Randomized Algorithms for Online List Access with Precedence Constraints

Author: Please provide author information

Abstract

We consider a generalization of the online list access problem with constraints on the relative order of some pairs of nodes in the list. The task is to devise an online algorithm that adjusts a linked list of n nodes serving a sequence of node access requests σ. The cost of accessing a node v corresponds to v's distance from the head of the list. After serving a request, the algorithm may rearrange the nodes via transpositions; each transposition costs d, where d is a parameter. The precedence constraints are given at the beginning, and for each constraint (u, v), the node u must be in front of v in every configuration of the list.

Our main contribution is the design and analysis of a family of randomized online algorithms for this problem. In particular, we present a $\sqrt{7} \approx 2.64$-competitive randomized algorithm against the oblivious adversary for online list access with precedence constraints. Our algorithms build on the Markov-Move-to-Front family of algorithms for the classic online list access problem. Generalizing these algorithms to the setting with precedence constraints requires new ideas. To this end, in our analysis we partition the inversions into *hidden inversions* and *visible inversions*, to capture the positional relation of a pair of nodes to their precedence constraints.

Furthermore, we present an optimal offline algorithm for list access with precedence constraints in the P^d model and show that its running time improves as the list becomes more constrained.

2012 ACM Subject Classification Theory of computation → Online algorithms

Keywords and phrases Online algorithms, randomized algorithms, list access

1 Introduction

This paper considers a natural generalization of the online list access problem [22], called *online list access with precedence constraints* [18]. In this problem, we manage a set of items arranged in a linked list. The nodes of the list must obey a partial order: if we have a precedence constraint (u, v), u must appear before v in any configuration of the list. We are given a sequence of *access* requests to the nodes of our list. Upon receiving an access request to a node v, an algorithm searches linearly through the list: starting from the head, it traverses nodes until it finds v. The access cost is proportional to the position of the node. After serving a request, the nodes of the list can be reordered, and for each transposition of (neighboring) nodes, the algorithm pays d, an integer parameter given at the beginning. If there are no precedence constraints, the problem is equivalent to the classic online list access.

We often refer to precedence constraints as *dependencies* between nodes. In this view, we are given a directed acyclic graph G (the *dependency graph*) inducing a partial order among the nodes that is equivalent to the reachability relation in G. If there exists an edge (u, v) in G (a node v depends on a node u), then in every configuration v must be in front of u.

The model finds applications in processing pipelines and assembly lines, where *some* stages can be executed in an arbitrary order, and the other should stay in a fixed order. In the context of communication networks, our model can be used in packet classification with the classification rules arranged in a linked list; the rules whose domains overlap need to be examined in a fixed order. For an overview of the approach, we refer to [18].

We are interested in online algorithms that achieve a low (strict) *competitive ratio*: ideally, the cost of the online algorithm should be close to the cost of an optimal offline algorithm that knows σ ahead of time. Specifically, the competitive ratio is defined as the online algorithm's cost divided by the offline algorithm's cost. For an overview of competitive analysis, we refer to Appendix B.

1.1 Contributions

We make the following technical contributions for list access with precedence constraints.

Our main contribution is designing and analyzing a family of randomized online algorithms for list access with precedence constraints. Our family of algorithms includes an algorithm that achieves a competitive ratio of $\sqrt{7} \approx 2.64$ against the oblivious adversary when $d = 1$, and the ratio improves as d grows. This ratio matches the competitiveness of the best currently known RANDOM-RESET algorithm [4, 21] from the classic list access problem. Although our algorithms build on foundations of Markov algorithms [10] for the classic online list access, the analysis must be strengthened, not to deteriorate the competitive ratios. To this end, we characterize a special type of inversions, called *hidden inversions*, to use in the potential function analysis framework of Sleator and Tarjan [22].

Furthermore, we design and analyze an optimal offline algorithm for list access with precedence constraints in the P^d model and show that its running time improves with more dependencies.

1.2 Related Work

The online list access problem has been studied for decades [15, 22], and remains an active field of research [2]. Its most common application models dictionaries organized in linked lists, with further applications in data compression [7].

List access cost models. The cost models for list access have evolved throughout the years. The first and probably the most well-known one is the *free exchange* model (alternatively known as *standard cost model*), where moving the accessed node to the front of the list is free. An extension of this model, called *generalized cost model*, assumes that the access cost can be any function of the distance of accessed node [22]. Another variant of the cost model is the P^d model [21], which keeps the access cost equals to 1, but assumes that the cost of each transposition is increased to $d \geq 1$. A subclass of P^d model with $d = 1$ is called *paid exchange model*. In this paper, we focus on the general P^d model. Some papers studied a model with batch rearrangements with linear cost [13, 16].

Deterministic algorithms for online list access. In the paid exchange model, the best known deterministic algorithm is Move-To-Front (MTF) by Sleator and Tarjan [22] which is 4-competitive. The survey [13] suggests that the deterministic algorithm Move-To-Front-Every-Other-Access can be shown to be 3-competitive. Another important algorithm is TIMESTAMP [1]. It is known that no deterministic algorithm can be better than 3-competitive; this lower bound is due to Reingold et al. [21].

Randomized algorithms for online list access. In the randomized setting, the best known algorithm in the paid exchange model is RANDOM-RESET [21] that is $\sqrt{7} \approx 2.64$-competitive against the oblivious adversary, but it was suggested that randomly mixing RANDOM-RESET strategies for different values of the counter improves the competitive ratio [4]. The best algorithm for large d was given by Albers et al.: $(5+\sqrt{17}) \approx 2.2808$-competitive as d grows approaches infinity [2]. The best known lower bound in the paid exchange model against the oblivious adversary is 1.8654 [2]. The algorithms COUNTER, and RANDOM-RESET are members of the Markov family of algorithms for list access [10].

Offline algorithms for list access. An optimal solution for the offline variant of list access problems is NP-hard to compute [5]. The problem was first studied by Reingold and Westbrook [20], where they developed an algorithm with a running time that contains a factorial term in the number of elements. Their algorithm used the subset transfer method. An improvement of the subset transfer method has been suggested by Divakaran [8] in a non-peer-reviewed manuscript, which may be investigated in future work.

List access with precedence constraints. The closest work to ours is by Pacut et al. [18], who initiated the study of list access with precedence constraints and presented a 4-competitive deterministic algorithm, together with empirical studies in the context of input locality.

