Self-Adjusting Linear Networks

Chen Avin[1] [0000−0002−6647−8002] Ingo van Duijn[2⋆] Stefan Schmid[3] [0000−0002−7798−1711]

[1] School of Electrical and Computer Engineering, Ben Gurion University of the Negev, avin@cse.bgu.ac.il

[2] Department of Computer Science, Aalborg University, ingo@cs.aau.dk

[3] Faculty of Computer Science, University of Vienna, stefan_schmid@univie.ac.at

Abstract. Emerging networked systems become increasingly flexible, reconfigurable, and "self-∗". This introduces an opportunity to adjust networked systems in a demand-aware manner, leveraging spatial and temporal locality in the workload for *online* optimizations. However, it also introduces a tradeoff: while more frequent adjustments can improve performance, they also entail higher reconfiguration costs. This paper initiates the formal study of *list* networks which self-adjust to the demand in an online manner, striking a balance between the benefits and costs of reconfigurations. We show that the underlying algorithmic problem can be seen as a distributed generalization of the classic dynamic list update problem known from self-adjusting datastructures: in a network, requests can occur between *node pairs*. This distributed version turns out to be significantly harder than the classical problem it generalizes. Our main results are a $\Omega(\log n)$ lower bound on the competitive ratio, and a (distributed) online algorithm that is $\mathcal{O}(\log n)$-competitive if the communication requests are issued according to a *linear order*.

Keywords: self-adjusting datastructures · competitive analysis · distributed algorithms · communication networks.

1 Introduction

Communication networks are becoming increasingly flexible, along three main dimensions: routing (enabler: software-defined networking), embedding (enabler: virtualization), and topology (enabler: reconfigurable optical technologies, for example [17]). In particular, the possibility to quickly reconfigure communication networks, e.g., by migrating (virtualized) communication endpoints [9] or by reconfiguring the (optical) topology [12], allows these networks to become *demand-aware*: i.e., to adapt to the traffic pattern they serve, in an online and "self-∗" manner. In particular, in a *self-adjusting* network, frequently communicating node pairs can be moved *topologically closer*, saving communication costs (e.g., bandwidth, energy) and improving performance (e.g., latency, throughput).

⋆ Corresponding author.

However, today, we still do not have a good understanding yet of the algorithmic problems underlying self-adjusting networks. The design of such algorithms faces several challenges. As the demand is often not known ahead of time, *online* algorithms are required to react to changes in the workload in a clever way; ideally, such online algorithms are "competitive" even when compared to an optimal offline algorithm which knows the demand ahead of time. Furthermore, online algorithms need to strike a balance between the benefits of adjustments (i.e., improved performance and/or reduced costs) and their costs (i.e., frequent adjustments can temporarily harm consistency and/or performance, or come at energy costs).

The vision of self-adjusting networks is reminiscent of self-adjusting datastructures such as *self-adjusting lists* and *splay trees*, which optimize themselves toward the workload. In particular, the *dynamic list update problem*, introduced already in the 1980s by Sleator and Tarjan in their seminal work [23], asks for an online algorithm to reconfigure an unordered linked list datastructure, such that a sequence of lookup requests is served optimally and at minimal reconfiguration costs (i.e., pointer rotations). It is well-known that a simple *move-to-front* strategy, which immediately promotes each accessed element to the front of the list, is *dynamically optimal*, that is, has a constant competitive ratio.

This paper initiates the study of a most basic self-adjusting linear *network*, which can be seen as a *distributed* variant of the dynamic list update problem, generalizing the datastructure problem to networks: while datastructures serve requests originating from the front of the list (the "root") to access data items, networks serve *communication* requests between *pairs of nodes*. The objective is to move nodes which currently communicate frequently, closer to each other, while accounting for reconfiguration costs.

1.1 Related Work

One important area of related work arises in the context of the dynamic list update problem. Since the groundbreaking work by Sleator and Tarjan on amortized analysis and self-adjusting datastructures [23], researchers have also explored many interesting variants of self-adjusting datastructures, also using randomized algorithms [21] or lookaheads [1,3], or offline algorithms [5,20]. The deterministic Move-To-Front (MTF) algorithm is known to optimally solve the standard formulation of the list update problem: it is 2-competitive [23], which matches the lower bound [4]. To the best of our knowledge, the competitive ratio in the randomized setting (against an oblivious adversary) is still an open problem: the best upper bound so far is 1.6 [3], and the best lower bound 1.5 [24]. The randomized algorithm [3] makes an initial random choice between two known algorithms that have different worst-case request sequences, relying on the BIT [21] and TIMESTAMP [2] algorithms.

