The Itinerant List Update Problem

Neil Olver[1]*, Kirk Pruhs[2]**, Kevin Schewior[3]***, René Sitters[1], and Leen Stougie[1]

[1] Department of Econometrics and Operations Research, Vrije Universiteit Amsterdam, and CWI, Amsterdam, The Netherlands {n.olver,r.a.sitters,l.stougie}@vu.nl

[2] Computer Science Department, University of Pittsburgh, United States kirk@cs.pitt.edu

[3] Institut für Informatik, Technische Universität München, Munich, Germany, and Département d'Informatique, École Normale Supérieure Paris, PSL University, Paris, France kschewior@gmail.com

Abstract. We introduce the *itinerant list update problem (ILU)*, which is a relaxation of the classic list update problem in which the pointer no longer has to return to a home location after each request. The motivation to introduce ILU arises from the fact that it naturally models the problem of track memory management in Domain Wall Memory. Both online and offline versions of ILU arise, depending on specifics of this application. First, we show that ILU is essentially equivalent to a dynamic variation of the classical minimum linear arrangement problem (MLA), which we call DMLA. Both ILU and DMLA are very natural, but do not appear to have been studied before. In this work, we focus on the offline ILU and DMLA problems. We then give an $O(\log^2 n)$-approximation algorithm for these problems. While the approach is based on well-known divide-and-conquer approaches for the standard MLA problem, the dynamic nature of these problems introduces substantial new difficulties. We also show an $\Omega(\log n)$ lower bound on the competitive ratio for any randomized online algorithm for ILU. This shows that online ILU is harder than online LU, for which $O(1)$-competitive algorithms, like Move-To-Front, are known.

1 Introduction

We introduce a variation of the classical list update problem, which we call the *itinerant list update problem (ILU)*. The setting consists of n (data) items, that without loss of generality we will assume are the integers $[n] = \{1, \ldots, n\}$, stored linearly in n locations on a track (tape). The track has a single read/write head. Requests for these items arrive over time. In response to the arrival of a request for an item x, the algorithm can perform an arbitrary sequence of the following unit cost operations:

Move: Move the head to the left, or to the right, one position.

* Supported in part by an NWO Veni grant.

** Supported in part by NSF grants CCF-1421508 and CCF-1535755, and an IBM Faculty Award.

*** Supported by DFG grant GRK 1408, Conicyt Grant PII 20150140, and DAAD PRIME program.

Swap: Swap the item pointed to by the head with the adjacent item on the left, or the adjacent item on the right.

In order to be a feasible response, at some point in this response sequence, the tape head must point to the position holding x. The objective is to minimize the total cost over all requests. In the offline version of ILU, the request sequence is known in advance, and in the online problem, only after the previous request has been serviced.

Our motivation for introducing ILU is that it captures the problem of dynamic memory management of a single track of Domain Wall Memory (DWM). Here, dynamic means that the physical memory location where a data item is stored may change over the execution of the application. DWM technology is discussed in more detail in the full version of the paper, but for our algorithmic purposes it is sufficient to know that conceptually, a track of DWM can be viewed as a tape with a read/write head. At least in the near term, it is envisioned that DWM will be deployed close to the processor in the memory hierarchy, and used as scratchpad memory instead of cache memory, so the stored data there would not have a copy in a lower level of the memory hierarchy [6,7,16,14,13,10]. If the application is an embedded application, where the sequence of memory accesses is (essentially) known before execution, then dynamic memory management can be handled at compile time, and thus is an offline problem [7]. If the sequence of memory accesses is not known before execution, then dynamic memory management would be handled by the operating system at run time, and the problem is online. Additionally in this case, at run time there would need to be an auxiliary data structure translating virtual memory addresses to physical memory addresses. We abstract away this issue (which is independent of the memory technology), and model these two settings by the offline and online ILU problems.

1.1 Relationship of ILU to List Update and Minimum Linear Arrangement

The main difference between ILU and the standard list update problem (LU) is that in LU there is an additional feasibility constraint. At the end of each response sequence, the head has to return to a fixed home position. If the head has a home position, there is a simple $O(1)$-approximation, which is also online: The Move-To-Front (MTF) policy, which moves the last-accessed item to the home position (and moves intermediate items one position further from the home) can be shown to be $O(1)$-competitive by simple modifications to the analysis of MTF in [19]. There, the home position is the first position, and costs are defined somewhat differently.

However, the natural adaptations of MTF for online ILU are all $\tilde{\Omega}(n)$-competitive, see the full version of the paper. These lower bounds hint at an additional difficulty of online ILU relative to the standard list update problem. In both problems it seems natural for the online algorithm to aggregate recently accessed items together. However, in the standard list update problem it is obvious where to aggregate these items, near the home location, while in ILU, it seems unclear where these items should be aggregated.

In the standard formulation of the list update problem [19,2], MTF is 2-competitive, which is optimal for deterministic algorithms [19]. The optimal competitive ratio for randomized algorithms (against an oblivious adversary) is between 1.5 [20] and 1.6 [1]. The offline version of the list update problem is shown to be NP-hard in [3,4], and there is an exact algorithm with running time $O(2^n n! m)$, where m is the number of requests [18]. For a survey of many further results related to the list update problem, see [2,12].

If items can only be reordered once at the beginning (so the memory management is not allowed to be dynamic), then offline ILU is essentially the classical minimum linear arrangement problem (MLA) [11]. In MLA, the input is an edge-weighted graph G with n vertices. The output is an embedding of the vertices of G into a track with n locations. The objective is to minimize the sum over the edges of the weight of the edge times the distance between the endpoints of the edge in the track. Here in the ILU application, the weight of an edge (x, y) roughly corresponds to the number of times that item y is requested immediately after item x is requested. We will make the connection between ILU and MLA more precise shortly.

Hansen [11] gave a polynomial-time $O(\log^2 n)$-approximation algorithm for MLA. This algorithm is a divide-and-conquer algorithm where the divide step computes a balanced cut (say using [15]) to determine a partition of the items into two sets, where all the items in the first set will eventually be embedded to the left of all the items in the second set. As noted by Rao and Richa [17], this same algorithmic design technique can be used to obtain approximation algorithms with similar approximation ratios for the minimum containing interval graph problem, and the minimum storage-time product problem. The algorithm by Feige and Lee [9], which achieves the currently best known approximation guarantee of $O(\sqrt{\log n} \log \log n)$ for these problems, combines rounding techniques for semidefinite programs [5] and spreading-metric techniques [17].

1.2 Our Results

We have already mentioned the connection of ILU to the minimum linear arrangement problem; we now make the connection more precise by defining the dynamic minimum linear arrangement (DMLA) problem. The setting for DMLA is the same as the setting for ILU: a linear track of items $[n]$. A sequence of graphs $H_1, H_2, \ldots$ arrives over time, with the vertex set $V(H_t) = [n]$ for each time t. In response to the graph H_t, the algorithm can first perform an arbitrary sequence of swaps of adjacent items on the track; each such swap has a cost of 1. After this, the *service cost* for H_t is (as in MLA) the sum over the edges of the distance between the current positions of the endpoints in the track. The objective is to minimize the overall cost due to both swaps and service costs. Note that in DMLA there is no concept of a track head, and swaps can be made anywhere on the track. Once again, DMLA has both an online and offline version. The standard MLA problem is essentially a special case of the offline DMLA problem in which all of the many arriving graphs are identical (so there is nothing to be gained from reordering the track).