1.3 Organization

The remainder of this paper is organized as follows. First, in Section 2 we recall the online list access problem on the precedence constraint setting and the cost model used. Then, in Section 3 we recall the algorithm Move-Recursively-Forward [18] and concepts related to its design, upon which we build our randomized algorithms. We state the main contributions of this paper in Section 4, where we present a whole family of randomized (Markov-based) algorithms. Then, in Section 5 we shift our attention to offline algorithms and design an optimal algorithm for list access with precedence constraints. Finally, we conclude our work in Section 6.

2 Model

2.1 Online List Access with Precedence Constraints

We recall the model for online list access with precedence constraints [18].

The list and the precedence constraints. We are given a linked list consisting of n nodes, and a set of constraints for the nodes' relative order in the list. The constraints are given in the form of a directed acyclic graph (DAG) G, called a *dependency graph*. We say that a node u is a *dependency* of a node v if there exists a directed edge (v, u) in G. The nodes must comply with the order induced by G: for each node, all its dependency nodes must precede it in any configuration of the list.

Access requests and their cost. We are given a request sequence σ of accesses to nodes of the list, arriving over time (indexed by t) in an online fashion. Upon receiving a request σ_t, an algorithm searches the list linearly from the head for the requested node. For the access, the algorithm pays the cost equal to the position of the node in the list. The position of a node is its distance to the head of the list. The position of the first node in the list is 1.

Rearrangement cost and the P^d model. After serving the request, the algorithm may rearrange the list by transposing neighboring nodes while complying with the precedence constraints encoded by G. In this paper, we analyze the algorithms in the P^d model, introduced by Reingold and Westbrook [21]. In the P^d model, the rearrangement cost is scaled by a positive integer d, a parameter. Immediately after serving the request, the algorithm may perform any number of *paid exchanges*, at the cost of d per each transposition of neighboring nodes, but the dependencies must be respected.

The goal of the online algorithm is to minimize the total cost of access and node rearrangements. We study all the algorithms in this paper in the P^d model.

3 Algorithmic Building Blocks

We build our solutions based on some concepts from previous works. In [18] a deterministic algorithm achieving 4-competitiveness on the precedence constrained setting and paid model was introduced with the name *Move-Recursively-Forward (MRF)*. It is a natural generalization of the well-known *Move-To-Front (MTF)* algorithm [22] that also achieves 4-competitiveness in the P^1 model. Instead of moving the requested node to the front of the list as MTF does, MRF moves multiple nodes but amounting to the same number of transpositions that MTF uses (also identical to the node's position in the list), thus essentially incurring the same cost.

One of the key concepts that we use in designing algorithms in this paper is a direct dependency of a node u, a node that is both the dependency of u, and is positioned in the list, so it would be encountered first if u starts moving towards the front of the list. Direct dependency limits the rearrangements of a single node: to move a node closer to the front of the list, the direct nodes must be moved forward too.

▶ **Definition 3.1** (direct dependency). *For a node u, we say a node v is u's direct dependency if and only if v is the precedence constraint of u (there exists edge $(u, v) \in G$) which is located at the furthest position in the list among all u's dependencies.*

Now, we revisit how the concept of direct dependencies gave rise to the algorithm MRF [18]. Figure 1 assists in the explanation of how the MRF uses the dependency chain to rearrange nodes after access. Consider an access request σ_t at time t addressed at a node y. Say that for the current list configuration, z is the direct dependency of y. First, the algorithm services the access and pays its incurred cost. Then, it proceeds to rearrange the list by swapping the position of y with its neighbors towards the head of the list until it reaches z. Note that y can not be swapped forward any further without incurring an infeasible transposition. Instead, the algorithm simply leaves y at its reached position and starts swapping z position with its own neighbors towards the head. Once the direct dependency of z is reached, the algorithm repeats the procedure recursively. When the algorithm encounters a node without dependencies, it moves the node to the front of the list and ends the procedure. We refer to the nodes that MRF moves forward after the request as *sequence of direct dependencies*, defined formally as follows.

▶ **Definition 3.2** (sequence of direct dependencies). *For a node u the* sequence of direct dependencies *is a sequence of nodes ending with u, where the node at position i is a direct dependency of the node at position $i+1$. The sequence begins with a node without dependencies.*

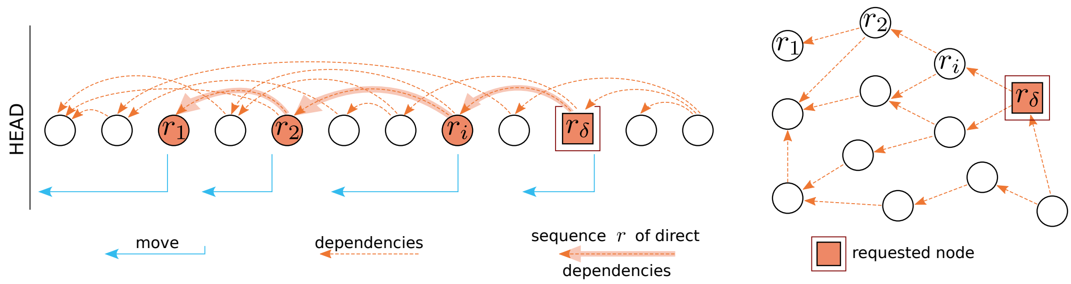

Figure 1 An example of a sequence of direct dependencies for a node r_δ: $\{r_1, r_2, \ldots, r_\delta\}$. Upon a request to the node r_δ, the algorithm Move-Recursively-Forward moves every node r_i from the sequence just behind its direct dependency (see the blue arrows below the list). The accessed node is depicted as a square orange node, and the nodes from the direct dependency chain are depicted with circular orange nodes. At the left, we depict the precedence constraints for the nodes in the list, as well as the sequence of direct dependencies $(r_1, \ldots, r_\delta)$ of the requested node and the moves (transpositions) to be performed by MRF. At the right, we depict the DAG inducting the precedence constraints between the nodes.

We can find the sequence of direct dependencies by recursively following the direct dependencies, starting from r_δ, until encountering the first node that does not have dependencies.

The algorithm Move-Recursively-Forward was analyzed [18] using a potential function, defined in terms of inversions. The inversion is the central concept in the analysis of the presented algorithms in this paper.

▶ **Definition 3.3** (Inversion). *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 denote the set of all inversions between lists L_1 and L_2 by $\mathsf{inv}(L_1, L_2)$. In the potential function analysis of our algorithms, we always consider inversions between ALG's and OPT's list, i.e., inversions are chosen from the set $\mathsf{inv}(\text{ALG}, \text{OPT})$.

