Online List Access with Precedence Constraints

Maciej Pacut
Faculty of Computer Science, University of Vienna, Austria

Juan Vanerio
Faculty of Computer Science, University of Vienna, Austria

Vamsi Addanki
Faculty of Computer Science, University of Vienna, Austria

Arash Pourdamghani
Faculty of Computer Science, University of Vienna, Austria

Gabor Retvari
Department of Telecommunications and Media Informatics, Budapest University of Technology and Economics Budapest, Hungary

Stefan Schmid
Faculty of Computer Science, University of Vienna, Austria

Abstract

This paper considers a natural generalization of the online list access problem in the paid exchange model, where additionally there can be precedence constraints ("dependencies") among the nodes in the list. For example, this generalization is motivated by applications in the context of packet classification. Our main contributions are constant-competitive deterministic and randomized online algorithms, designed around a procedure Move-Recursively-Forward, a generalization of Move-To-Front tailored to handle node dependencies. Parts of the analysis build upon ideas of the classic online algorithms Move-To-Front and BIT, and address the challenges of the extended model. We further discuss the challenges related to insertions and deletions.

2012 ACM Subject Classification Theory of computation → Online algorithms

Keywords and phrases Online algorithms, competitive analysis, list access, packet classification

1 Introduction

The list access problem is a fundamental online algorithmic problem, originally introduced by Sleator and Tarjan in 1985 in their seminal work on amortized analysis [19]. In a nutshell, in the list update problem, we consider a linked list data structure in which accessing a node costs proportionally to its distance from the head of the list. We are looking for an algorithm that adaptively reorders the list in order to minimize the access cost, when faced with a sequence of access requests revealed in an online manner by an adversary. As usual in competitive analysis, we are interested in the competitive ratio: we compare the overall cost of an online algorithm to the cost of an optimal offline algorithm which knows the access sequence ahead of time.

This paper initiates the study of a generalization of this classic problem where there can be precedence constraints among the nodes in the list. These precedence constraints define a relative order among some nodes in the list, which means that some nodes must be placed before others. We assume that the partial order is given in form of a directed acyclic graph that induces this order. This generalization is natural and motivated by practical applications, e.g., in the context of packet classification in communication networks. For example, rules in an IP router or in a firewall are often ordered to ensure a correct handling. A detailed motivating example will follow below.

The consideration of constraints poses an algorithmic challenge. In contrast to existing online algorithms for the list access problem, such as Move-to-Front and BIT which upon access, can flexibly move nodes towards the head of the list, constraints may prevent such optimizations or at least make them costly. The constraints raise the question whether in order to move forward an accessed node u, it may be worthwhile to recursively move forward the nodes v on which u depends as well, and to which extent. In particular, it is easy to see that if the directed acyclic graph describing the precedence constraints has depth d, then a strategy that aggressively moves nodes forward is at best $\Omega(d)$-competitive.

Furthermore, we consider insertions and deletions, a particularly challenging, yet integral part of the problem. Although insertions and deletions were handled in the original work on list update [19], many papers considering the paid exchange model do not address this issue, and requests are restricted to accesses only [2, 13, 14]. We start with results for the access-only variant of the problem, and then we consider insertions and deletions, and discuss the assumptions needed to handle them efficiently.

Our main contribution in this paper are two constant-competitive online algorithms for the list access problem with precedence constraints, a deterministic one and a randomized one. Before we present our contributions in detail, we present our model. Subsequently, we give a practical motivation for the problem and then put our contribution into perspective with regard to related work.

1.1 Preliminaries and Competitive Ratio

The sequence of requests σ is revealed one-by-one, in an online fashion. Upon seeing a request, the algorithm must serve it without the knowledge of future requests. We measure the performance of an online algorithm by comparing to the performance of an optimal offline algorithm. Formally, let $\mathrm{DET}(\sigma)$, resp. $\mathrm{OPT}(\sigma)$, be the cost incurred by a deterministic online algorithm DET, resp. by an optimal offline algorithm OPT, for a given sequence of requests σ. In contrast to DET, 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 that provide worst-case guarantees. In particular, DET is said to be α-*competitive* if there is a constant β, such that for any input sequence σ it holds that

$$\mathrm{DET}(\sigma) \leq \alpha \cdot \mathrm{OPT}(\sigma) + \beta.$$

Note that β cannot depend on input σ but can depend on other parameters of the problem, such as the number of nodes. The minimum α for which DET is α-competitive is called the *competitive ratio* of DET. We say that DET is *strictly* α-*competitive* if additionally $\beta = 0$.

We say that a randomized online algorithm RAND is α-*competitive* if

$$E[\mathrm{RAND}(\sigma)] \leq \alpha \cdot \mathrm{OPT}(\sigma) + \beta$$

for any input sequence σ and a fixed constant β. The input sequence and the benchmark solution OPT is generated by an adversary. We distinguish between the notion of competitiveness against various adversaries, having different knowledge about RAND and different knowledge while producing the offline benchmark solution OPT. Competitive ratios for a given problem may vary depending on the power of the adversary. In our work, we design algorithms against an *oblivious offline adversary* that must produce an input sequence in advance, merely knowing the description of the algorithm it competes against (in particular, it may have access to probability distributions that the algorithm uses, but not the random outcomes), and pays an optimal offline cost for the sequence. For a comprehensive overview of adversary types, see [7].

1.2 Model

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

The list and the requests. Consider a set of n nodes arranged in a linked list. Over time, we receive requests from a sequence σ, describing *accesses* to nodes from our list[1]. Upon receiving an access 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 accessed node. Accessing the node at position i in the list costs i (accessing the first node in the list costs 1).

