constraints). Upon receiving a deletion request, the algorithm deletes the node.

Theorem D.1. *The algorithm Move-Recursively-Forward with insertions and deletions is strictly 4-competitive for $\alpha = 1$, with a transitive partial order, 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.

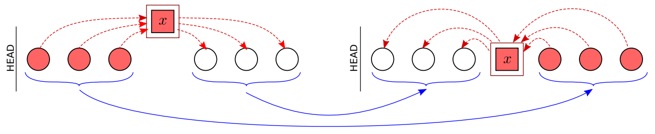


Figure 11: Consider a set of n nodes in a list, with no dependencies between them. Then, consider the insertion of a node x, which reveals the depicted dependencies (arrow points to the node that must precede in the list). Here, x has $\frac{n}{2}$ dependencies from the first half list (the red nodes) 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; thus, the cost of rearrangement is $(\frac{n}{2})^2$.

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 the 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 3.8, 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. □

D.3 Packet Classification Challenges: Non-Transitivity

The desired property of partial orders is *transitivity*: the relation that is a *transitive closure* of itself. 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 2.

N	Proto	Src IP	Dst IP	Src P	Dst P	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 2: 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 11. The table fields are: N is the rule priority, Action determines the label assigned to a packet, and the middle fields are packet header fields.

Lack of transitivity may lead to costly rearrangements required to insert a single node. Figure 11 depicts an example.

How to deal with insertions without preprocessing rules? Faced with the need to rearrange before insertion, we recommend finding a minimum rearrangement that satisfies new constraints. Examples such as Figure 11 cannot happen too often: each time they add dependencies, and the adversary would need to delete nodes to deceive the algorithm again. We leave the analysis of this strategy to future work.

E OVERVIEW OF ONLINE ALGORITHMS AND COMPETITIVE ANALYSIS

We measure the performance of an online algorithm by comparing its cost with the cost of an optimal offline algorithm OPT. Formally, let ALG(σ), resp. OPT(σ), 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 concept can be naturally extended to randomized algorithms; we say that a randomized online algorithm ALG is *c-competitive* if

$$E[\text{ALG}(\sigma)] \leq c \cdot \text{OPT}(\sigma) + b,$$

for any possible input sequence σ and a fixed constant b. In this context, the input sequence and the benchmark solution OPT are generated by an adversary.

In the randomized setting, competitive ratios for a given problem may vary depending on the adversary's power; in this work we design algorithms against an *oblivious adversary*. An adversary of this type only knows the description of the algorithm, and it generates the entire input sequence before the start of the algorithm. For a randomized algorithm, the oblivious adversary is aware of the probability distribution used by the algorithm; however, it has no knowledge of the algorithm's random choices.

For a broader overview of the competitive analysis framework, we refer the reader to [10].