4 A Family of Randomized Algorithms

In this section, we first present the main result of this paper: a family of randomized algorithms called Markov Move Recursively Forward (MMRF). We show that the competitive ratio of MMRF in our model with precedence constraints matches the competitive ratio of Markov-Move-to-Front [10] in the model without precedence constraints under potential function analysis. The novelty of our analysis lies in the concept of hidden inversions and a potential function based on hidden inversions. We demonstrate how our algorithm results in a 2.64-competitive algorithm which is also the best-known ratio in the classic model without precedence constraints. We note that generalizing the result poses an algorithmic challenge (see Section C), and new analytical ideas are needed.

4.1 MMRF: Markov-Move-Recursive-Forward

We present MMRF, a family of randomized algorithms for the list access problem with precedence constraints. Each algorithm in the family is characterized by a Markov chain, which is initialized for every item 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 [10], 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 1 The algorithm Markov-Move-Recursively-Forward.

```
Initialization : Each node's Markov chain is initialized according to the stationary
                 distribution π.
Input : An access request to node σ_t
Access σ_t
Run the procedure MMRF(σ_t)
procedure MMRF(y):
    Let z be the direct dependency of y
    if state(y) is 0 then
        Move node y to pos(z) + 1          ▷ Move y behind its direct dependency
    end if
    Transition to state j with probability p_{state(y)→j}
    if pos(z) ≠ 0 then                     ▷ If a dependency is found
        Run the procedure MMRF(z)          ▷ Recursion
    end if
    Exit
```

Algorithm overview. Our algorithm MMRF relies on the MMRF procedure to handle dependencies. Each node in the list is associated with a Markov chain (defined above), and the initialization is done according to the stationary distribution π. We denote the state of a node y by $state(y)$. Upon request to an item y, the node y is moved forward in the list, however only behind its *direct dependency* not always to the head of the list. Upon receiving a request to node σ_t, we run the procedure MMRF(σ_t). The procedure MMRF(y) computes z, the direct dependency of y. Our algorithm then executes MMRF(z) which triggers recursion until no direct dependency is found, i.e., z is the head of the list.

Algorithm MMRF definition. Let $\mathsf{pos}(z)$ 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 direct dependency node z, treated as the virtual head of the list. The state of y then transitions to a state j with probability $p_{state(y)\to j}$ and the procedure recursively calls MMRF(z) if $\mathsf{pos}(z) \neq 0$. We present the pseudocode of MMRF in Algorithm 1.

4.2 Types of Inversions

We introduce a concept of *hidden inversions*, a type of inversions defined by both the counter value of the nodes and the relative position of the nodes with respect to their dependencies. Hidden inversions limit the effect of inversions changing type: each hidden inversion contributes a *neutral* value to the potential function (independent of the state).

We classify inversions into two types: *hidden* and *visible*. The intuition behind classifying inversions into hidden and visible is that the movement of a node can only destroy visible inversions. Consider the example in Figure 2. Hidden inversions with respect to the node r_2 are those that cannot be destroyed; since r_2 can only move forward to a position behind its direct dependency r_1. The movement of r_2 can however destroy all visible inversions, i.e., the inversions between r_2 and r_1.

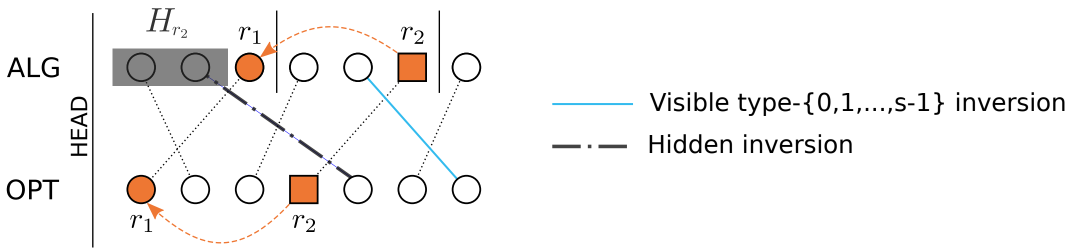

Figure 2 Consider the node r_2 and its direct dependency r_1. The region from the head of the list until r_1 is the *hidden* region H_{r_2} with respect to r_2. Any inversion of the form (u, r_2) is classified as *(i) hidden inversion* if u lies in the hidden region of r_2, otherwise *(ii) visible inversion* if u lies between r_1 and r_2. For intuition, notice that the movement of r_2 can only destroy visible inversions and cannot destroy any hidden inversions. This is due to the precedence constraints i.e., r_2 cannot be moved ahead of its dependency r_1.

▶ **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 direct dependency 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 v's direct dependency 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.*

4.3 Definitions Related to a Request

We recall the notations of sets and sequence of nodes relevant to our analysis.

Nodes r_j. Consider a single request to a node σ_t and the sequence of direct dependencies computed recursively by MMRF procedure at time t. Let r be the sequence of the nodes that the algorithm executes on, ordered by increasing distance to the head. Let δ be the length of r. We emphasize that r contains the requested node at the last position, $\sigma_t = r_\delta$.

Values k and ℓ. To compare the cost of ALG and OPT, we define values k and ℓ related to the number of nodes in front of the requested node σ_t in ALG's and OPT's list. Precisely, let k be the number of nodes before σ_t in both ALG's and OPT's lists, and let ℓ be the number of nodes before σ_t in ALG'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 MMRF. Hence, we generalize the notion of k and ℓ to sets of elements related to positions of individual nodes r_j in ALG's and OPT's lists. Precisely, let K_j be the set of elements before r_j in both ALG's and OPT's lists for $j \in [1, \delta]$, and let L_j be the set of elements before r_j in ALG's list but after r_j in OPT's list. We note that these sets are generalizations of k and ℓ: for the accessed node r_δ we have $k = |K_\delta|$ and $\ell = |L_\delta|$.

Sets S_j. The sets of nodes between the nodes r in ALG's list are crucial to the analysis. Intuitively, the node r_i's movement is confined to all the nodes from the set S_i. Let S_1 be the elements between the head of ALG's list and r_1 (included). For $j \in [2, \delta]$, let S_j be the set of elements between r_j and r_{j-1} (with r_{j-1} excluded) in ALG's list.

4.4 The Analysis of MMRF

Our analysis in this section is based on amortized cost analysis in the P^d model. Hereafter in this section, we refer to MMRF as ALG and an optimal algorithm as OPT. We analyze the competitiveness of ALG against an *oblivious* adversary.