We use DMLA1 to refer to the DMLA problem restricted to instances where each request graph H_t has a single edge. We show the following.

Theorem 1. *Consider offline ILU, DMLA, and DMLA1. If there is a polynomial-time $f(n)$-approximation algorithm for one of these problems, there are polynomial-time $O(f(n))$-approximation algorithms for the two other problems as well, as long as $f(n) = O(\text{polylog}\, n)$.*

The proof will be provided in the full version of the paper. It involves a somewhat intricate sequence of reductions, summarized in Table **??**.

As our aim is a polylogarithmic approximation to ILU, we can henceforth restrict our attention to DMLA1. Our main theorem is the following.

Theorem 2. *There is a polynomial-time $O(\log^2 n)$-approximation algorithm for offline DMLA1, implying the same for DMLA and ILU.*

As the DMLA problem generalizes the standard MLA problem, it is natural to suspect that the divide and conquer algorithmic design approaches used in, e.g., [11,9], might be applicable. It turns out that the dynamic nature of the problem introduces major new difficulties. We discuss these difficulties, and how we succeed in bypassing them, in Section 2.1. We believe that our more sophisticated algorithm design and analysis techniques may also be useful for other linear arrangement problems, where the simpler techniques used for MLA, the minimum containing interval graph problem and the minimum storage-time product problem are also not sufficient [11,9,17].

We now turn to online ILU. We have already seen that it seems much harder than the classical list update problem. This is confirmed by the following theorem, proved in Section 3.

Theorem 3. *The competitive ratio of any randomized online ILU algorithm against an oblivious adversary is $\Omega(\log n)$.*

In the construction, the algorithm only gradually "learns" that certain items should be close to each other to handle the requests cheaply. To profitably aggregate these items, however, the algorithm would need to know where to aggregate them, which requires more global information. This manifests the difficulties encountered when trying to adapt MTF.

It remains a very interesting and challenging open problem to give a polylog-competitive algorithm for online ILU. The reductions in the full version of the paper show that online ILU, DMLA and DMLA1 are also equivalent, and it suffices to give a polylog-competitive algorithm for online DMLA1. We anticipate that the insights obtained in the analysis of the approximation algorithm will be crucial in making progress.

2 Approximation Algorithm

We now prove Theorem 2, by giving an $O(\log^2 n)$-approximation algorithm for DMLA1; by Theorem 1 this implies the same for ILU and DMLA. The design of our algorithm is described in Section 2.2, and its analysis in Section 2.3. We first give a technical overview of our algorithm and analysis.

2.1 Overview

As the starting point for our algorithm for DMLA1 was the divide-and-conquer algorithm for MLA by Hansen [11], we start by discussing this algorithm. The MLA algorithm finds an approximate mininimum balanced cut of the input graph G into a "left" side and a "right" side (the balance is randomly selected). The algorithm then recurses on the subgraph of G induced by the "left" vertices, and on the subgraph of G induced by the "right" vertices, This recursion is "simple", in the sense that the subproblems are just smaller instances of the MLA problem. This recursive process constructs a laminar family[4] of subsets of the vertices of G, with each set labeled left or right, and from which the ordering can be obtained in the natural way. One issue that must be addressed in the analysis is ruling out the possibility that choosing a high-cost balanced cut at the root can drastically reduce costs at lower levels of the recursion, so that taking a low-cost balanced cut at the root is already an unfixable mistake. In [11] this is handled by showing that the MLA problem has the following *subadditivity property*: the optimal cost for the left subinstance plus the optimal cost for the right subinstance is at most the optimal cost for the original instance. This subadditivity property makes it straightforward to observe that a c-approximate algorithm for minimum balanced cut implies a $c \cdot h$-approximate algorithm for MLA, where $h = \Theta(\log n)$ is the height of the recursion tree.

In order to adapt this algorithm and analysis from MLA to DMLA1, the first question is to determine what should play the role of the input graph. Our algorithm operates on a time-expanded graph G, defined in Definition 1 (see Figure 1), that contains a vertex (x, t) for every item x and time t. A cut in G can again be interpreted as dividing the nodes into a left and right part, but now in a dynamic way: an item x might be in the left side of the cut at some time t_1, but on the right side at another time t_2. "Consistency edges" of the form $\{(x, t-1), (x, t)\}$ play a role in encoding swap costs; the edge contributes to the cut if item x switches sides between times $t-1$ and t. However, we now encounter a significant complication: a balanced cut of G does not suffice. One instead needs a cut that is balanced *at each time*; in other words, a constant fraction of the items should be to the "left" at any given time t. This is crucial for the same reason as in MLA; to ensure, in essence, that the expected distortion between the original line metric and the random tree metric described by the laminar family is not too large.

Before describing how our algorithm finds per-time balanced cuts, we note a major hurdle. Firstly, we cannot hope for a "simple" recursion. Consider the left side of some per-time balanced cut; this will typically have some items that enter and leave this set over time. So from the left subproblem's point of view, items are arriving and leaving over time. It is tempting to try to define a generalization of DMLA1 in which items are allowed to enter and leave. However, we failed to find a formulation of such a problem that (a) we could approximately solve in an efficient manner, and (b) has the subadditivity property which is so critical to

[4] A family $\mathcal{F} \subseteq 2^S$ of sets over some ground set S is called *laminar* if, for all $F_1, F_2 \in \mathcal{F}$, we have $F_1 \cap F_2 = \emptyset$, $F_1 \subseteq F_2$, or $F_1 \supseteq F_2$.

the MLA analysis. Rather than surmounting this hurdle, we more or less bypass it, as we will now describe.

Let us return to the issue of finding per-time balanced cuts. Our algorithm proceeds as follows. First, compute a balanced cut of G; if this is sufficiently cheap, it is easily argued (by virtue of the consistency edges) that the cut is in fact per-time balanced. Otherwise, we find a balanced cut of the subgraph corresponding to those vertices up to some time r, where r is chosen as large as possible but such that the balanced cut is cheap, and hence per-time balanced. Again our algorithm then recurses on the left and right subgraphs of the vertices of G up to time r, but then also recurses on the subgraph consisting of vertices with times after r. This however means that we make no effort whatsoever to prevent a complete reordering between times r and $r+1$; there may be a complete "reshuffle" of the items, and this is not captured in the cost of any of the balanced cuts.

So the final major hurdle is to bound the cost of these reshufflings, which can occur in all levels of the recursion. Because we don't know how to show that DMLA1 has an appropriate subadditivity property, we cannot charge these reshuffling costs locally. This is unlike in the analysis of the MLA algorithm, where all charging is done locally. Instead, we charge, in a somewhat delicate way, the cost of reshuffling at a given level to cuts higher up in the laminar family.