The precedence constraints (dependencies). We are given a directed acyclic graph G, ofter called the *dependency graph*. The dependency graph induces a partial order among the nodes that is equivalent to the reachability relation in G. The nodes must obey the partial order in the list at all time. We say that a node v is a *dependency* of a node u if there exists an edge (u, v) in G. Then, in every configuration of the list, v must be in front of u. We assume that the given initial configuration of the nodes obeys the precedence constraints induced by G.

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

Our goal is to design online algorithms that perform closely to offline optimal algorithms. For a more detailed description of the model in both the deterministic and randomized setting, see Appendix 1.1.

1.3 Our Contributions

We initiate the study of a natural and practically motivated generalization of the online list access problem where there can be precedence constraints among nodes. Our main contribution are two constant-competitive online algorithms for this problem. Our algorithms are designed around a recursive procedure Move-Recursively-Forward that generalizes the Move-To-Front algorithm, accounting for dependencies. We also shed light on the challenges of supporting insertions and deletions in the setting with precedence constraints.

1.4 Novelty and Related Work

Already various online problems have been studied in settings with dependencies and precedence constraints. In scheduling with precedence constraints [5], a job may be scheduled only after all its predecessors are completed. Another example which is more closely related to our work is caching with dependencies [6]: the problem is motivated by the fact that routers in communication networks forward packets based on the longest common prefix match, and hence an element can be brought into the cache only if all its dependencies are in the cache. However, we are not aware of any work on online list access problems with dependencies.

[1] We refer to Section 4 for a discussion about node insertions and deletions.

List access problems were already studied for several different cost models. Much prior work considers the *paid exchange model* (studied in this paper) in which every transposition incurs a cost to the algorithm. For the setting without dependencies, it is known that no deterministic algorithm can be better than 3-competitive; this lower bound is due to Reingold et al. [18]. The survey [13] suggests that the deterministic algorithm Move-To-Front-every-other-access can be shown to be 3-competitive. In the randomized setting, the best known algorithm is an extension of COUNTER algorithm [4, 18] that is $\frac{29}{11} \approx 2.63$-competitive against oblivious adversaries, and a lower bound (in the paid exchange model) against the oblivious adversary is 1.8654 [3].

There exist interesting results on list access problems with lower list rearrangement cost. The most popular variant is the *free exchange model*, in which moving an accessed node forward is free. In such a setting, the algorithm Move-To-Front is 2-competitive, and this result is tight [19]. Other papers considered settings in which rearrangements of large portions of the list have linear costs [12, 16]. The list access problem was also studied under a *generalized access cost* model [19], where the cost of accessing an i-th node is a general function $f(i)$.

The main technical challenge and novelty in our paper is the design of an efficient operation for moving nodes closer to the front of the list. The operation Move-To-Front, known from the setting without dependencies, would violate the order of nodes, and new algorithmic ideas are needed that account for dependencies. Our candidate procedure Move-Recursively-Forward (cf. Section 2) moves the accessed node forward, but additionally it moves forward a carefully chosen set of its dependencies, while retaining the linear cost of node rearrangements.

The candidate operation is expected to decrease the cost of future accesses. To capture this property, we measure the distance to an optimal offline algorithm, and count the number of *inversions* of pairs of nodes. We argue that some crucial types of inversions are destroyed during the operation, to show that the algorithm is making progress towards the optimal solution. The challenge lies in doing so without knowing the optimal solution's configuration.

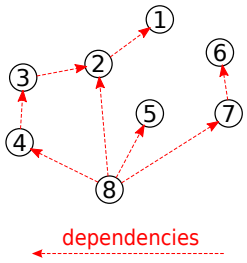
1.5 A Practical Motivation

List access with precedence constraints is motivated by practical applications in the context of packet classification [10], a fundamental task performed by switches, routers and middleboxes (e.g., firewalls) in communication networks [1, 8, 10, 11, 20]: upon the arrival of a packet, its header it inspected in order to determine to which flow it belongs, and hence which predefined rule needs to be applied to process it.

A common application for packet classifiers is implementing traffic filtering in *network firewalls*, where the rules distinguish legitimate packets, which are to be accepted by the firewall, from malicious traffic that needs to be dropped [1, 11].

These rules often have dependencies, e.g., the matched domains overlap and the rules need to be performed in a specific (partial) order, i.e., a rule assumes that another rule has been checked before. Figure 1 depicts an example of a dependency DAG induced by a table of rules.

Packet classification is implemented using processing pipelines where match-action rules are organized in a data structure such as a linked list. A packet is first matched against the rule at the head of the list, and then, depending on the action, passed on to be classified further by the rules down in the list, according to the list order (respecting the precedence constraints). With this perspective, matching the packet to its rule in a list can be seen as the find operation for a node in a list.



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

Due to its simplicity, this linear lookup structure is commonly applied in practice; e.g., in the default firewall suite of the Linux operating system kernel called `iptables` [15], the OpenFlow reference switch [17], and in many QoS classifiers. This approach however can be inefficient if a significant part of the list needs to be traversed before the final packet classification can be performed.

The motivation behind our optimization model is to render the data structure used for packet classification self-adjusting: if frequently used rules appear closer to the head of the list, the overhead of list traversal could be improved significantly, and hence packet classification sped up. This requires monitoring the importance of rules, which may change over time, and dynamically promote rules accordingly, while accounting for the precedence constraints to ensure policy compliance.

2 Deterministic Algorithm

In this section we propose a deterministic 4-competitive algorithm DET for online list access with dependencies. We design the algorithm so that the cost of reorganizing the list after access (by operation *Move-Recursively-Forward*) is in the order of the cost of the access (Lemma 2.5). To use the potential function analysis framework, introduced by Sleator and Tarjan [19], we assure that the algorithm properly influences the potential change to bound the amortized cost of the algorithm (we elaborate in Section 2.2).