First, we discuss the potential function used to relate the cost ALG and OPT. The potential function is designed around the concept of hidden inversions (cf. Section 4.2). Second, we claim that the state of each node in ALG remains at the stationary distribution and is independent of other nodes in the list. 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 $T = \sum_{i=0}^{s-1} \pi_i \cdot h_i$.

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.

▶ **Observation 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.*

4.4.1 How Does Node Movement Influence 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 r_j that moves forward in a call of procedure MMRF. The move of r_j may cause any of the following:

- 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. In the following, we first prove that the expected change in potential due to any inversions that change 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.

▶ **Lemma 4.4.** *In moving a node r_j forward in the list, the expected change in potential due to inversion type changes from hidden to a visible type is zero.*

Proof. Consider any inversion (u, v) that changes type from hidden to any visible type-i. Let (u, r_j, v) be the order of nodes in the list, where r_j is the direct dependency of v. Note that the presence of r_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, r_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 r_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 4.5.** *In moving a node r_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 (r_j, v) that changes type from a visible type to hidden. Let u be the direct dependency of v such that (u, r_j, v) is the order of nodes in the list. Note that the presence of r_j between u and v is required such that r_j lies in the visible region of v. For such an inversion to change from visible to hidden, r_j must move ahead of u so that r_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 r_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 $\sum_{i=0}^{s-1} \pi_i = 1$. ◀

▶ **Theorem 4.6.** *The expected change in the potential due to inversion type changes from and to hidden inversions is zero.*

Proof. The claim follows from Lemma 4.4 and Lemma 4.5, by summing over all inversions that change their type to hidden and all inversions that change their type from hidden. ◀

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 4.7.** *The change in potential due to a created hidden inversion equals the expected change in potential due to a created visible inversion.*

Proof. For each created hidden inversion, the change in potential $\Delta\Phi = d + T$. 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 $T = \sum_{i=0}^{s-1} \pi_i \cdot h_i$. ◀

4.4.2 Amortized Cost of a Request

Consider an irreducible Markov chain M with s states, stationary distribution $\pi = (\pi_0, \pi_1, ..., \pi_{s-1})$ and transition probabilities $P = (p_{i \to j})$. The goal of our analysis in this section is to determine the upper bound of competitive ratio of MMRF algorithm that operates on M.

Let $a(t)$ be the amortized cost of ALG in serving a request t and the cost incurred by OPT be $C_{\mathrm{OPT}}(t)$. 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 r_δ be the requested item. Since the movement of the items r_j (see Algorithm 1) is independent for all $1 \leq j \leq \delta$, 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 r_j. This accounts for the access cost, paid transpositions and all the changes in potential.

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

4.5 Bounding the Competitive Ratio

Finally, we combine the observations made so far to bound the competitive ration of MMRF.

▶ **Theorem 4.8.** *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.*

Using hidden inversions and the results of Section 4.4.1, analysis of the amortized cost of request is straight-forward. We defer the proof to Appendix D.1.2 and sketch it next.

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 $C = \max\{1 + \pi_0 \cdot (2d + T), 1 + \frac{T}{d}\}$. 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. First, we analyze each node r_j separately and obtain the expected amortized reconfiguration cost due to the item r_j. Then we sum over all nodes r_j on which MMRF is executed and add the access cost of $k + l + 1$ to obtain the total amortized cost of the request. In each step, we analyze all changes in inversions i.e., created, destroyed, and inversion type changes.

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

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 4.7, irrespective of whether the created inversion is hidden or visible, the expected change in potential is $d + T = C_{OPT} \cdot (1 + \frac{T}{d})$.

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. The full formal proof appears in Appendix D.1.2.

We summarize the competitive ratios of some randomized algorithms in the MMRF family. We consider the COUNTER and RANDOM-RESET class of algorithms that were first proposed by Reingold et al. [21] for the classic list access problem without precedence constraints. Both COUNTER and RANDOM-RESET can be represented as a Markov chain. Given a Markov chain M that corresponds to COUNTER and RANDOM-RESET class, with s states, Table 1 (in the appendix) summarizes the competitive ratio of MMRF algorithms in the model with precedence constraints. Interestingly, the competitive ratios in our model match the ratios in the classic model. Notably, RANDOM-RESET remains the best even with precedence constraints for $d = 1$ and has a ratio of 2.64.

5 An Offline Algorithm

This section introduces an optimal offline algorithm for list access with precedence constraints in the P^d model for any positive integer d. Our method generalizes the dynamic algorithm introduced by Reingold and Westbrook [20] and benefits from the permutation generation technique proposed by Ono and Nakano [17].

The running time of our algorithm is proportional to the number of possible node permutations (respecting dependencies) and the length of the input sequence, providing a significant improvement on instances with dense dependency structures.

5.1 Subset Transfer

A building block of our offline algorithm is the subset transfer operation which was first suggested by [20]. This operation lets us build our algorithm based on a restricted set of operations among all possibilities, reducing the running time of our algorithm significantly.

▶ **Definition 5.1.** *A* subset transfer *is a set of paid exchanges taking place between serving the request σ_{t-1} and σ_t, consists of moving only a subset of nodes preceding σ_t in the list to the right after it.*

The following lemma enables us to prove Theorem 5.3 in lists with precedence constraints. We show that the cost of transforming one list to another (using paid exchanges) is exactly d times the number of *inversions* between the two lists.

The proof of the lemma is based on induction and looking at properties of the first node one of the lists. We defer the details of the proof to Appendix D.2.

▶ **Lemma 5.2.** *The cost of reconfiguring list L_2 to list L_1 (that share the same dependency graph) using paid exchanges is d times the number of inversions between L_1 and L_2.*

In the next theorem, we show that subset transfer operations are sufficient for an optimal offline algorithm.

▶ **Theorem 5.3.** *There exists an optimal offline algorithm for list access with precedence constraints that only performs subset transfers.*

The schema of the proof of Theorem 5.3 is due to Reingold and Westbrook [20], and we extend it to the general case of lists with precedence constraints, benefiting from Lemma 5.2. The idea of the proof is transforming an optimal sequence of paid exchanges to another sequence of paid exchanges that only uses subset transfer, without any additional cost. The details of the proof are in Appendix D.2.

5.2 Design of the Algorithm

We now detail our dynamic algorithm and show how the cost for an access request can be updated from the costs of the previous request. We also discuss the permutation generation method which is required for the algorithm. In the end, we formulate the running time and show how the dynamic algorithm can be implemented with improved space complexity.