Finally we have to relate the expected reduced cost of the algorithm to the cost of the optimum. This is broadly similar to the MLA algorithm analysis, with some extra technical work to handle the dynamic aspect. In the end we obtain an $O(\log^2 n)$ approximation factor. As in the analysis of the MLA algorithm, we lose one log factor in the approximation of the balanced cut, and one log factor that is really the height of the recursion tree.

2.2 Algorithm Design

We begin by introducing some needed notation.

For a graph $G = (V, E)$ and a subset $W \subseteq V$, $G[W]$ denotes the subgraph induced by W, and $E[W]$ the set of edges in $G[W]$. For a set $S \subseteq V$, $\delta(S)$ denotes the set of edges crossing the cut S. We also use the less standard notation $\delta_W(S)$ to denote $\delta(S) \cap E[W]$. The *balance* of a cut S in G is simply $|S|/|V|$. Furthermore, by T we refer to the total number of requests. The following "time-expanded graph" will be used throughout the algorithm.

Definition 1. *$G = (V, E)$ is defined as follows:*

- *There is a vertex (x, t) for each item $x \in [n]$ and each time t, where $t \in \{0, 1, 2, \dots, T\}$. We call the set of nodes at time t* layer *t, and denote it by L_t.*
- *For each time $t \in [T]$, and the single edge $\{x, y\} \in H_t$, there is an edge $e_t := \{(x, t), (y, t)\}$. We call these* request edges.
- *For each item x and $t \in [T]$ there is an edge $\{(x, t-1), (x, t)\}$. We will call these* consistency edges.

Let E^{r} and E^{c} denote the set of request and consistency edges respectively.

Note that the cost of any solution is certainly at least T, since each request incurs a cost of at least 1. Thus we can afford to return the items to their original order after every n^2 requests, at only a constant-factor increase in cost. This splits up the instance into completely independent sub-instances with at most n^2 requests each, and so we may assume that $T \le n^2$. Next, we prove this formally.

Lemma 1. *If there is a polynomial-time $f(n)$-approximation algorithm A for DMLA1 with $T \le n^2$, then there is a polynomial-time $O(f(n))$-approximation algorithm B for DMLA1 in general.*

Proof. Given some DMLA1 instance I with one edge at a time and without restrictions on T, Algorithm B first cuts I into contiguous sub-instances $I'_1, I'_2, \dots, I'_k$ of n^2 requests each (possibly except for the last one). Then B calls A on each of these sub-instances and then connects these solutions up by moving back to the initial order before each new sub-instance, at a total additional cost of at most $(k-1)n^2$.

Then we have

$$\begin{aligned}
\mathrm{cost}^B(I) &\le \sum_{i=1}^{k} \mathrm{cost}^A(I'_i) + (k-1)n^2 \\
&\le f(n) \cdot \sum_{i=1}^{k} \mathrm{cost}^{\mathrm{OPT}}(I'_i) + (k-1)n^2 \\
&\le f(n) \cdot (\mathrm{cost}^{\mathrm{OPT}}(I) + (k-1)n^2) + (k-1)n^2, \\
&\le O(f(n)) \cdot \mathrm{cost}^{\mathrm{OPT}}(I).
\end{aligned}$$

In the second-to-last step we use that the optimal solution for I can be transformed into optimal solutions for $I'_1, I'_2, \dots, I'_k$ by moving to the identical orders, again at a total additional cost of at most $(k-1)n^2$. In the last step, we use that $\mathrm{cost}^{\mathrm{OPT}}(I)$ is at least the number of requests in I and $f(n) \ge 1$.

This is important when applying approximation algorithms to G (or subgraphs of it) whose approximation guarantees depend on the size of the input graph. We proceed with further definitions.

Definition 2. *For any $W \subseteq V$, let*

$$\begin{aligned}
t_{\min}(W) &:= \min\{t : (x,t) \in W \textit{ for some } x \in [n]\}, \\
t_{\max}(W) &:= \max\{t : (x,t) \in W \textit{ for some } x \in [n]\}.
\end{aligned}$$

We say an item x is present *in W if $(x,t) \in W$ for some $t \in \{0,1,\dots,T\}$. We say x is* permanent *in W if $(x,t) \in W$ for all $t_{\min}(W) \le t \le t_{\max}(W)$; all other items present in W are called* temporary *in W. Let $\alpha(W)$ be the number of items present in W, and $\beta(W)$ the number of temporary items in W. For any $t_{\min}(W) \le r \le t_{\max}(W)$, let $W_{(r)} = \{(x,t) \in W : t \le r\}$. By* layer t of W, *we refer to the set $L_t \cap W$.*

Algorithm Description. The first stage of the algorithm will recursively and randomly hierarchically partition G. The output of this first stage will be described by a laminar family $\mathcal{L}$ on V, with each set $S \in \mathcal{L}$ labeled either *left* or *right*.

So let W be a subset of V, representing the vertex set of a subproblem. Throughout, c will denote a positive constant chosen sufficiently small; $c = 1/100$ suffices. We assume that

$$\beta(W) \leq c\alpha(W); \tag{1}$$

this is of course true when $W = V$ (because $\beta(V) = 0$), and we will ensure that it holds for each subproblem that we create. If $\alpha(W) < 16/c$, we will terminate, and this subproblem will be a leaf of the laminar family constructed. So asume $\alpha(W) \geq 16/c$ from now on. The algorithm chooses κ uniformly from the interval $[\frac{1}{2} - c, \frac{1}{2} + c]$, which is used as the balance parameter for a certain balanced cut problem. The problem differs depending on whether $t_{\min}(W) = 0$ or $t_{\min}(W) > 0$, since the initial ordering is fixed. If $t_{\min}(W) > 0$, define $\bar{G}_{(r)} = G[W_{(r)}]$. If $t_{\min}(W) = 0$, define $\bar{G}_{(r)}$ as the graph obtained from $G[W_{(r)}]$ by choosing z so that the set $A = \{(x, 0) \in W : x \leq z\}$ has cardinality $\kappa\alpha(W)$, and then contracting A into single node s, and the nodes $\{(x, 0) \in W : x > z\}$ into a single node t. Let $\bar{W}_{(r)}$ be the vertex set of $\bar{G}_{(r)}$.