Algorithm DET. The algorithm uses a recursive procedure *Move-Recursively-Forward* (MRF for short). The procedure MRF(y) moves the node y forward (by transposing it with the preceding nodes) until it encounters any of its dependency nodes, say z, and recursively calls MRF(z). Upon receiving an access request to a node σ_t, DET locates σ_t on the list and invokes the procedure MRF(σ_t). In Figure 2, we depict an example run of MRF after serving a request by DET.

We present the pseudocode of DET in Algorithm 1. We say that a node y is a *direct dependency* of a node z if y is the dependency of z that is located at the furthest position on the list. By $\mathsf{pos}(z)$ we denote the position of node z in the list maintained by the algorithm, counting from the head of the list (recall that the position of the first node is 1).

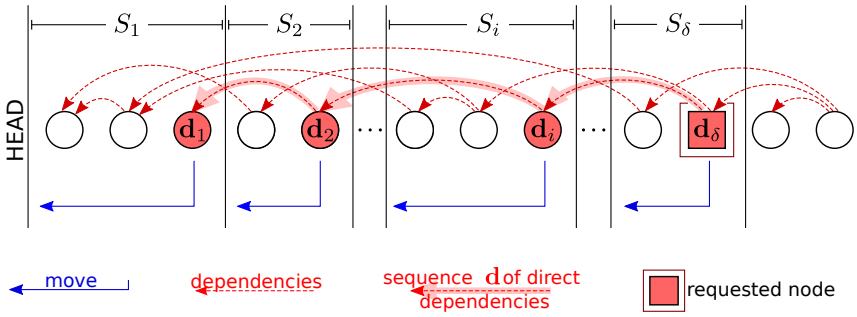


Figure 2 An example of handling a request by the algorithm DET. The red nodes $\mathbf{d}_i$, formally introduced in Section 2.1, are the nodes that the algorithm moves forward (to the position denoted by a blue arrow). Among multiple dependencies (red dashed arrows), the algorithm DET moves forward only the nodes that constitute the path of the direct dependencies (bold red arrow).

Algorithm 1 The algorithm DET.

Input : An access request to node σ_t
Access σ_t
Run the procedure MRF(σ_t)

Procedure MRF(y):
  if y has no dependencies **then**
    Move y to the front of the list
  else
    Let z be the direct dependency of y
    Move node y to $\mathsf{pos}(z) + 1$
    Run the procedure MRF(z)

We claim that this algorithm is 4-competitive. Before proving the claim (Theorem 2.6), we overview the analysis, introduce sets relevant to the analysis and reason about their size.

2.1 Analysis overview

To perform an amortized analysis of DET's cost, we use a potential function similar to the one used in the analysis of Move-To-Front [19], defined in terms of number of inversions. We bound the change in the potential due to node rearrangements by a similar function as in Move-To-Front's analysis (Theorem 2.1). In our case multiple nodes change their positions, and the proof requires a careful analysis of inversions affected by each of them. We note that the cost of node rearrangements after accessing a node is bounded by the cost of accessing the requested node (Lemma 2.5).

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

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

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

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

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

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

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

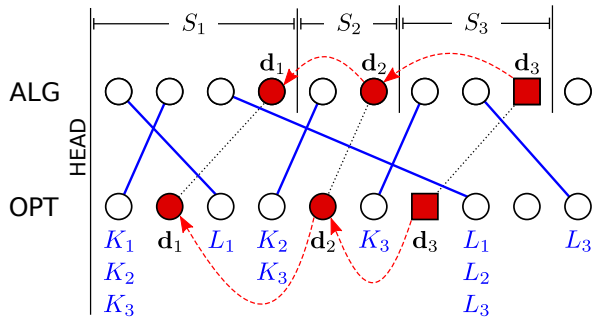


Figure 3 The example illustrates central definitions of sets of nodes used in our analysis. We depict the positions of nodes in both DET's and OPT's list (joined by solid blue lines). The dotted black lines between the nodes $\mathbf{d}_i$ help in determining the assignment of nodes to sets: in K_i we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

2.2 Bounding the Change of Inversions

The operation Move-Recursively-Forward, defined as a procedure in Algorithm 1 was designed to mimic the operation Move-To-Front [19] in introducing and destroying inversions.

▶ **Theorem 2.1.** *Consider a request to the node σ_t, and fix a configuration of OPT at time t. Then, the change in the number of inversions due to DET's node rearrangement after serving the request is at most $k - \ell$.*

To prove this claim, we consider the influence of the Move-Recursively-Forward operation on values k and ℓ (defined for the currently requested node) by inspecting the sets K_j and L_j (defined for the nodes $\mathbf{d}_j$). We separately bound the number of inversions created (Lemma 2.3) and destroyed (Lemma 2.4). Before showing these claims, we inspect the basic relations between the sets K_j, L_j and S_j (Lemma 2.2).

▶ **Lemma 2.2.** *The following relations hold*

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

See Figure 3 for an illustration of a graphical argument.

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

Next, we prove the equality (2). We show inclusion both ways. Consider any element $y \in L_j$. The sets $\{S_j\}$ 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 $\mathbf{d}_i$ in DET's list,
- y is after $\mathbf{d}_j$ in OPT's list (it belongs to L_j), and hence it is after $\mathbf{d}_i$ in OPT's list (the order of $\mathbf{d}$ is fixed due to dependencies).