Details of the dynamic algorithm. Consider $C_{OFF}(L,t)$ to be the minimum cost of serving access requests up to time t and ending up with the list L. Assume $\mathsf{pos}_X(\sigma_t)$ to be the position of accessed node at time t in a list X. We fill the dynamic table of our algorithm by finding a list L' that minimizes the cost of serving requests up to the request at time $t-1$, plus the cost of accessing the node at time t (which is equal to the position of σ_t in the list L'), and the cost of transforming L' to L, which we know based on Lemma 5.2 is d times the number of inversions between L and L'. In summary, we calculate the cost $C_{OFF}(L,t)$ as follows. The optimal cost for serving all requests is the minimum of $C_{Off}(L,m)$ over all possible lists respecting dependencies.

$$C_{OFF}(L,t) = min_{L'}[C_{OFF}(L',t-1) + \mathsf{pos}_{L'}(\sigma_t) + d \cdot \mathsf{inv}(L',L)].$$

However, we do not need to check all possible lists to find the optimal one. Based on theorem 5.3, it is sufficient to only check lists L' that can be transformed to L using only a subset transfer. Finding those lists is based on a procedure that we call *reverse subset transfer*. Concretely, the procedure $Rev(L,t)$ constructs all lists L' that can be transformed to L using subset transfers.

We describe the procedure $Rev(L,t)$ in terms of a recursive subroutine $Rev(L,1,\mathsf{pos}_L(\sigma_t))$. The subroutine $Rev(L,i,j)$ generates all lists that can be transformed into L using subset transfer, such that the requested node is placed at the position j in the list L, and the subset transfer only involves elements from the position j in the list L' or afterwards. To do so, we consider the node at position $j+1$ and move it one step closer to the head of the list(if this movement respects dependencies). Assume that we moved the node to the position k, we then invoke the recursive call $Rev(L, k+1, j+1)$ to possibly move the next nodes in L.

Generating permutations respecting precedence constraints. Our algorithm relies on generating all permutations of nodes respecting precedence constraints. Algorithms for another interpretation of this problem, namely generating all topological sorts, have been known for quite a while [12, 14, 19]. However, most of the old approaches create each permutation in $O(n)$. We benefit from the algorithm proposed by Ono and Nakano [17]. The running time of their algorithm only differs within a constant factor to the size of all possible permutations.

The running time of the offline algorithm. To find the optimal solution, it suffices to fill all the entries of our dynamic table. Therefore, the running time of our algorithm is equal to the number of entries of the dynamic table times the time required to fill each of them.

The number of entries of our dynamic table is $m \cdot |Perm|$, in which $|Perm|$ shows the number of possible permutations. The time required for each entry of our dynamic table equals the time required for running the reverse subset transfer procedure. Each step of this procedure requires $O(1)$ time, and there are at most 2^n choices of nodes to be considered. Hence, the sum of the running time of the procedure for a list and a request is $O(2^n)$. Summing the cost over all possible lists gives us the total running time of $O(2^n \cdot m \cdot |perm|)$. We can see that the running time of the optimal offline algorithm improves as the number of permutations reduces, which happens as more precedence relations are introduced.

Optimizing the required space. The required space for a trivial implementation has both the number of permutations *and* the number of requests as a factor in it. However, as we only need costs from the previous request to find the cost of each access request, maintaining a table with the size of twice the number of permutations is sufficient.

6 Conclusions and Future Directions

We successfully transferred a family of randomized algorithms for online list access to online list access with precedence constraints without deteriorating the competitiveness. Moreover, we showed how an optimal offline algorithm could be designed for the setting with precedence constraints. Our results suggest that introducing precedence constraints makes the problem *no harder* than online list access.

Although we reach the competitiveness of the classic list access, several avenues of research remain open. The transferred family of algorithms does not include the TIMESTAMP algorithm [1], and an interesting question arises if this algorithm can be adapted to the setting with precedence constraints with unchanged competitive ratio. The algorithms for online list access problem improve with *locality of reference* [3], and experimental results for the case with precedence constraints [18] confirm a similar trend, which may be explained analytically. For offline algorithms, an improvement of the subset transfer method has been suggested by Divakaran [8] in a non-peer-reviewed manuscript, and this direction may be investigated in future work, also in the context of precedence constraints.

References

1. Susanne Albers. Improved randomized on-line algorithms for the list update problem. *SIAM J. Comput.*, 27(3):682–693, 1998.
2. 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, 2020.
3. Susanne Albers and Sonja Lauer. On list update with locality of reference. *J. of Computer and System Sciences*, 82(5):627–653, 2016.
4. Susanne Albers and Michael Mitzenmacher. Revisiting the counter algorithms for list update. *Information processing letters*, 64(3):155–160, 1997.
5. Christoph Ambühl. Offline list update is np-hard. In *8th Annual European Symposium on Algorithms, ESA*, volume 1879, pages 42–51. Springer, 2000.
6. Allan Borodin and Ran El-Yaniv. *Online computation and competitive analysis.* cambridge university press, 2005.
7. Michael Burrows and David Wheeler. A block sorting lossless data compression algorithm. *Technical Report 124, Digital Equipment Corporation*, 1994.
8. Srikrishnan Divakaran. An optimal offline algorithm for list update. *CoRR*, abs/1404.7638, 2014.
9. Theodoulos Garefalakis. *A family of randomized algorithms for list accessing.* Thesis, 1997.
10. Theodoulos Garefalakis. A new family of randomized algorithms for list accessing. In *Proc. of European Symposium on Algorithms, ESA*, volume 1284, pages 200–216, 1997.
11. Mark Kac. On the notion of recurrence in discrete stochastic processes. *Bulletin of the American Mathematical Society*, 53(10):1002–1010, 1947.
12. Alan D. Kalvin and Yaakov L. Varol. On the generation of all topological sortings. *J. Algorithms*, 4(2):150–162, 1983.
13. 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 of *Lecture Notes in Computer Science*, pages 251–266. Springer, 2013.
14. Donald E. Knuth and Jayme Luiz Szwarcfiter. A structured program to generate all topological sorting arrangements. *Inf. Process. Lett.*, 2(6):153–157, 1974.
15. John McCabe. On serial files with relocatable records. *Operations Research*, 13(4):609–618, 1965.
16. J. Ian Munro. On the competitiveness of linear search. In *Proc. of the European Symposium, ESA*, volume 1879, pages 338–345, 2000.
17. Akimitsu Ono and Shin-Ichi Nakano. Constant time generation of linear extensions. In *Proc. of International Symposium on Fundamentals of Computation Theory, FCT*, volume 3623 of *Lecture Notes in Computer Science*, pages 445–453. Springer, 2005.
18. Maciej Pacut, Juan Vanerio, Vamsi Addanki, Arash Pourdamghani, Gabor Retvari, and Stefan Schmid. Self-adjusting packet classification, 2021. arXiv:2104.08949.
19. Gara Pruesse and Frank Ruskey. Generating linear extensions fast. *SIAM J. Comput.*, 23(2):373–386, 1994.
20. Nick Reingold and Jeffery R. Westbrook. Off-line algorithms for the list update problem. *Inf. Process. Lett.*, 60(2):75–80, 1996.
21. Nick Reingold, Jeffery R. Westbrook, and Daniel Dominic Sleator. Randomized competitive algorithms for the list update problem. *Algorithmica*, 11(1):15–32, 1994.
22. Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, February 1985.