We also note that the self-adjusting linear network design problem can be considered a special case of general online problems such as the online Metrical Task System (MTS) problems. However, given the exponential number of possible configurations, the competitive ratio of generic MTS algorithms will be high if applied to our more specific problems (at least according to the existing bounds). Furthermore, we note that in case of list request graphs, the problem can also be seen as a learning problem and hence related to bandits theory [13].

In terms of reconfigurable networks, there exist several static [8,11] and dynamic [22,19,16] algorithms for bounded-degree networks, as well as hybrid variants [15] which combine static and reconfigurable links. However, these solutions do not apply to the list and do not provide performance guarantees over time (with the notable exception of [16] in a different model); the latter also applies to recent work on node migration models on the grid [7].

The paper closest to ours is by Olver et al. [18] who introduced the Itinerant List Update (ILU) problem: a relaxation of the classic dynamic list update problem in which the pointer no longer has to return to a home location after each request. The authors show an $\Omega(\log n)$ lower bound on the randomized competitive ratio and also present an offline polynomial-time algorithm and prove that it achieves an approximation ratio of $O(\log^2 n)$. In contrast, we in our paper focus on online algorithms and request graphs forming a list (or grid). In fact, we show that the lower bound $\Omega(\log n)$ even holds in this case, at least for deterministic algorithms. We also present an online algorithm which matches this bound in our model.

1.2 Formal Model

We initiate the study of pairwise communication problems in a dynamic network reconfiguration model, using the following notation:

- Let $d_G(u, v)$ denote the *(hop) distance* between u and v in a graph G.
- A *communication request* is a pair of communicating nodes from a set V.
- A *configuration* of V in a graph N (the host network) is an injection of V into the vertices of N; $C_{V \hookrightarrow N}$ denotes the set of all such configurations.
- A configuration $h \in C_{V \hookrightarrow N}$ is said to *serve* a communication request $(u, v) \in V \times V$ at cost $d_N(h(u), h(v))$.
- A finite *communication sequence* $\sigma = (\sigma_0, \sigma_1, \ldots, \sigma_m)$ is served by a sequence of configurations $h_0, h_1, \ldots, h_m \in C_{V \hookrightarrow N}$.
- The cost of serving σ is the sum of serving each σ_i in h_i plus the reconfiguration cost between subsequent configurations h_i, h_{i+1}.

- The reconfiguration cost between h_i, h_{i+1} is the number of *migrations* necessary to change from h_i to h_{i+1}; a migration swaps the images of two nodes u and v under h.
- $E_i = \{\sigma_1, \ldots, \sigma_i\}$ denotes the first i requests of σ interpreted as a set of edges on V, and $R(\sigma) = (V, E_m)$ denotes the *request graph* of σ.

In particular, we study the problem of designing a self-adjusting *linear network*: a network whose topologoy forms a d-dimenstional grid. We are particularly interested in the 1-dimensional grid in this paper, the line:

Definition 1 (Distributed List Update). *Let V, h, and σ be as before, with*

$$N = (\{1, \ldots, n\}, \{(1,2), (2,3), \ldots, (n-1, n)\}$$

representing a list graph. The cost of serving a $\sigma_i = (u, v) \in \sigma$ is given by $|h(u) - h(v)|$, i.e. the distance between u and v on N. Migrations can only occur between nodes configured on adjacent vertices in N.

Recall that the cost incurred by an algorithm A on σ is the sum of communication and reconfiguration costs. In the realm of online algorithms and competitive analysis, we compare an online algorithm *ON* to an offline algorithm *OFF* which has complete knowledge of σ ahead of time. We want to devise online algorithms *ON* which minimize the competitive ratio ρ:

$$\rho = \max_{\sigma} \frac{\text{cost}(\mathit{ON}(\sigma))}{\text{cost}(\mathit{OFF}(\sigma))}$$

As a first step, we in this paper consider the Distributed List Update problem for the case where the request graph $R(\sigma)$ has constant *graph bandwidth*: i.e. graphs for which there is a configuration in a line network such that any request can be served at constant cost. We refer to such a request graph as *linear demand*.