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

▶ **Lemma 2.3.** *Consider a request to the node σ_t, and fix a configuration of OPT at time t. Due to rearrangements after serving the request, DET creates at most k inversions.*

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

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

$$\sum_{j=1}^{\delta} A_j \leq \sum_{j=1}^{\delta} |S_j \cap K_j| = |\bigcup_{j=1}^{\delta}(S_j \cap K_j)| \leq |\bigcup_{j=1}^{\delta} K_j|,$$

where the second step holds as the sets S_j are disjoint, and the last step follows by basic properties of sets. By Lemma 2.2, equation (1), we have $|\bigcup_{j=1}^{\delta} K_j| = |K_\delta|$, thus by combining the above inequalities, we have $\sum_{j=1}^{\delta} A_j \leq |K_\delta| = k$, and we conclude that the claim holds. ◀

▶ **Lemma 2.4.** *Consider a request to the node σ_t, and fix a configuration of OPT at time t. Due to rearrangements after serving the request, DET destroys at least ℓ inversions.*

Proof. Let D_j be the number of inversions destroyed by moving a single node $\mathbf{d}_j$ by DET, for $j \in [1, \delta]$. To bound D_j, we inspect the set S_j of the nodes that $\mathbf{d}_j$ overtakes, and we reason based on their positions in OPT's list. Moving $\mathbf{d}_j$ forward destroys all inversions with nodes

in $S_j \cap L_j$. No other node changes its relation to the set S_j, hence the inversions for nodes in S_j are influenced only by the movement of $\mathbf{d}_j$. This gives us the bound $D_j \geq |S_j \cap L_j|$.

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

$$\sum_{j=1}^{\delta} D_j \geq \sum_{j=1}^{\delta} |S_j \cap L_j| = |\bigcup_{j=1}^{\delta} (S_j \cap L_j)|,$$

where the second step holds as the sets S_j are disjoint. By Lemma 2.2, equation (2) we have $|\bigcup_{j=1}^{\delta}(S_j \cap L_j)| = |\bigcup_{j=1}^{\delta} L_j|$. Finally, by basic properties of sets, $|\bigcup_{j=1}^{\delta} L_j| \geq |L_\delta| = \ell$, and by combining all the above bounds we have $\sum_{j=1}^{\delta} D_j \geq \ell$, and we conclude that the lemma holds. ◀

Combining Lemmas 2.3 and 2.4 gives us the joint bound on the change in the number of inversions, and proves the Theorem 2.1. We note that this bound is consistent with the bound on the changes in inversions for the algorithm Move-To-Front [19], where the inversions were considered with respect to the accessed node only.

2.3 Bounding the Competitive Ratio

Finally, we show the main result of this section: the competitive ratio of the algorithm DET is 4. First, we observe that the cost of the rearrangements after handling each request is bounded by the cost of the access (Lemma 2.5), and then we apply the bounds on the number of inversions (Theorem 2.1), to finally bound the ratio using a potential function argument (Theorem 2.6).

▶ **Lemma 2.5.** *Consider a single request to a node y at position $\mathsf{pos}(y)$ handled by the algorithm DET. The rearrangements after serving the request to y costs at most $\mathsf{pos}(y)$.*

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

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

Let $\mathbf{d}$ be the set of the dependencies of y, and let δ be the number of y's dependencies (including y). Then, the node $\mathbf{d}_i$ moves to the position $\mathsf{pos}(\mathbf{d}_{i-1}) + 1$ (for formality of the argument, we assume an artificial dependency at the head of the list, $\mathsf{pos}(\mathbf{d}_0) = 0$). Thus, in total the number of transpositions is

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

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

Events overview. In the analysis, we distinguish between the following types of events that occur throughout algorithms' execution:

(A) A *request 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 DET throughout this event.

Potential function. We define the potential function Φ in terms of number of inversions in DET's list with respect to OPT's list. Precisely, the potential function is defined as *twice the number of inversions*.

Finally, we prove the main result of this section, that the algorithm DET is 4-competitive.

▶ **Theorem 2.6.** *The algorithm DET is strictly 4-competitive.*

The proof uses a potential function argument similar to Move-To-Front [19], and it internally uses the Theorem 2.1 to reason about changes in inversion due to Move-Recursively-Forward runs.

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

First, we bound the cost of DET incurred while serving an access request to a node σ_t at time t (a *request event*). This cost consists of the access cost and the rearrangement cost. To access the node σ_t, the algorithm incurs the cost $\mathsf{pos}(\sigma_t)$, and by Lemma 2.5 the rearrangement cost is bounded by $\mathsf{pos}(\sigma_t)$, hence $C_{DET}(t) \leq 2 \cdot \mathsf{pos}(\sigma_t)$.

Next, we bound the amortized cost for every access request served by DET. The amortized cost is $C_{\mathrm{DET}}(t) + \Delta\Phi(t)$ for each time t. By Theorem 2.1, we bound the change in the number of inversions due to DET's rearrangement after serving the request at time t by $\Delta I \leq k - \ell$. Thus, the change in the potential is $\Delta\Phi(t) \leq 2(k - \ell)$. As $\mathsf{pos}(\sigma_t) = k + \ell + 1$, combining these bounds gives us

$$C_{\mathrm{DET}}(t) + \Delta\Phi(t) \leq 2 \cdot \mathsf{pos}(\sigma_t) + 2\,(k - \ell) \leq 2\,(k + \ell + 1) + 2\,(k - \ell) \leq 4 \cdot C_{\mathrm{OPT}}(t),$$

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

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

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

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

We assume that DET 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 $C_{\mathrm{DET}}(\sigma) \leq 4 \cdot C_{\mathrm{OPT}}(\sigma)$. ◀

3 Randomized Algorithm

In this section we propose a 3-competitive randomized algorithm RAND for online list access with dependencies. We design the algorithm around concepts from the algorithm BIT [18] and the procedure Recursively-Move-Forward from Section 2.