A Lifting Classic List Access Algorithms to MMRF Family

The two classic algorithms for online list access that we discuss were first proposed by Reingold et al. [21], and extended to many other randomized algorithms [2, 4]. Here we detail the transformation for two of these algorithms, which are also considered by [10].

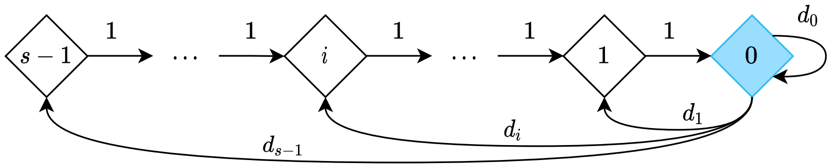

Figure 3 An example of the Markov chain representation of RANDOM-RESET(s, D). The diamonds represent states of the Markov chain, and the arrows are transitions between two states, indexed with the transition probability. The state 0 that initiates the movement in the list is shown in blue.

COUNTER with precedence constraints. 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 with precedence constraints. In the RANDOM-RESET(s, D) 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 D. Formally, RANDOM-RESET(s, D) is MMRF algorithm with the following transition probability (depicted in Figure 3).

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

	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 d in 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. [21], which shows the same ratios can be achieved even in lists with precedence constraints.

B Overview of Online Algorithms and Competitive Analysis

Online algorithms receive as input a sequence $\sigma = \sigma_1, \ldots, \sigma_t, \ldots, \sigma_m$ of requests one at a time without knowledge of the future requests. This means that at any given time t, the algorithm has no information about requests $\sigma_{t+\tau}\ \forall\tau > 0$. The algorithm must serve each request right after receiving it, at a certain cost that depends on the system state. However, they can take actions to minimize the total cost of serving the whole sequence, although said actions may themselves have a cost of their own that must be accounted for.

Deterministic and randomized algorithms. A fundamental classification of online algorithms lies in how predictable they are. Essentially, *deterministic algorithms* are those whose actions and state at any point in time t can be is perfectly well-known given the algorithm description and the (sub) sequence of requests up to time t.

On the other hand, *randomized algorithms* make use of at least one source of randomness. Consequently, even with perfect knowledge of the algorithm (including the involved probabilities distributions) and the current subsequence of requests, the available knowledge about its state and behavior remains probabilistic.

Competitiveness. Performance of online algorithms is typically evaluated with the competitive analysis [22]. Under this framework, the performance of an algorithm can be measured by comparing its cost with the cost of an optimal offline algorithm over all possible sequences. The goal is then to design online algorithms with worst-case guarantees against the optimal. Let $\mathrm{ALG}(\sigma)$, resp. $\mathrm{OPT}(\sigma)$, be the cost incurred by a deterministic online algorithm ALG, resp. by an optimal offline algorithm, for a given sequence of requests σ. In contrast to ALG, which learns the requests one at a time as it serves them, OPT has complete knowledge of the entire request sequence σ *ahead of time*.

In particular, ALG is said to be *strictly c-competitive* if for any input sequence σ it holds that

$$\mathrm{ALG}(\sigma) \leq c \cdot \mathrm{OPT}(\sigma).$$

The minimum c for which ALG is c-competitive is called the *competitive ratio* of ALG.

The concept can be naturally extended to randomized algorithms; we say that a randomized online algorithm RAND is *c-competitive* if

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

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. Notice that competitive ratios for a given problem may vary depending on the adversary's power; recall that different adversarial models have different knowledge about RAND while producing the offline benchmark solution OPT.

For an overview of the competitive analysis framework, we refer the reader to [6].

Adversaries. The goal of an adversary is to generate a request sequence that maximizes the competitive ratio of the algorithm. Under this assumption, there are several adversarial models that distinguish themselves by the amount of information they have about the algorithm. The key distinction is whether the adversary knows the outcome of the random choices made by the algorithm on past requests.

In this paper, we design algorithms against the *oblivious offline 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 an extensive overview of adversary types, we refer to [6].

C Challenges in Randomizing MRF

Using ideas from MRF to design a randomized algorithm achieving a better competitive ratio turns out to be non-obvious. This is due to the insufficiency of analysis techniques from the classic list update problem, and new ideas for the potential function analysis are needed. For instance, distinguishing inversions based only on their type (the current state of the node in the back) cannot express any information concerning precedence constraints.

To emphasize the problem, we describe a naive adaption of the well-known BIT algorithm [21] to the model with precedence constraints: Initialize the list with 0 or 1 bit counters uniformly at random; upon request to a node, execute Move-Recursively-Forward if the item's bit value is 0; flip the bit on every request. This strategy leads to a competitive ratio no better than 3 using existing potential function analysis; extending counters beyond two bits does not help.

To better understand the problem, consider the following aspects of BIT's analysis in the setting without precedence constraints. In the classic model without precedence constraints, the Move-to-Front action in BIT destroys all inversions w.r.t the requested item. In contrast, in our model with precedence constraints, moving an item behind its direct dependency destroys only the inversions between the two items. All other inversions w.r.t the moving item change their type, which leads to the competitive ratio of 3, which does not reach the competitiveness of BIT in the classic list access (2.75-competitive).

To address this issue, we must limit the influence of changing type in inversions due to nodes' bits flipping. To this end, we introduce the concept of *hidden inversions*, a type of inversions defined by both the counter value of the nodes and the relative position of the nodes with respect to their dependencies. We elaborate in the next section.