We now compute a cut S_r in $\bar{G}_{(r)}$ with

$$|S_r| \in [(\kappa - 8c)|\bar{W}_{(r)}|, (\kappa + 8c)|\bar{W}_{(r)}|], \tag{2}$$

in such a way that $|\delta_{W_{(r)}}(S_r)| = O(\log |\bar{W}_{(r)}| \cdot |\delta_{\bar{W}_{(r)}}(S^*_r)|)$, where S^*_r is the minimum cut with

$$|S^*_r| \in \big[(\kappa - 4c)|\bar{W}_{(r)}|, (\kappa + 4c)|\bar{W}_{(r)}|\big]. \tag{3}$$

Note that the intervals in (2) and (3) are non-empty because $\alpha(W) \geq 16/c$, implying that both S^*_r and S_r exist; further $O(\log |\bar{W}_{(r)}|) \subseteq O(\log n)$ because $T \leq n^2$. Bicriteria approximation algorithms to balanced cut required to compute some $|S_r|$ as above are well known [15,21]. In the case $t_{\min}(W) = 0$, we ensure that S_r is chosen so that $s \in S_r$, by replacing S_r with $W_{(r)} \setminus S_r$ if necessary. Note that if $|\delta_{\bar{W}_{(r)}}(S_r)| \leq c\alpha(W)$ (which will be the case of interest), S_r will separate s and t. This is because, if $s, t \in S_r$ ($s, t \notin S_r$ analogously), $|\delta_{\bar{W}_{(r)}}(S_r)|$ is at least the number of items x permanent in W for which there is a $(x, \tau) \in W_{(r)} \setminus S_r$. Using the balance requirement (2), we can lower bound this quantity by $(\frac{1}{2} - 9 \cdot c)\alpha(W) - \beta(W) \geq (\frac{1}{2} - 10 \cdot c)\alpha(W)$, which exceeds $c\alpha(W)$ for c sufficiently small. We can interpret S_r as a cut in $G[W_{(r)}]$ by uncontracting s and t.

Roughly, the plan now is to pick r^* as large as possible such that $|\delta_{W_{(r^*)}}(S_{r^*})|$ is not too big; small enough so that we can be sure that at each time t between $t_{\min}(W)$ and r^*, $S_{r^*} \cap L_t$ has size roughly $\kappa\alpha(W)$. However, some additional care is needed, since we also would like that $|\delta_{W_{(r^*)}}(S_{r^*})|$ is not too small—unless $r^* = t_{\max}(W)$. This is needed so that the edges in $\delta_{W_{(r^*)}}(S_{r^*})$ can be charged to later.

Thus, we proceed as follows. We define the cuts $\bar{S}_r$ inductively by: $\bar{S}_{t_{\min}(W)} = S_{t_{\min}(W)}$, and for $r > t_{\min}(W)$, $\bar{S}_r$ is either S_r, or an *extension* S' of $\bar{S}_{r-1}$ to layer r,

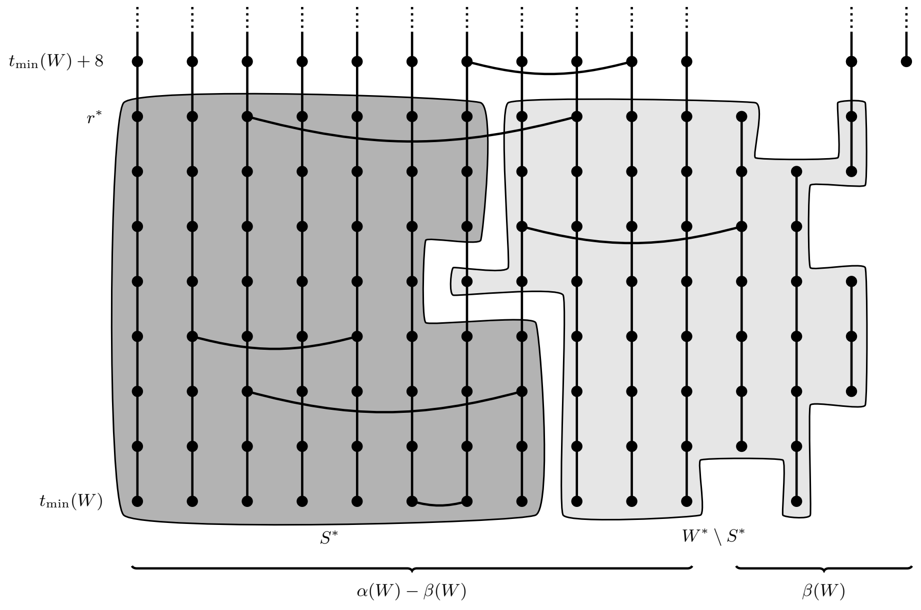

Fig. 1. An example of $G[W]$, S^*, and $W^* \setminus S^*$. For the sake of illustration, we do not require S^* to satisfy the inequalities with the same constants as in our algorithm.

whichever is cheaper. This extension is obtained, roughly speaking, by duplicating the layer $r-1$ nodes of $\bar{S}_{r-1}$, i.e., taking all (x,r) for which $(x,r-1)$ is in $\bar{S}_{r-1}$. But in order to ensure that S' is sufficiently balanced, we adjust this so that $|S' \cap L_r| \in [(\kappa-8c)|W \cap L_r|, (\kappa+8c)|W \cap L_r|]$, by adding or removing an arbitrary set of items of minimum cardinality that is sufficient to satisfy this requirement. Then clearly S' satisfies (2), since inductively $\bar{S}_{r-1}$ satisfied it for $\bar{W}_{(r-1)}$. Note that if q items are added or removed, then $|\delta_{W_{(r)}}(S')| \le |\delta_{W_{(r-1)}}(\bar{S}_{r-1})| + q + 1$; here we use that H_r consists of only a single edge.

Our algorithm sets r^* to be the maximum $r \le t_{\max}(W)$ such that

$$\beta(W_{(r)}) + |\delta_{W_{(r)}}(\bar{S}_r)| \le \tfrac{1}{4} c\alpha(W). \tag{4}$$

We argue that $r = t_{\min}(W)$ always fulfills this inequality, so that r^* does always exist: In this case, $\beta(W_{(r)}) = 0$ and $|\delta_{W_{(r)}}(\bar{S}_r)| \le 1$. To see the latter, distinguish two cases. If $t_{\min}(W) = 0$, then the corresponding layer has been contracted into two nodes, there is only one possible balanced cut, and it can be cut only by a single request edge. If on the other hand $t_{\min}(W) > 0$, then there is a balanced cut of cost 0. Since $\alpha(W) \ge 16/c$ by assumption, the inequality holds.

For convenience let $S^* := \bar{S}_{r^*}$ and $W^* := W_{(r^*)}$. We illustrate this in Figure 1.

We now note a property that will be required later in the analysis.

Property 1. If $r^* < t_{\max}(W)$, then $\beta(W) + |\delta_{W^*}(S^*)| = \Omega(\alpha(W))$.

Proof. Let S' be the extension of S^* that was considered by the algorithm at time r^*+1, and let q be the number of items that needed to be added to or removed from the duplication of layer r^* of S^* in order to ensure $|S' \cap L_{r^*+1}| \in [(\kappa - 8c)|W \cap L_{r^*+1}|, (\kappa + 8c)|W \cap L_{r^*+1}|]$. We bound q: Since S^* fulfills (2), there is a layer r' such that $|S^* \cap L_{r'}| \in [(\kappa - 8c)|W \cap L_{r'}|, (\kappa + 8c)|W \cap L_{r'}|]$. Now note that, when going from layer r' of S^* to layer r^*+1, each item that we need to add or remove in order to restore the balance requirement is due to an edge in $E[W^*]$ being cut by S^* or an item temporary in W leaving or entering. So we have $q \leq |\delta_{W^*}(S^*)| + \beta(W)$.