Algorithm RAND. We maintain an additional bit of memory for each node: we assign a binary counter $\mathbf{b}(y)$ to each node y with initial value chosen uniformly at random from $\{0, 1\}$. Upon receiving a request to a node σ_t, if its bit was 0, we call the procedure $\mathrm{MRF}(\sigma_t)$, defined in Algorithm 1, and then we flip the bit of σ_t regardless of its previous value. We present the pseudocode of RAND in Algorithm 2.

Algorithm 2 The randomized algorithm RAND.

Initialization : Assign a bit to each node uniformly at random
Input : A request to node σ_t
Access σ_t
if $\mathbf{b}(\sigma_t)$ is 0 **then**
    Run the procedure $\mathrm{MRF}(\sigma_t)$
Flip $\mathbf{b}(\sigma_t)$

Analysis. We compare the costs of RAND and an optimal offline algorithm OPT on σ using the potential function Φ. In the definition of Φ, we distinguish between two types of inversions; we say that an inversion of the ordered pair (y, σ_t) is a *type i inversion* if $\mathbf{b}(\sigma_t) = i$ for $i \in \{0, 1\}$. Precisely, we use the potential function

$$\Phi = \frac{5}{2} \cdot I_0 + \frac{7}{2} \cdot I_1,$$

where I_0 is the number of inversions of type 0 and I_1 is the number of inversions of type 1.

In proving our claims, we use an observation that the values bits of RAND are independent and remain uniformly distributed as they change over time.

▶ **Observation 3.1.** *For any node y, the value of $\mathbf{b}(y)$ is 0 with probability $\frac{1}{2}$ and 1 with probability $\frac{1}{2}$ at any time, and is independent of its position in OPT's list and other nodes' bits.*

We distinguish between the following types of events that occur throughout algorithm's execution:

(A) An *access request event* $R^i(\sigma_t)$ for $i \in \{0, 1\}$. The algorithm serves the request to the node σ_t with $\mathbf{b}(\sigma_t) = i$ and performs the Move-Recursively-Forward procedure if $\mathbf{b}(\sigma_t) = 0$. 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 RAND throughout this event.

We start by analyzing the amortized cost of RAND for a single access request event (Lemma 3.2). If the accessed node's bit has value 1, then none of the nodes change their position, but some inversions may change their type and entail the change in potential, accounted in the amortized cost. If the accessed node's bit has value 0, then the algorithm runs the procedure Move-Recursively-Forward, and we bound the amortized cost of this rearrangement.

Let $C_{\mathrm{RAND}}(t)$ and $C_{\mathrm{OPT}}(t)$ be the cost incurred at time t by RAND and OPT, respectively. Recall that k is the number of nodes before σ_t in both RAND's and OPT's lists, and ℓ is the number of nodes before σ_t in RAND's list, but after σ_t in OPT's list. For succinctness, we define $E_0[Y]$ as the expected value of a variable Y when $\mathbf{b}(\sigma_t) = 0$, and $E_1[Y]$ as the expected value of Y when $\mathbf{b}(\sigma_t) = 1$.

▶ **Lemma 3.2.** *Consider a request to a node σ_t handled by the algorithm RAND.*

(1) If $\mathbf{b}(\sigma_t) = 1$, then $E_1[C_{RAND}(t) + \Delta\Phi] \leq k + 1$.

(2) If $\mathbf{b}(\sigma_t) = 0$, then $E_0[C_{RAND}(t) + \Delta\Phi] \leq 5(k + 1)$.

Proof. Before bounding the amortized cost, we split the change in potential $\Delta\Phi$, distinguishing between three reasons for the potential to change: (1) the inversions added, (2) the inversions destroyed and (3) the inversions that changed their type. Precisely, we split the change of the potential into three parts, $\Delta\Phi = A + D + F$, where A is a random variable denoting the change in potential due to new inversions added, D is a random variable denoting the change in potential due to old inversions destroyed and F is a random variable denoting the change in potential due to inversions that flipped (changed) their type. Together, these three parts account for all the changes while serving a request by RAND.

Recall that $E_0[Y]$ is the expected value of a variable Y when $\mathbf{b}(\sigma_t) = 0$, and $E_1[Y]$ is the expected value of Y when $\mathbf{b}(\sigma_t) = 1$.

Consider the case $\mathbf{b}(\sigma_t) = 1$. The algorithm flips the bit of σ_t to 0, and it does not perform any node rearrangements. No inversions are destroyed or added, thus $E_1[D] = 0$ and $E_1[A] = 0$. The algorithm incurs the cost $k + \ell + 1$ due to access, and type of ℓ inversions flipped from 1 to 0, therefore

$$E_1[C_{\text{RAND}}(t) + A + D + F] \leq (k + \ell + 1) - \ell = k + 1,$$

which concludes the first claim of the lemma.

Consider the case $\mathbf{b}(\sigma_t) = 0$. The algorithm invokes the MRF procedure and then flips the bit of σ_t to 1. We split the amortized cost into four parts:

(1) **The cost incurred by the algorithm.**
The algorithm incurs the cost $k+\ell+1$ for access and by Lemma 2.5 at most $k+\ell+1$ for rearrangements, thus

$$E_0[C_{\text{RAND}}(t)] \leq 2 \cdot (k + \ell + 1).$$

(2) **The change in potential due to inversions that flipped type from 0 to 1.**
Out of all ℓ inversions with respect to σ_t, some are destroyed and the others flipped type. The bit of σ_t changes to 1, thus some inversions may have flipped type from 0 to 1. Precisely, all inversions (y, σ_t) for $y \in L_\delta$ flip their type unless σ_t moves in front of y. Any single such flip from type 0 to type 1 increases the potential by $\frac{7}{2} - \frac{5}{2} = 1$. The node σ_t moves in front of the set of nodes S_δ, thus

$$E_0[F] \leq (\frac{7}{2} - \frac{5}{2}) \cdot (\ell - |S_\delta \cap L_\delta|) = \ell - |S_\delta \cap L_\delta|.$$