D Omitted Proofs

D.1 Proofs from Section 4

▶ **Theorem 4.8.** *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.*

Before analyzing the competitive ratio of MMRF, we first state and prove the results required to obtain the competitive ratio. We present the proof of Theorem 4.8 at the end of this section.

D.1.1 Amortized Cost of a Request

We first state an important property of the Markov chain, which plays a crucial role in our analysis.

▶ **Lemma D.1.** *Given a Markov Chain with s states, stationary distribution π, transition probabilities $(p_{i\to j})$ and the hitting time h_i from state i to 0, the following equality holds:* $\sum_{i=1}^{s-1}\sum_{k=0}^{s-1} \pi_i \cdot p_{i\to k} \cdot (h_k - h_i) = 0.$

Proof.

$$
\begin{aligned}
\sum_{i=1}^{s-1}\sum_{k=0}^{s-1} \pi_i \cdot p_{i\to k} \cdot (h_k - h_i) &= \sum_{i=1}^{s-1} \pi_i \left(\sum_{k=0}^{s-1} p_{i\to k} \cdot h_k - \sum_{k=0}^{s-1} p_{i\to k} \cdot h_i \right) \\
&= \sum_{i=1}^{s-1} \pi_i \left(\left(\sum_{k=0}^{s-1} p_{i\to k} \cdot h_k \right) - (h_i) \right) \\
&= -\sum_{i=1}^{s-1} \pi_i \cdot h_i + \sum_{i=1}^{s-1}\sum_{k=0}^{s-1} \pi_i \cdot p_{i\to k} \cdot h_k \\
&= -\sum_{i=1}^{s-1} \pi_i \cdot h_i + \sum_{k=0}^{s-1} h_k \sum_{i=1}^{s-1} \pi_i \cdot p_{i\to k} \\
&= -\sum_{i=1}^{s-1} \pi_i \cdot h_i + \sum_{k=0}^{s-1} h_k (\pi_k - \pi_0 \cdot p_{0\to k}) \\
&= -(T - \pi_0 \cdot h_0) + T - \pi_0 \cdot \sum_{k=0}^{s-1} h_k \cdot p_{0\to k} \\
&= \pi_0 \cdot h_0 - \pi_0 \cdot h_0 \\
&= 0
\end{aligned}
$$

◀

We now analyze the expected amortized reconfiguration cost due the movement of a single relay node r_j and later sum over all recursive calls of MMRF procedure.

▶ **Lemma D.2.** *Consider an access request σ_t served by ALG, and consider a single run of the procedure MMRF for some node r_j during the recursive call of MMRF procedure. The expected cost of transpositions that r_j participated in, and the potential change due to these transpositions is given by,* $E[C^j_{re}(t) + \Delta\Phi^j] \leq |K_j \cap S_j| \cdot (2d + T) \cdot \pi_0 - |L_j \cap S_j| \cdot h_0 \cdot \pi_0.$

Proof. Let the state of the node r_j be $\mathsf{state}(r_j) = i$ at the time of access and changes to $\mathsf{state}'(r_j) = k$ after reconfiguration.

- **Case-1:** The state of r_j is $\mathsf{state}(r_j) = i$ where $i \neq 0$ and hence r_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) There are $|L_j \cap S_j|$ inversions of type i (at the time of access), which flip to type k since the state of r_j changes from i to k. The change in potential due to flipped inversions is then $|L_j \cap S_j| \cdot (h_k - h_i)$

(4) Since the item does not move forward, no inversions change from hidden to type i and vice versa i.e., $H^j = 0$

(5) No new inversions are created i.e., $A^j = 0$

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

$$E[C^j_{re}(t) + \Delta\Phi^j \mid (\mathsf{state}(r_j) = i), (\mathsf{state}'(r_j) = k)] \leq |L_j \cap S_j| \cdot (h_k - h_i)$$

– **Case-2:** The state of r_j is $\mathsf{state}(r_j) = 0$ and hence r_j is moved forward in the list by paid transpositions. The node r_j is moved to the position just after its direct dependency.

(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 r_j moves forward i.e., $B^j = -(d + h_0) \cdot |L_j \cap S_j|$

(3) There are no old inversions which change their type and hence $F^j = 0$

(4) The expected change in potential due to any inversion that changes from type i to hidden and from hidden to a type i 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, $E[A^j] \leq |K_j \cap S_j| \cdot (d + T)$

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

$$E[C^j_{re}(t) + \Delta\Phi^j \mid (\mathsf{state}(r_j) = 0), (\mathsf{state}'(r_j) = k)] \leq -|L_j \cap S_j| \cdot h_0 + |K_j \cap S_j| \cdot (2d + T)$$

The expected amortized reconfiguration cost for the node r_j is obtained as follows:

$$\begin{aligned}
&E[C^j_{re}(t) + \Delta\Phi^j] \\
&= \sum_{i=1}^{s-1}\sum_{k=0}^{s-1} \pi_i \cdot p_{i\to k} \cdot E[C^j_{re}(t) + \Delta\Phi^j \mid (state(r_j) = i), (state'(r_j) = k)] \\
&+ \sum_{k=0}^{s-1} \pi_0 \cdot p_{0\to k} \cdot E[C^j_{re}(t) + \Delta\Phi^j \mid (state(r_j) = 0), (state'(r_j) = k)] \\
&= \sum_{i=1}^{s-1}\sum_{k=0}^{s-1} \pi_i \cdot p_{i\to k} \cdot (|L_j \cap S_j| \cdot (h_k - h_i)) \\
&+ \sum_{k=0}^{s-1} \pi_0 \cdot p_{0\to k} \, (-|L_j \cap S_j| \cdot h_0 + |K_j \cap S_j| \cdot (2d + T)) \\
&= |L_j \cap S_j| \cdot \sum_{i=1}^{s-1}\sum_{k=0}^{s-1} \pi_i \cdot p_{i\to k} \cdot (h_k - h_i) + |K_j \cap S_j| \cdot (2d + T) \cdot \pi_0 - |L_j \cap S_j| \cdot h_0 \cdot \pi_0 \\
&= |L_j \cap S_j| \cdot \left(-h_0 \cdot \pi_0 + \sum_{i=1}^{s-1}\sum_{k=0}^{s-1} \pi_i \cdot p_{i\to k} \cdot (h_k - h_i) \right) + |K_j \cap S_j| \cdot (2d + T) \cdot \pi_0
\end{aligned}$$

Using Lemma D.1, we substitute $\sum_{i=1}^{s-1}\sum_{k=0}^{s-1} \pi_i \cdot p_{i\to k} \cdot (h_k - h_i) = 0$ in the above equation to obtain $E[C^j_{re}(t) + \Delta\Phi^j] \leq |K_j \cap S_j| \cdot (2d + T) \cdot \pi_0 - |L_j \cap S_j| \cdot h_0 \cdot \pi_0$

◀

We now 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 D.3.** *The amortized cost of serving a request σ_t by ALG is $k \cdot (1+\pi_0 \cdot (2d+T))+1$.*

Proof. Recall from Equation 1 that the amortized cost of ALG in serving a request is the sum of access cost plus the amortized reconfiguration cost.