1.3 Contributions

This paper initiates the study of a most basic self-adjusting network, a line, which optimizes itself toward the dynamically changing linear demand, while amortizing reconfiguration cost. The underlying algorithmic problem is natural and motivated by emerging reconfigurable communication networks (e.g., based on virtual machine migration or novel optical technologies [10,17]). The problem can also be seen as a distributed version of the fundamental dynamic list update problem. Our first result is a negative one: we show that unlike the classic dynamic list update problem, which admits for constant-competitive online algorithms, there is an $\Omega(\log n)$ lower bound on the competitive ratio of any deterministic online algorithm for the distributed problem variant. Our second main contribution is a (distributed) online algorithm which is $\mathcal{O}(\log n)$-competitive for long enough sequences, given that the communication patterns exhibit linear demand.

1.4 Organization

The remainder of this paper is organized as follows. In Section 2, we put the problem and its challenges into perspective with respect to the list update problem. We then first derive the lower bound in Section 3 and present our algorithm and upper bound in Section 4. We conclude in Section 5.

2 From List Update to Distributed List Update

To provide an intuition of the challenges involved in designing online algorithms for distributed list update problems and to put the problem into perspective, we first revisit the classic list update problem and then discuss why similar techniques fail if applied to communicating node *pairs*, i.e., where requests not only come from the front of the list.

The *(dynamic) list update problem* [23] introduced by Sleater and Tarjan over 30 years ago is one of the most fundamental and oldest online problems: Given a set of n elements stored in a linked list, how to update the list over time such that it optimally serves a request sequence $\tau = (\tau_1, \tau_2, \ldots)$ where for each i, $\tau_i \in V$ is an arbitrary element stored in the list? The cost incurred by an algorithm is the sum of the access costs (i.e. scanning from the *front* of the list to the accessed element) and the number of *swaps* (switching two neighboring elements in the list). As accesses to the list elements start at the front of the list, it makes sense to amortize high access costs by moving frequently accessed elements closer to the front of the list. In fact, the well-known *Move-To-Front* (MTF) algorithm even moves an accessed element to the front *immediately*, and is known to be *constant competitive*: its cost is at most a factor 2 (or some other constant, depending on the cost model) worse than that of an optimal offline algorithm which knows the entire sequence τ ahead of time [23]. Throughout the literature, slightly different cost models have been used for the list update problem, though they only differ by a constant factor. Generally, a *cursor* is located at the head of the list at each request. Then, the algorithm can perform two operations, each operation incurring unit cost. i) *Move* the cursor to the left, or to the right, one position; the element in the new position is referred to as *touched*. ii) *Swap* the element at the cursor with the element one position to the left or right; the cursor also moves.

In the Distributed List Update problem, upon a request $\sigma_i = (s_i, t_i)$, the cursor is placed at s_i instead of the head of the list, and t_i needs to be looked up. To demonstrate the significance of this difference, we first present a paraphrased version of the proof by Tarjan and Sleator showing the dynamic optimality of MTF. After that, we showcase a simple access sequence differentiating the two problems.

2.1 An Expositional Proof for the Optimality of MTF

While the potential argument used to show dynamic optimality of the move-to-front strategy for the list access problem yields a very elegant and succinct proof [23], it lacks intuition which makes it difficult to generalise the argument. The key idea in the potential argument is to compare the execution of MTF to the execution of an arbitrary algorithm A. The algorithm is fixed for the analysis, but any valid algorithm can be used, e.g. the optimal offline algorithm. The state (represented by a list) of MTF and A are juxtaposed at every access, comparing how the order of elements in both lists differ. In fact, it is sufficient to only consider the relative order of two arbitrary but fixed elements, call them u and v. Consider the order of u and v in the state of A before it performs the ith access. If this order is the same as in MTF *before* it performs the ith acces, let $b_i = 0$ and otherwise $b_i = 1$.

Similarly, if their relative order is the same in MTF *after* its *ith* access, let $a_i = 0$ and otherwise $a_i = 1$. This describes an inversion sequence $b_1 a_1 b_2 a_2 \ldots b_m a_m$. Figure 1 illustrates this for MTF and an arbitrarily chosen algorithm A on a sequence $\tau = 6, 3, 1, 3, 6$, with the inversions of 1 and 6 described by the sequence 01111011100.