(3) **The change in potential due to inversions destroyed.**
When the procedure Move-Recursively-Forward is invoked, each node $\mathbf{d}_i$ for $i = \{1, 2, \ldots, \delta\}$ moves in front of the nodes from S_i (recall that $\sigma_t = \mathbf{d}_\delta$). Thus, the inversions $S_i \cap L_i$ are destroyed, and the change in the potential depends on the bit of $\mathbf{d}_i$. The bit of σ_t is determined, $b(\sigma_t) = 0$, and by Observation 3.1, the bits of $\mathbf{d}_i$ for $i < \delta$ are equally likely to be 0 or 1. If the bit of the moving node is 0, a single inversion destroyed contributes $-\frac{5}{2}$ to the potential change, otherwise it contributes $-\frac{7}{2}$, and we bound the total change in the potential by

$$\begin{aligned} E_0[D] &\leq \underbrace{-\frac{5}{2} \cdot (|S_\delta \cap L_\delta|)}_{\sigma_t = \mathbf{d}_\delta} - \sum_{i=1}^{\delta-1} \underbrace{(\frac{1}{2} \cdot \frac{5}{2} + \frac{1}{2} \cdot \frac{7}{2}) \cdot |S_i \cap L_i|}_{\text{nodes } \mathbf{d}_i \text{ for } i<\delta} \\ &\leq \quad -\frac{5}{2} \cdot (|S_\delta \cap L_\delta|) - 3 \cdot \sum_{i=1}^{\delta-1} |S_i \cap L_i|. \end{aligned}$$

(4) **The change in potential due to inversions added.**
When the procedure Move-Recursively-Forward is invoked, each node $\mathbf{d}_i$ for $i = \{1, 2, \ldots, \delta\}$ moves in front of the nodes from S_i (recall that $\sigma_t = \mathbf{d}_\delta$). Thus, the inversions $S_i \cap K_i$ will be added. By Observation 3.1, bits of the nodes are equally likely to be 0 or 1. If bit of the node that caused inversion is 0, a single inversion added contributes $\frac{5}{2}$ to the potential change, otherwise it contributes $\frac{7}{2}$, and we bound the total change in the potential in the following way

$$E_0[A] \leq \sum_{i=1}^{\delta} \frac{1}{2} \cdot (\frac{5}{2} \cdot |S_i \cap K_i|) + \frac{1}{2} \cdot (\frac{7}{2} \cdot |S_i \cap K_i|) \leq \sum_{j=1}^{k} \left(\frac{1}{2} \cdot \frac{5}{2} + \frac{1}{2} \cdot \frac{7}{2} \right) = 3k,$$

where in the second inequality we used $\sum_{j=1}^{\delta} |S_j \cap K_j| = |\bigcup_{j=1}^{\delta} (S_j \cap K_j)| \leq |\bigcup_{j=1}^{\delta} K_j|$, since we know the sets S_j are disjoint.

Finally, we combine the bounds (1-4) for the case $b(\sigma_t) = 0$. We highlight that it is necessary to bound the expected cost of D and F together, obtaining

$$\begin{aligned} E_0[C_{\text{RAND}}(t) + D + F] &\leq \underbrace{2 \cdot (k + \ell + 1)}_{C_{\text{RAND}}} + \underbrace{(\ell - |S_\delta \cap L_\delta|)}_{F} \underbrace{-\frac{5}{2} \cdot (|S_\delta \cap L_\delta|) - 3 \cdot \sum_{j=1}^{\delta-1} |S_j \cap L_j|}_{D} \\ &\leq 2(k+1) + 3\ell - 3 \cdot (|S_\delta \cap L_\delta|) - 3 \cdot \sum_{j=1}^{\delta-1} |S_j \cap L_j| = 2(k+1), \end{aligned}$$

where the last equality holds because of $\bigcup_{j=1}^{\delta} (S_j \cap L_j) = \bigcup_{j=1}^{\delta} L_j$ by Lemma 2.2 and $\ell = |\bigcup_{j=1}^{\delta} L_j|$. Including the bound for added inversions for the case $\mathbf{b}(\sigma_t) = 0$ gives us the total expected cost $E_0[C_{\text{RAND}}(t) + A + D + F] \leq 5(k+1)$, which concludes the second claim of the lemma. ◀

Now, we are ready to prove the main result of this section.

▶ **Theorem 3.3.** *The algorithm RAND is strictly* 3*-competitive against an oblivious offline adversary.*

Proof. Consider a sequence of access requests σ. We bound the ratio of RAND to OPT using a potential function argument.

First, we consider a *request event*. We use the Lemma 3.2 to bound the amortized cost, and we combine the bounds for the cases $\mathbf{b}(\sigma_t) = 0$ and $\mathbf{b}(\sigma_t) = 1$. By Observation 3.1, the probability that $\mathbf{b}(\sigma_t) = 0$ is $\frac{1}{2}$, and the probability that $\mathbf{b}(\sigma_t) = 1$ is $\frac{1}{2}$. Hence, we bound the expected amortized cost of RAND in the following way, and relate it to the cost of OPT

$$\begin{aligned} E[C_{\text{RAND}}(t) + \Delta\Phi(t)] &= \frac{1}{2} \cdot E_0[C_{\text{RAND}}(t) + \Delta\Phi(t)] + \frac{1}{2} \cdot E_1[C_{\text{RAND}}(t) + \Delta\Phi(t)] \\ &\leq \frac{1}{2} \cdot 5(k+1) + \frac{1}{2} \cdot (k+1) = 3(k+1) \leq 3 \cdot C_{\text{OPT}}(t), \end{aligned}$$

where the last inequality holds as $C_{\text{OPT}}(t) \geq k + 1$.