Then $|\delta_{W_{(r^*+1)}}(S')| \leq |\delta_{W^*}(S^*)| + q + 1 \leq 2|\delta_{W^*}(S^*)| + \beta(W) + 1$. Since the Condition (4) was not satisfied by r^*+1, we deduce that $\frac{1}{4}c\alpha(W) - \beta(W^*) < |\delta_{W_{(r^*+1)}}(S')|$. Combining these inequalities and using that $\beta(W^*) \leq \beta(W)$ yields $\beta(W) + |\delta_{W^*}(S^*)| \geq \frac{1}{8}c\alpha(W) - 1$. Since $\alpha(W) \geq 16/c$, the claim follows.

Furthermore, we have the following lemma.

Lemma 2. *$\alpha(S^*)$ and $\alpha(W^* \setminus S^*)$ are both at least $\frac{1}{4}\alpha(W)$.*

Proof. Let $\rho = r^* - t_{\min}(W) + 1$. Observe that $\rho \cdot \alpha(S^*) \geq |S^*| \geq (\kappa - 8c)|W^*| \geq (\frac{1}{2} - 9c)|W^*|$. These inequalities follow by the definition of ρ and $\alpha(\cdot)$, Inequality (2), and the choice of κ and c, respectively. Since $|W^*|$ is at least ρ times the number of permanent items in W, $\alpha(S^*) \geq (\frac{1}{2} - 9c)(\alpha(W) - \beta(W)) \geq (\frac{1}{2} - 9c)(1-c)\alpha(W) \geq \frac{1}{4}\alpha(W)$, where the last inequality follows by our choice of c. A symmetric argument holds for $W^* \setminus S^*$. □

Since the number of temporary items in S^* is at most the number of temporary items in W^* plus the size of the cut $\delta_{W^*}(S^*)$, (4) yields that $\beta(S^*) \leq \frac{1}{4}c\alpha(W)$, and so $\beta(S^*) \leq c\alpha(S^*)$ by the lemma. Similarly, $\beta(W^* \setminus S) \leq c\alpha(W^* \setminus S^*)$. This shows that the algorithm may recurse in S^* and $W^* \setminus S^*$. Eventually this recursion yields labeled families $\mathcal{L}_{\text{left}}$ and $\mathcal{L}_{\text{right}}$ on S^* and $W^* \setminus S^*$, respectively. If $W \setminus W^* \neq \emptyset$, the algorithm also iterates on $W \setminus W^*$, obtaining $\mathcal{L}_{\text{rest}}$. The resulting labeled laminar family $\mathcal{L}$ is $\mathcal{L}_{\text{left}} \cup \mathcal{L}_{\text{right}} \cup \mathcal{L}_{\text{rest}} \cup \{S^*, W^* \setminus S^*\}$, where S^* is labeled left and $W^* \setminus S^*$ is labeled right. We call S^* and $W^* \setminus S^*$ *siblings.* Note that W may have many direct children in $\mathcal{L}$, but each layer intersects precisely one sibling pair. Also note that since $|S^*| \in \left[(\frac{1}{2} - 9c)|W^*|, (\frac{1}{2} + 9c)|W^*|\right]$, $\mathcal{L}$ has logarithmic depth.

Once $\mathcal{L}$ has been constructed, the ordering $\prec_t$ of the algorithm's list A_t at time t is determined in the following manner. If there exists a set in $\mathcal{L}$ containing (x,t) but not (y,t), and the maximal such set S is labeled left, then x is to the left of y in A_t, that is, $x \prec_t y$; if S is labeled right, x is to the right of y, that is, $y \prec_t x$. If there is no set in $\mathcal{L}$ containing (x,t) but not (y,t), we let $x \prec_t y$ if and only if $x < y$, i.e., we order x and y according to the initial ordering. In the latter case, we say *$x \prec_t y$ by default.* Note that this rule yields the correct ordering for A_0, since for any $x < y$, at the moment where they are separated, we ensure that $(x,0) \in S^*$ and $(y,0) \notin S^*$.

2.3 Algorithm Analysis

Definition 3. *Given two orderings A, B of the items, and any two distinct items x, y, we say that (x, y) is a* discordant pair (for A and B) *if x and y have a different relative order in A and in B.*

Note that the number of discordant pairs for A and B is precisely the permutation distance between A and B, i.e., the minimum number of swaps of adjacent items required to obtain order B starting from order A.

Definition 4. *For $S \in \mathcal{L}$, define* $\mathrm{cost}(S)$ *to be $\alpha(S) \cdot |\delta(S)|$ and* $\mathrm{cost}(\mathcal{L})$ *to be $\sum_{S \in \mathcal{L}} \mathrm{cost}(S)$. Let* $\mathrm{parent}(S)$ *be the parent of S, meaning the minimal set $S' \in \mathcal{L} \cup \{V\}$ with $S' \supsetneq S$ (which is unique by the laminarity of $\mathcal{L}$). (By including V here, we ensure that every set in $\mathcal{L}$ has a parent.) Let* $\mathrm{pair}(S)$ *be the union of S and its sibling (i.e., the other child of* $\mathrm{parent}(S)$ *in $\mathcal{L}$ which covers the same layers as S). Note that $S \subsetneq \mathrm{pair}(S) \subseteq \mathrm{parent}(S)$*

Lemma 3. *The cost of the algorithm is at most* $8 \cdot \mathrm{cost}(\mathcal{L}) + O(\mathrm{OPT})$ *(irrespective of the random choices made in the algorithm).*

Proof. We argue separately for each time t. First consider the swap costs, so let $t \geq 1$, and define $E_t^{\mathrm{c}} = \bigl\{\{(x, t-1), (x, t)\} : x \in [n]\bigr\}$. The proof strategy for this part is to consider discordant pairs for A_{t-1} and A_t, assigning them to certain sets $S \in \mathcal{L}$, and later counting the number of discordant pairs assigned to each set $S \in \mathcal{L}$. Hence suppose $x \prec_{t-1} y$ and $y \prec_t x$ (so (x, y) is a discordant pair for A_{t-1} and A_t).

First consider the case that $x \prec_{t-1} y$ by default; the case where $y \prec_t x$ by default is analogous. As $x \prec_{t-1} y$ by default, there is a set T minimal in $\mathcal{L}$ with $(x, t-1), (y, t-1) \in T$. Since (x, y) is a discordant pair, however $y \prec_t x$ not by default. Hence, the construction of A_t from $\mathcal{L}$ tells us that there is a unique left-labeled set $U \in \mathcal{L}$ with $(y, t) \in U$, $(x, t) \in \mathrm{parent}(U) \setminus U$. We say that U *certifies* that $y \prec_t x$. Note that $U \cap T = \emptyset$ or $(\mathrm{parent}(U) \setminus U) \cap T = \emptyset$ (or both). Assign discordant pair (x, y) to the corresponding set out of $U, \mathrm{parent}(U) \setminus U$.