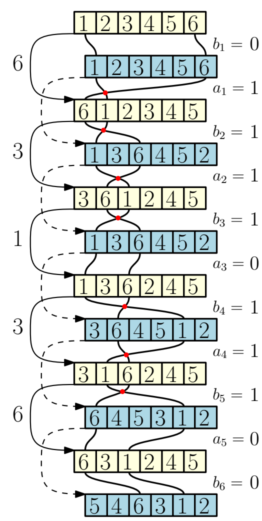


Fig. 1: MTF (yellow) and A (blue) on $\tau = 6, 3, 1, 3, 6$

Suppose that $\tau_i \in \{u, v\}$ and that MTF touches u and v while accessing τ_i. The proof by Tarjan and Sleator boils down to three observations.

Observation 1 *MTF inverts u and v relative to A by accessing τ_i, i.e. $b_i \neq a_i$.*

Observation 2 *If $b_i = 0$,* MTF *and A agree on the order of u and v before τ_i. Since* MTF *touches both, A also touches both in order to access τ_i.*

Observation 3 *For $b_i = 1$, let $j < i$ be the largest index such that $b_j = 0$ or $a_j = 0$ (note that j exists because $b_1 = 0$). When $a_j = 0$, and thus $b_{j+1} = 1$, A inverts u and v and therefore must have touched both. When $b_j = 0$, and thus $a_j = 1$,* MTF *inverts u and v and one of them is τ_j. By Observation 2, if $b_j = 0$ and* MTF *touches u and v to access τ_j, then A does as well.*

The last observation is essentially the amortised argument rephrased as a charging argument. We can now easily prove the dynamic optimality of MTF.

Theorem 1 (Tarjan & Sleator). MTF *is 4-competitive.*

Proof. We prove that for all $\tau_i = v$ where MTF touches u, there is a move by A touching u. MTF first moves the cursor to τ_i, and then swaps τ_i to the front. Along the way it touches u twice, once with a move and once with a swap, incurring a cost of 2.

For $b_i = 0$ (resp. $b_i = 1$), we use Observation 2 (resp. 3) to charge the cost to A touching u while accessing τ_i (resp. τ_j). By Observation 1, $b_i \neq a_i$, and thus for any $\tau_k \in \{u, v\}$ with $i < k$, the largest index $j' < k$ with $b_{j'} = 0$ or $a_{j'} = 0$ must be at least i, and therefore $j < i \leq j'$. This guarantees that MTF charges at most a cost of 4 to one move of A. Since all the cost incurred by MTF is charged to some move of A, the claim follows. □

In the original work by Tarjan and Sleator, MTF is shown to be 2-competitive. This is because their cost model allows accessed elements to be moved to the front 'for free'. If we allow this as well, the cursor touches u only once to access v, resulting in a factor 2.

2.2 The Challenge of Distributed List Update

Generalizing dynamic list update to Distributed List Update introduces a number of challenges which render the problem more difficult. First, the natural inversion argument no longer works: a reference point such as the front of the list is missing in the distributed setting. This makes it harder to relate algorithms to each other and hence also to define a potential. Second, for general request graphs $R(\sigma)$, an online algorithm needs to be able to essentially "recognize" certain patterns over time.

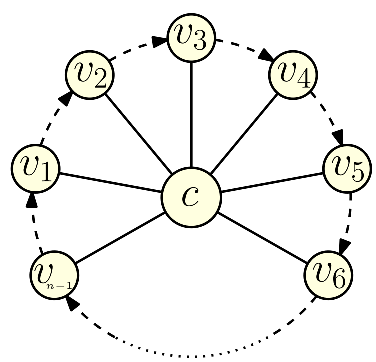


Fig. 2: A star graph used to construct a cyclic sequence of requests $\sigma_c = (c, v_1), (c, v_2), \ldots, (c, v_{n-1}), (c, v_1), \ldots$

Regarding the latter, consider the set of nodes $V = \{v_1, ..., v_n\}$ and let τ_c be a cyclic sequence: for all $\tau_i, \tau_{i+1} \in \tau_c$ with $\tau_i = v_j$ and $\tau_{i+1} = v_k$ it holds that $j+1 = k(\mod n-1)$. From this we construct a similar sequence σ_c for Distributed List Update on the set of nodes $V \cup \{c\}$, with $\sigma_i = (c, \tau_i)$. This yields a star graph $R(\sigma_c)$ as denoted in Figure 2. An offline algorithm can efficiently serve the cyclic order by first embedding the elements in the order $v_1, ...v_k$, and then moving the element c one position further after every request. If the cost of embedding the initial order is dominated by serving all requests, then the amortized cost is $\mathcal{O}(1)$ per request (per cycle there are $n-1$ moves of cost $\mathcal{O}(1)$ and once c is moved a distance n). However, in the list update model, any sequence cycling through all elements is a worst-case sequence with $\Omega(n)$ per request. This demonstrates that a "dynamic cursor" can mean a factor n difference in cost. What the sequence σ_c also demonstrates, is that aggregating elements around a highly communicative node is suboptimal; in the particular case of σ_c, it is this central node that needs to be moved.