From Lemma D.2, for each node r_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 r_j for $1 \leq j \leq \delta$ account for all transpositions at time t, thus we sum over all the nodes r_j to obtain the amortized reconfiguration cost of ALG.

$$E[C_{re}(t) + \Delta\Phi] = \sum_{j=1}^{\delta} E[C^j_{re}(t) + \Delta\Phi^j] \leq k \cdot (2d+S) \cdot \pi_0 - l \cdot h_0 \cdot \pi_0$$

The last inequality holds due to the following results from the initial work on the list update problem with precedence constraints [18].

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

We bound the access cost of ALG by $C_{acc}(t) \leq k + \ell + 1$. Finally, from Equation 1 and using the result from Lemma D.2, we obtain the amortized cost of ALG in serving a request,

$$E[a(t)] \leq \overbrace{(k+l+1)}^{access\ cost} + \overbrace{(k \cdot (2d+S) \cdot \pi_0 - l \cdot h_0 \cdot \pi_0)}^{amortized\ reconfiguration\ cost} \leq k \cdot (1 + \pi_0 \cdot (2d+T)) + 1$$

The last inequality holds since $\pi_0 \cdot h_0 = 1$ from Kac's Lemma [9, 11].

◀

D.1.2 Bounding the Competitive Ratio

Events. 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 and includes any paid transpositions made by ALG. We assume a fixed configuration of OPT throughout this event.
- **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.

Proof. 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 $E[a(t)] \leq C \cdot C_{\mathrm{OPT}}$, where $C = \max\{1 + \pi_0 \cdot (2d+T), 1 + \frac{T}{d}\}$.

- **Event-1:** We use the result from Lemma D.3 to bound the amortized cost

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

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.2 Proofs from Section 5

▶ **Lemma 5.2.** *The cost of reconfiguring list L_2 to list L_1 (that share the same dependency graph) using paid exchanges is d times the number of inversions between L_1 and L_2.*

Proof. We prove by induction on the number of nodes. In the base case, there exists a single node in both lists L_1 and L_2, the lists are the same, and the number of inversions is zero.

Now consider node v as the node in front of the list L_1. As the first node in the list, it is not dependent on any other node. Since L_1 and L_2 share the same dependency graph, v can move in front of L_2 as well, without violating any precedence constraints. The inversions that v participate in are with all nodes in front of v in L_2. Hence, the number of inversions multiplied by d is the same as the cost of moving v in front of L_2. We remove node v in both lists, ending with lists with decreased size.

From the induction hypothesis, we can assume that the cost of transforming lists with smaller sizes is d times the number of inversions between them. Therefore, the total cost of reconfiguring L_2 to L_1 would be d times the number of inversions between them. ◀

▶ **Theorem 5.3.** *There exists an optimal offline algorithm for list access with precedence constraints that only performs subset transfers.*

Proof. Assume E_i to be a sequence of paid exchanges by an optimal algorithm before the access request i, and after accessing the previous request.Also, define the sequence of all paid exchanges by the optimal algorithm as $E = \langle E_1, \ldots, E_m\rangle$.

Based on E, we construct $E' = \langle E'_1, \ldots, E'_m\rangle$, such that each sequence of paid exchanges only includes *subset transfer*. We name the initial list of nodes before any paid exchanges as L_0. Consider L_1 to be the list after applying exchanges in E_1 (and L'_1 the list after E'_1).

Let set BB be the nodes before the position of the first requested node, $\mathsf{pos}(\sigma_1)$, in both L_0 and L_1. Similarly, define set BA as the nodes before $\mathsf{pos}(\sigma_1)$ in L_0 but after $\mathsf{pos}(\sigma_1)$ in L_1, and the set AB as the nodes after $\mathsf{pos}(\sigma_1)$ in L_0 but before $\mathsf{pos}(\sigma_1)$ in L_1. Then, we consider the sequence E'_1 to be the subset transfer on all nodes in the set BA. Such a subset transfer is possible since all the nodes that move after σ_1 during E_1 should not have a dependency relation with σ_1. Furthermore, performing the subset set transfer from the nearest node to σ_1 keeps the order among nodes in BA.

Consider E''_1 to be the minimum number of paid exchanges for transforming L'_1 into L_1. If we show that $|E_1| \geq |E'_1| + |E''_1|$, then we replace the E with $\langle E'_1, E''_1 \cup E_2, \ldots, E_m >$ that costs less than E and has one more subset transfer operation. Repeating the procedure described until this point on $\langle E''_1 \cup E_2, \ldots, E_m\rangle$, will transfer E to E' (that only consists of subset transfers).

Now we prove $|E_1| \geq |E'_1| + |E''_1|$. Using Lemma 5.2, we know that the minimum number of paid exchanges for reconfiguring a list to another is d times the number of inversions between the two lists. Therefore, we have $|E_1| \geq d \cdot |\mathsf{inv}(L_0, L_1)|$. On the other hand, $\mathsf{inv}(L_0, L_1')$ and $\mathsf{inv}(L_1', L_1)$ are disjoint and each represent $|E_1'|$ and $|E_1''|$. That is because all in inversions in $\mathsf{inv}(L_0, L_1')$ are between nodes in BA and σ_t or nodes in BB, but none of these inversions appear in $\mathsf{inv}(L_1', L_1)$, as nodes in BA are already moved after σ_t.

So we have $|E_1'| + |E_1''| = d \cdot (|\mathsf{inv}(L_0, L_1')| + |\mathsf{inv}(L_1', L1)|) = d \cdot |\mathsf{inv}(L_0, L_1)|$. Considering the fact that the cost of the initial sequence of paid exchanges is higher than $d \cdot |\mathsf{inv}(L_0, L_1)|$, we end up $|E_1| > |E_1'| + |E_1''|$. ◀