Now consider the case that neither $x \prec_{t-1} y$ by default nor $y \prec_t x$ by default. Again we know that there is a left-labeled set $U \in \mathcal{L}$ certifying $y \prec_t x$. Similarly, there is a left-labeled set T certifying that $x \prec_{t-1} y$. We prove the following claim.

Claim. $(x, t) \notin T$ or $(y, t-1) \notin U$ (or both).

Proof (Claim). Suppose not. Then T contains $(x, t-1)$ and (x, t), and U contains $(y, t-1)$ and (y, t). Moreover T is a maximal set in $\mathcal{L}$ containing $(x, t-1)$ and not $(y, t-1)$, and U is a maximal set containing (y, t) and not (x, t); we deduce that $T' := \mathrm{parent}(T) = \mathrm{parent}(U)$. But this contradicts the assumption that T and U are both labeled left; within the subproblem induced by T', the algorithm produces only one set labeled left containing nodes at time t. □

So in this case assign discordant pair (x, y) to T if $(x, t) \notin T$, and to U if $(y, t-1) \notin U$ (if both occur, the assignment can be arbitrary). Now consider the assignment of discordant pairs (in both cases) from the perspective of a set $S \in \mathcal{L}$. If (x, y) is assigned to S, then for $(z, \bar{z}) \in \{(x, y), (y, x)\}$ we have

– $(z, t-1) \in S$, $(\bar{z}, t-1) \in \text{parent}(S) \setminus S$, and $(z, t) \notin S$,
– or $(z, t) \in S$, $(\bar{z}, t) \in \text{parent}(S) \setminus S$, and $(z, t-1) \notin S$.

Thus we can bound the total number of discordant pairs assigned to S by

$$\begin{aligned} &2 \cdot (|\{z : (z, t-1) \in S, (z, t) \notin S\}| \\ &\quad + |\{z : (z, t) \in S, (z, t-1) \notin S\}|) \cdot \alpha(\text{parent}(S)) \\ &= 2 \cdot |\delta(S) \cap E_t^{\text{c}}| \cdot \alpha(\text{parent}(S)) \\ &\leq 8 \cdot |\delta(S) \cap E_t^{\text{c}}| \cdot \alpha(S), \end{aligned}$$

where the last inequality follows by Lemma 2. So the cost to optimally reorder A_{t-1} to A_t, being exactly the number of discordant pairs, is at most $8 \cdot \sum_{S \in \mathcal{L}} \alpha(S) \cdot |\delta(S) \cap E_t^{\text{c}}|$.

We now consider the service cost at time t. The request pair is an edge $e_t = \{(x, t), (y, t)\}$ in G; assume that $x \prec_t y$, otherwise relabel x and y. The algorithm pays the distance between x and y in A_t. If $x \prec_t y$ by default, the distance between x and y in A_t is at most $16/c$ by construction, and the cost is taken care of by the $O(\text{OPT})$ term in the cost bound, because OPT pays at least one for the considered request. Otherwise consider the set S certifying that $x \prec_t y$. Then $S' := \text{parent}(S)$ contains both (x, t) and (y, t), and $\alpha(S')$ clearly bounds the distance between x and y in A_t (because any item outside of S' is either to the left of both x and y or to the right of both). Since $\alpha(S') \leq 4 \cdot \alpha(S)$ (again Lemma 2), we conclude that the service cost at time t is at most $4 \cdot \sum_{S \in \mathcal{L}} \alpha(S) \cdot \mathbf{1}_{e_t \in \delta(S)}$. Combining the swap and service costs at all times, we obtain the lemma. □

Before we state the next lemma, we note that that for any $S \in \mathcal{L}$, $\text{pair}(S)$ is exactly the set within which S was a good balanced cut, and $|\delta_{\text{pair}(S)}(S)|$ is exactly the cost of S in this balanced cut problem.

Definition 5. *We define* $\text{cost}_{\text{core}}(S)$ *to be* $\alpha(S) \cdot |\delta_{\text{pair}(S)}(S)|$, *and* $\text{cost}_{\text{core}}(\mathcal{L})$ *to be* $\sum_{S \in \mathcal{L}} \text{cost}_{\text{core}}(S)$.

Lemma 4. *Irrespective of the random choices made by the algorithm, we have* $\text{cost}_{\text{core}}(\mathcal{L}) = \Omega(\text{cost}(\mathcal{L}))$.

Proof. Begin by assigning to each pair (e, S), where $S \in \mathcal{L}$ and $e \in \delta(S)$, a charge of $\alpha(S)$. Our goal is to redistribute this charge to pairs (e, S) where $e \in \delta_{\text{pair}(S)}(S)$, and where each such pair gets a total charge of $O(\alpha(S))$. This clearly implies the lemma.

For a set $S \in \mathcal{L}$, we call an edge of the form $\{(x, t_{\max}(S)), (x, t_{\max}(S)+1)\}$ that crosses S a *top shuffle edge for* S, and similarly an edge of the form $\{(x, t_{\min}(S)-1), (x, t_{\min}(S))\}$ a *bottom shuffle edge for* S. Let $Q(S)$ denote the set of shuffle edges (either top or bottom) for S. Note that $Q(S) \subseteq \delta(S) \setminus \delta_{\text{pair}(S)}(S)$. Now notice that we have the following downward-closed property.

Claim. If e is a shuffle edge for some $S \in \mathcal{L}$, and $e \in \delta(T)$ for some $T \in \mathcal{L}$ with $T \subset S$, then e is a shuffle edge for T as well.

We reassign the charge in stages. In the first stage, we reassign all the charges involving an edge e to a maximal $S \in \mathcal{L}$ with $e \in \delta(S)$. Note that there are two possible choices for S. If $e \in E^{c}$, choose the S containing the earlier endpoint of e and not the later one; if $e \in E^{r}$, make any choice. Now consider any (e, S). It may receive charge from multiple consistency edges, whose initial charges are geometrically decreasing starting from $\alpha(S)$, and a single request edge with initial charge $\alpha(S)$. So (e, S) has charge $O(\alpha(S))$ after this reassignment. Moreover, by the above choice of S, no bottom shuffle edges have any charge remaining.

For the next stage, we prove the following statement.

Claim. For any $S \in \mathcal{L}$, the total charge in $Q(S)$ is $O\big(\alpha(S) \cdot |\delta(S) \setminus Q(S)|\big)$.

Proof (Claim). Let $W = \text{parent}(S)$ and $U = \text{pair}(S)$. There are two cases. The first case is if $t_{\max}(S) = t_{\max}(W)$. In this case, all top shuffle edges for S cross W as well, and so have no charge. The second case is if $t_{\max}(S) < t_{\max}(W)$. In this case, since each shuffle edge of S currently has charge at most $O(\alpha(S))$, it suffices to show that $|Q(S)| = O(|\delta(S) \setminus Q(S)|)$. Notice that $|\delta(S) \setminus Q(S)| \geq |\delta_U(S)| + \beta(W)$, because every non-shuffle edge crossing S is either contained in U, or crosses W, meaning it was a temporary node in W. But since $t_{\max}(S) < t_{\max}(W)$, we know from Property 1 that $|\delta_U(S)| + \beta(W) = \Omega(\alpha(W))$. The claim follows since $|Q(S)| \leq \alpha(S)$ and $\alpha(S) \leq 4 \cdot \alpha(W)$, by Lemma 2. □

It follows that, in the next stage, we can now redistribute all charge on the shuffle edges of a set $S \in \mathcal{L}$ to other edges, maintaining that no edge of $\delta(S)$ has a charge more than $O(\alpha(S))$.