Next, we show that this bound accounts for *paid exchange events*, the paid rearrangements by OPT at time t (the second type of event in our analysis). Each transposition of OPT increases the number of inversions by 1, which increases the LHS by $\frac{1}{2} \cdot \frac{5}{2} + \frac{1}{2} \cdot \frac{7}{2} = 3$ in expectation, because bits of the nodes are independent of the OPT decisions, and both types of inversions are equally probable to be created. To complement the bound at the RHS, for each transposition OPT pays 1, which increases the RHS by 3.

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

$$E[C_{\text{RAND}}(\sigma) + \Phi(m) - \Phi(0)] \leq 3 \cdot C_{\text{OPT}}(\sigma).$$

We assume that RAND and OPT started with the same list, thus the initial potential $\Phi(0) = 0$, and the potential is always non-negative, therefore $\Phi(m) \geq 0$, and we conclude that $E[C_{\text{RAND}}(\sigma)] \leq 3 \cdot C_{\text{OPT}}(\sigma)$. ◀

Finally, we discuss the competitive ratio in comparison to well-known randomized algorithm for the classic list access. The algorithm BIT (that inspired the design of RAND) is 2.75-competitive in the paid exchange model without precedence constraints [18]. The algorithm RAND achieves slightly worse competitive ratio than BIT due to inversions that change their type. The algorithms differ in the operation on the accessed node: the operation Move-To-Front destroys all inversions with respect to the node that changes its bit, and the operation Move-Recursively-Forward may change the type of some inversions.

We note that the improvement beyond the competitive ratio 3 with randomized algorithms is non-obvious. We inspected a wide range of known randomized algorithms (such as COUNTER, RANDOM RESET and Markov-Move-To-Front [4, 9, 18]) and we report that simply changing the operation from Move-To-Front to Move-Recursively-Forward still results in the competitive ratio 3. The best lower bound for list update in the paid exchange model against oblivious adversaries is 1.8654 [3].

4 Handling Insertions and Deletions

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

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

Inserting a new node may require extensive rearrangements to even meet the precedence constraints. In Figure 4 we present an example of such a costly rearrangement required to insert a single node.

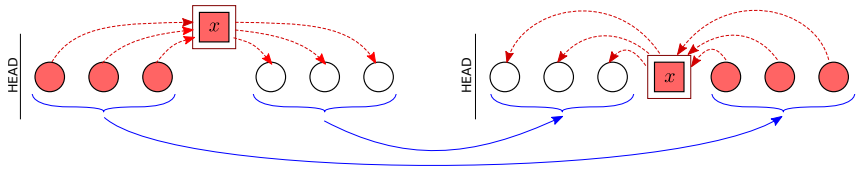


Figure 4 Consider a set of n nodes in a list, with no dependencies between them. Then, consider an insertion of a node x, which reveals the depicted dependencies. Here, x has $\frac{n}{2}$ dependencies from the first half list (the red nodes) to x, and $\frac{n}{2}$ dependencies from x to the nodes of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

Constant-competitiveness with additional assumptions. To remedy incurring such a large cost at insertion, we discuss and rationalize the assumptions that guarantee a feasible place to insert any node without rearrangements. With these assumptions, we retain constant-competiveness of our algorithms DET and RAND. The Assumption 4.1 concerns the cost of each insertion, and the Assumption 4.2 concerns the structure of the dependency graph.

▶ **Assumption 4.1.** *Any algorithm incurs cost n inserting a node at any position in the list.*

Comment on Assumption 4.1. This assumption is common in the literature, as also in the classic list access problem, insertion require similar assumptions: nodes are inserted at the last position, and nodes are accessed before deletion [19]. We rely on an equivalent assumption in the proposed solution. Assumption 4.1 may be justified by a need to check the incoming node against possible dependencies with existing nodes. Upon receiving an insertion request of a node σ_t, an algorithm first determines the change in the dependency graph, by comparing σ_t with all nodes in the list. The cost of determining dependencies equals the number of nodes in the list at the time of σ_t's arrival. Then, any algorithm inserts the node σ_t at any position of its choice that respects the dependency DAG G, *without incurring further cost.*

▶ **Assumption 4.2.** *The dependency DAG G is* transitive.

Comment on Assumption 4.2. Consider the universe of all nodes, including the ones that 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.*

Consider the modification of the algorithms DET and RAND that upon receiving an insertion request, inserts the new node into an arbitrary feasible position (respecting the dependencies). Upon receiving an access request, the behavior of the algorithms remain unchanged. Upon receiving a deletion request, the algorithms delete the node. When RAND inserts a node, it assigns a bit to it uniformly at random from $\{0, 1\}$.

▶ **Theorem 4.3.** *For a sequence σ consisting of accesses, insertions and deletions, under Assumptions 4.1 and 4.2, the algorithms DET and RAND with insertions and deletions are strictly 4-competitive.*

With Assumption 4.2, it is always possible to insert a node without rearranging the list — a position that satisfies all constraints already exists in every configuration.

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

Consider any insertion request. Due to the transitivity assumption (Assumption 4.2) for the DAG, we know that there exists a position in any list where the incoming node can be inserted without node rearrangements. Recall that by our model assumption (Assumption 4.1), upon receiving an insertion request, any algorithm needs to check all the nodes in the list for possible dependencies, and then it can insert the new node in a position of its choice for free. Thus, for an insertion request, both DET and OPT pay exactly $C_{\text{OPT}}(t) = C_{\text{DET}}(t) = n$. Note that DET and OPT might place the node at different positions, thus we account for at most n inversions created, and consequently for the change in potential we have $\Delta\Phi(t) \leq 2n$. This gives us $C_{\text{DET}}(t) + \Delta\Phi(t) \leq 3n \leq 3 \cdot C_{\text{OPT}}(t)$.