In the final stage, we again reassign all charge of an edge to a maximal set that it crosses; each pair (e, S) still has a charge $O(\alpha(S))$. It remains true that no pair (e, S) with $e \in Q(S)$ gets any charge, because of Claim 2.3. So all charge for a set S is on edges that are not shuffle edges, and which do not cross parent(S); these are precisely the edges of $\delta_{\text{pair}(S)}(S)$. This completes the proof of Lemma 4. □

Lemma 5. $\mathbb{E}[\text{cost}_{\text{core}}(\mathcal{L})] = O(\log^2 n) \cdot \text{OPT}$.

Proof. Let A_t^* denote the ordering in the optimum solution after responding to request t. Let $S_1, \dots S_k$ be the left elements of $\mathcal{L}$ that are of some depth d in this laminar family. Let $W_i = \text{parent}(S_i)$ and $U_i = \text{pair}(S_i)$. (Note that the U_i's are disjoint, but many of the W_i's may be the same.) We fix the random choices made by the algorithm above level d (thus we may consider W_i to be deterministic for each i, although U_i is random). Define, for any $i \in [k]$ and $S \subseteq \text{pair}(S_i)$, $\text{cost}_{\text{core}}(S) = \alpha(S) \cdot |\delta_{\text{pair}(S_i)}(S)|$. Then for each S_i, we will show how to derive from OPT a (random) balanced cut C_i of $G[U_i]$, such that $\text{cost}_{\text{core}}(S_i) = O(\log n) \cdot \text{cost}_{\text{core}}(C_i)$ and

$$\sum_{i=1}^{k} \mathbb{E}[\text{cost}_{\text{core}}(C_i)] = O(\text{OPT}). \tag{5}$$

The expectation is over the random choices made by the algorithm at layer d. The result then follows, since $\mathcal{L}$ has depth $O(\log n)$.

Now fix some $i \in [k]$. Note that we can assume $\alpha(W_i) \geq 16/c$. We define C_i as follows. Let κ_i denote the random choice made by the algorithm for the subproblem U_i. and let $m_i = \lfloor \kappa_i \alpha(W_i) \rfloor$. Now consider the permanent items of W_i as they appear in A_t^*, and let $p_{i,t}$ be the position of the m_i'th such item, for any $i \in [k]$ and $t_{\min}(U_i) \leq t \leq t_{\max}(U_i)$. Note that the probability that $p_{i,t}$ takes any specific value is $O(1/\alpha(W_i))$. Then define

$$C_i = \{(x,t) \in U_i : x \text{ is at or to the left of position } p_{i,t} \text{ in } A_t^*\}.$$

Note that the C_i's are obviously disjoint sets, since the U_i's are disjoint.

We first prove that $\mathrm{cost}_{\mathrm{core}}(S_i) = O(\log n) \cdot \mathrm{cost}_{\mathrm{core}}(C_i)$ (irrespective of the random choice of κ_i). To see this, consider some layer of U_i, and observe that the number of nodes in it is in $[\alpha(W_i) - \beta(W_i), \alpha(W_i)] \subseteq [(1-c) \cdot \alpha(W_i), \alpha(W_i)]$ (using (1)). On the other hand, the number of nodes contained in any layer of C_i is in $[\lfloor \kappa_i \cdot \alpha(W_i) \rfloor, \lfloor \kappa_i \cdot \alpha(W_i) \rfloor + \beta(W_i)] \subseteq [\kappa_i \cdot \alpha(W_i) - 1, (\kappa_i + c) \cdot \alpha(W_i)]$ (again using (1)). Putting the two observations together and using $\alpha(W_i) \geq 16/c$ yields that any layer of C_i contains a fraction in $[\kappa_i - c, \kappa_i + 3c]$ of the nodes of the corresponding layer of U_i. Summing over all layers shows that the balance of C_i in U_i is in $[\kappa_i - c, \kappa_i + 3c] \subseteq [\frac{1}{2} - 2c, \frac{1}{2} + 4c]$. Therefore, $\alpha(C_i) = \Theta(\alpha(S_i))$. Applying this to $|\delta_{U_i}(S_i)| = O(\log n) \cdot |\delta_{U_i}(C_i)|$, which follows from the fact that C_i fulfills (3) in the definition of the algorithm, yields $\mathrm{cost}_{\mathrm{core}}(S_i) = O(\log n) \cdot \mathrm{cost}_{\mathrm{core}}(C_i)$.

Next, we show (5). Look at any request edge e_t. If it is not contained within U_i for some i, then it does not contribute to the left hand side of (5), so suppose it is contained in U_i. Let $q_1 < q_2$ be the positions of the endpoints of e_t in A_t^*; so OPT pays $q_2 - q_1$. Then the probability that e_t crosses C_i is $\mathbb{P}(q_1 \leq p_{i,t} < q_2)$, which is $O((q_2 - q_1)/\alpha(W_i))$. If e_t does cross C_i, it contributes $\alpha(C_i) = \Omega(\alpha(S_i))$, and so its expected contribution is $O(q_2 - q_1)$.

Now consider the swap cost. The swap cost in the optimal solution at time t is the number of discordant pairs for A_{t-1}^* and A_t^*. So assign the swap cost to an item x at time t to be equal to the number of such discordant pairs that include x. The sum of the costs assigned to all items over all times is then exactly twice the swap cost of OPT.

So fix some item x and time t. Suppose $\{(x, t-1), (x,t)\} \in E[U_i]$ for some i (otherwise again, it does not contribute to (5)). Let q_1 and q_2 be the number of permanent items in W_i to the left of x in A_{t-1}^* and A_t^*, respectively. Once again, the probability that $\{(x,t-1),(x,t)\} \in \delta_{U_i}(C_i)$ is at most $O(|q_2 - q_1|/\alpha(W_i))$, yielding an expected contribution to (5) of $O(|q_2 - q_1|)$. But $|q_2 - q_1|$ is at most the number of discordant pairs involving x. Summing over all x and t completes the proof. □

Combining Lemma 3, Lemma 4 and Lemma 5 yields that the cost of the algorithm is $O(\log^2 n \cdot \mathrm{OPT})$, as desired.

3 Lower Bound for ILU

We now prove that there is no randomized $o(\log n)$-competitive online algorithm for ILU.

Proof (Theorem 3). We apply Yao's principle: For a particular input distribution, we show that the expected cost for every deterministic online algorithm is a $\Omega(\log n)$ factor more than the expected optimal cost [8]. Conceptually, underlying the lower bound construction is a complete binary tree T, of depth $q = \Theta(\log n)$, with n leaves. We think of each internal node of T as having a left and right subtree; thus the leaves of T can be associated with the positions on the track. We begin by choosing an uniformly random initial assignment δ of items to leaves. The adversary initially orders the items in the track to match this assignment, and will not move the items after this.

The sequence of requests consists of q rounds. Each round i, $0 \leq i \leq q-1$, the request sequence π_i is a permutation of $[n]$. We define a *depth-d subtree* to be a subtree rooted at a node of depth d in T. Here we assume the root of T has depth 0. To obtain π_i, the depth-$(q-i)$ subtrees, of which there are 2^{q-i}, are first ordered uniformly at random (and independent of all other random choices). Then, while maintaining the order of the subtrees, the 2^i leaves within each of the depth-$(q-i)$ subtrees are uniformly randomly ordered (again independent of all other random choices). To make this more precise, let $v_{\sigma(1)}, \ldots, v_{\sigma(2^{q-i})}$ be a random permutation of the vertices of depth $q-i$ in T. For each $1 \leq j \leq 2^{q-i}$, let $v_{\rho_j(1)}, \ldots, v_{\rho_j(2^i)}$ be a random permutation of the leaves of the subtree of T rooted at $v_{\sigma(j)}$. Then $v_{\rho_j(k)}$ precedes $v_{\rho_{j'}(k')}$ in π_i if and only if j occurs before j' in σ, or $j = j'$ and k occurs before k' in ρ_j.

We now bound the costs for the optimum. The only swap cost is incurred initially and is no more than n^2. During π_i, movement between two items in the same depth-$(q-i)$ subtree costs at most 2^i. Thus the total movement between items in the same depth-$(q-i)$ subtrees costs at most $n2^i$. Movement between two such items costs at most n.

The movement cost for the first item is at most n. Also, movement between two items in different depth-$(q-i)$ subtrees costs at most n. There are exactly $2^{q-i}-1$ consecutive accesses to items in different depth-$(q-i)$ subtrees. Thus the total movement cost between items in different depth-$(q-i)$ subtrees is at most $n2^{q-i}$. Summing over i, we get that the adversary's cost is at most $\sum_{i=1}^{q} n2^i + n2^{q-i} = O(n^2)$.

We now bound the expected cost for the online algorithm. We can generously assume that the online algorithm knows the adversary's strategy for constructing the sequences π_i and that it sees π_i just before round i. Before seeing π_i, the online player knows the depth-$(q-i+1)$ subtrees of T, but it has no information at all on how depth-$(q-i+1)$ subtrees are paired to form the depth-$(q-i)$ subtrees. So an alternative equivalent way to randomly generate δ would be to at this time randomly pair the depth-$(q-i+1)$ subtrees.

For the moment assume that the online algorithm does not reorder the items during round i. Then consider the 2^i consecutive requests to the leaves in some depth-$(q-i)$ subtree in π_i. In expectation, at least a constant fraction of these consecutive accesses will be to items in different depth-$(q-i+1)$ subtrees. As the depth-$(q-i+1)$ subtrees are paired up randomly, any online algorithm will have to move in expectation $\Omega(n)$ positions in response to the requests in the different depth-$(q-i+1)$ subtrees. Hence, the expected cost for the algorithm for each round is $\Omega(n^2)$ if the online algorithm makes no swaps after seeing permutation π_i. But note that any swap made after seeing π_i can reduce the movement cost in round i by at most 2 since each item is requested only once in π_i. Hence, the cost for the online algorithm is $\Omega(n^2)$ per round, and $\Omega(n^2 \log n)$ in total.

Note that this construction can be repeated to rule out the possibility of a $o(\log n)$-competitive algorithm with an additive error term.

Acknowledgements. We acknowledge Suzanne Den Hertog (née van der Ster) for many helpful discussions. Part of this work was done while several of the authors were participating in the Hausdorff Trimester on Discrete Mathematics in Fall 2015.

References

1. Susanne Albers, Bernhard von Stengel, and Ralph Werchner. A combined BIT and TIMESTAMP algorithm for the list update problem. *Infor. Process. Lett.*, 56(3):135–139, 1995.
2. Susanne Albers and Jeffery Westbrook. Self-organizing data structures. In *Online Algorithms, The State of the Art*, pages 13–51, 1996.
3. Christoph Ambühl. Offline list update is NP-hard. In *European Symposium on Algorithms (ESA)*, pages 42–51, 2000.
4. Christoph Ambühl. SIGACT news online algorithms column 31. *SIGACT News*, 48(3):68–82, September 2017.
5. Sanjeev Arora, Satish Rao, and Umesh V. Vazirani. Expander flows, geometric embeddings and graph partitioning. *J. ACM*, 56(2), 2009.
6. Oren Avissar, Rajeev Barua Rajeev, and Dave Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Trans. Embed. Comput. Syst.*, 1(1):6–26, November 2002.
7. Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. In *Symposium on Hardware/Software Codesign*, pages 73–78, New York, NY, USA, 2002. ACM.
8. Allan Borodin and Ran El-Yaniv. On randomization in online computation. In *IEEE Conference on Computational Complexity (CCC)*, pages 226–238, 1997.
9. Uriel Feige and James R. Lee. An improved approximation ratio for the minimum linear arrangement problem. *Inf. Process. Lett.*, 101(1):26–29, 2007.
10. Shouzhen Gu, Edwin Sha, Qingfeng Zhuge, Yiran Chen, and Jingtong Hu. Area and performance co-optimization for domain wall memory in application-specific embedded systems. In *Proceedings of the 52Nd Annual Design Automation Conference*, Design Automation Conference, pages 20:1–20:6, New York, NY, USA, 2015. ACM.
11. Mark D. Hansen. Approximation algorithms for geometric embeddings in the plane with applications to parallel processing problems. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 604–609, 1989.

12. Shahin Kamali and Alejandro López-Ortiz. A survey of algorithms and models for list update. *Space-Efficient Data Structures, Streams, and Algorithms*, pages 251–266, 2013.
13. M. Kandemir, J. Ramanujam, and A. Choudhary. Exploiting shared scratch pad memory space in embedded multiprocessor systems. In *Design Automation Conference*, pages 219–224, 2002.
14. M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic management of scratch-pad memory space. In *Proceedings of the 38th Annual Design Automation Conference*, Design Automation Conference, pages 690–695, 2001.
15. Frank Thomson Leighton and Satish Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *J. ACM*, 46(6):787–832, 1999.
16. Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. In *European Design and Test Conference*, 1997.
17. Satish Rao and Andréa W. Richa. New approximation techniques for some linear ordering problems. *SIAM J. Comput.*, 34(2):388–404, 2004.
18. Nick Reingold and Jeffery Westbrook. Off-line algorithms for the list update problem. *Inf. Process. Lett.*, 60(2):75–80, 1996.
19. D.D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *CACM*, 28(2):202–208, 1985.
20. Boris Teia. A lower bound for randomized list update algorithms. *Inf. Process. Lett.*, 47:5–9, 1993.
21. David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms.* Cambridge University Press, 2011.