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

Similarly to the proof of Theorem 2.6, the amortized cost for an access request is $C_{\text{DET}}(t) + \Delta\Phi(t) \leq 4 \cdot C_{\text{OPT}}(t)$. We sum up the amortized bounds for all requests of the sequence σ, of all types (access, insertion, deletion), and we conclude that DET is 4-competitive.

Second, consider the algorithm RAND. The argument is similar to the proof for DET. Fix a sequence of requests σ of length m. We use a potential function Φ where, we distinguish between two types of inversions; we say that an inversion of the ordered pair (y, σ_t) is *type i* if $\mathbf{b}(\sigma_t) = i$ for $i \in \{0, 1\}$. Precisely, we use the potential function

$$\Phi = \frac{5}{2} \cdot I_0 + \frac{7}{2} \cdot I_1,$$

where I_0 is the number of inversions of type 0 and I_1 is the number of inversions of type 1.

Consider any insertion request. Due to the transitivity assumption (Assumption 4.2) for the DAG, we know that there exists a position in a list where the incoming node can be inserted without node rearrangements. Recall that by model assumption (Assumption 4.1), upon receiving an insertion request, any algorithm needs to check all the nodes in the list for possible dependencies, and then it inserts the node in a position of its choice for free. Also, for insertion in RAND, we assign a bit from $\{0, 1\}$ to a new node uniformly at random. Thus, for an insertion request, both RAND and OPT pay exactly $C_{\text{OPT}}(t) = C_{\text{RAND}}(t) = n$. Note that RAND and OPT may have placed the node at different positions, thus we account for at most n inversions created. By Observation 3.1, the bits of the nodes are independent

and equally likely to be 0 or 1, and we note that this property is true with insertions as well. Thus, we bound the expected change in potential by $E[\Delta\Phi(t)] \leq (\frac{1}{2} \cdot \frac{5}{2} + \frac{1}{2} \cdot \frac{7}{2}) \cdot n = 3n$. This gives us $E[C_{\text{RAND}}(t) + \Delta\Phi(t)] \leq 4n \leq 4 \cdot C_{\text{OPT}}(t)$.

In case of a deletion request, exactly ℓ inversions are removed, and since $C_{\text{OPT}}(t) = k$ and $C_{\text{DET}}(t) = k + \ell$, we have $E[C_{\text{RAND}}(t) + \Delta\Phi(t)] \leq (k + \ell) - \frac{5}{2} \cdot \ell \leq k \leq 1 \cdot C_{\text{OPT}}(t)$, where the first inequality holds since the deletion of type 0 inversion decreases the potential the least. Finally, we sum up the amortized bounds for all requests of the sequence σ, of all types (access, insertion, deletion), and we conclude that RAND is 4-competitive. ◀

Packet classification and transitivity. Finally, we note that the packet classification rule dependencies may not have the transitivity property (for an example, see Table 1). To deal with this issue without sacrificing competitiveness, either an amortized analysis is needed, or a solution that is related to the nature of the rules, such as rule preprocessing.

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

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

5 Conclusions and Future Directions

We introduced a generalization of the classic list access problem with dependencies, and showed that this generalization still admits constant-competitive online algorithms. We see two interesting avenues for future research. On the theoretical front, it would be interesting to have tight bounds on the competitive ratio, in both the deterministic and the randomized setting, and to analyze insertions and deletions without the transitivity assumption. On the applied front, it would be interesting to engineer our algorithms towards enhancing existing software-based packet classifiers.

Acknowledgements. Research supported by the Austrian Science Fund (FWF) and the Hungarian National Research, Development and Innovation Office NKFIH, I 5025-N, 2020-2024.

References

1 Subrata Acharya, Bryan N. Mills, Mehmud Abliz, Taieb Znati, Jia Wang, Zihui Ge, and Albert G. Greenberg. OPTWALL: A hierarchical traffic-aware firewall. In *Proceedings of the Network and Distributed System Security Symposium, NDSS*, 2007.

2 Susanne Albers. Online algorithms: a survey. *Math. Program.*, 97(1-2):3–26, 2003.

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

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

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

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

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

8 David Eppstein and S. Muthukrishnan. Internet packet filter management and rectangle geometry. In *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, SODA*, pages 827–835, 2001.

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

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

11 Hazem H. Hamed and Ehab Al-Shaer. Dynamic rule-ordering optimization for high-speed firewall filtering. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security, ASIACCS*, pages 332–342, 2006.

12 Shahin Kamali, Susana Ladra, Alejandro López-Ortiz, and Diego Seco. Context-based algorithms for the list-update problem under alternative cost models. In *Proceedings of the Data Compression Conference, DCC*, pages 361–370, 2013.

13 Shahin Kamali and Alejandro López-Ortiz. A survey of algorithms and models for list update. volume 8066 of *Lecture Notes in Computer Science*, pages 251–266. Springer, 2013.

14 Alejandro López-Ortiz, Marc P. Renault, and Adi Rosén. Paid exchanges are worth the price. *Theor. Comput. Sci.*, 824-825:1–10, 2020.

15 Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Mauricio Vásquez Bernal, Yunsong Lu, and Jianwen Pi. Securing Linux with a faster and scalable iptables. *SIGCOMM Comput. Commun. Rev.*, 49(3):2–17, 2019.

16 J. Ian Munro. On the competitiveness of linear search. In *Proceedings of the European Symposium, ESA*, volume 1879, pages 338–345, 2000.

17 ONF. Openflow reference release. https://github.com/mininet/openflow, 2013.

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

19 Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, February 1985.

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