Another pattern is a request sequence σ that forms a connected path in the request graph $R(\sigma)$. When restricted to only these patterns, Distributed List Update corresponds to the *Itinerant List Update Problem* (ILU) studied in [18]. In this work it is shown that deriving non-trivial upper bounds on the competitive ratio already seems notoriously hard (even offline approximation factors are relatively high). Note that the star example can be expressed as a path, i.e. $\sigma'_c = (c, v_1), (v_1, c), (c, v_2), (v_2, c), (c, v_3), \ldots$, demonstrating the significance of understanding simple request patterns for Distributed List Update. This is partly why in this paper we focus on request graphs with a linear demand.

3 A Lower Bound

This section derives a lower bound on the competitive ratio of any algorithm for Distributed List Update.

Theorem 2. *The competitive ratio* $\rho = \max_\sigma \frac{cost(\mathrm{ON}(\sigma))}{cost(\mathrm{OFF}(\sigma))}$ *for* Distributed List Update*, with* $|\sigma| = \Omega(n^2)$*, is at least* $\Omega(\log n)$*. This bound holds for arbitrarily long sequences, but if* $|\sigma| = \mathcal{O}(n^2)$*, it even holds if the request graph is a list graph.*

To prove this, we consider an arbitrary online algorithm *ON* for Distributed List Update. The main idea is to have an adaptive online adversary construct a sequence σ_{ON} that depends on the algorithm *ON*. The adversary constructs σ_{ON} so that the resulting request graph $R(\sigma_{ON})$ is a list graph. Because an offline algorithm knows $R(\sigma_{ON})$ in advance, it can immediately configure it and serve all requests at optimal cost of 1; since $|\sigma| = \Omega(n^2)$, the configuration cost of $\mathcal{O}(n^2)$ is negligable. We show that the online algorithm is forced to essentially reconfigure its layout $\log n$ times, resulting in the desired ratio. To facilitate our analysis, we use the same notion of the *distortion* of an embedding as is used in the Minimum Linear Arrangement (MLA) [14] problem.

Definition 2. *Given a request graph* $G = (V, E)$ *with* $E \subseteq V \times V$*, let* $E^+ = \{(u, v) \mid d_G(u, v) < \infty\}$ *denote the transitive closure of* E*.*
For $h \in C_{V \hookrightarrow N}$*, let* $\mathrm{d}_h(E)$ *denote the* distortion *of* E*, which is defined as:*

$$\mathrm{d}_h(E) = \sum_{(u,v) \in E^+} \mathrm{d}_h(u, v)$$

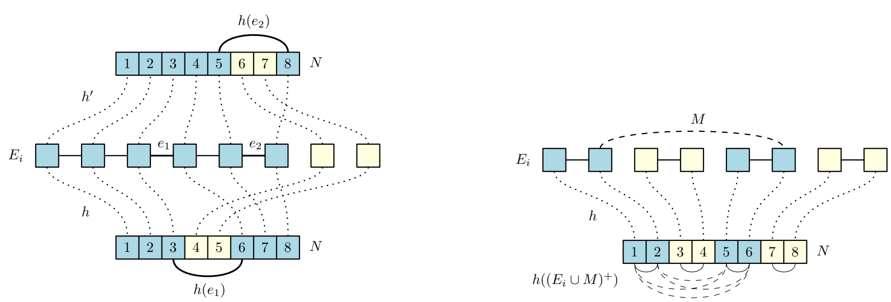
By summing over edges in E^+ (instead of E), the cost of a badly embedded edge $e \in E$ is essentially multiplied by the number of paths in E that contain e.

This means that the distortion of an embedding of a list is worse if the badly embedded edges occur in the middle of the list, see Figure 3a. To build σ_{ON}, the adversary gradually commits to the edges of $R(\sigma_{ON})$. Having already requested $\sigma_1, \ldots, \sigma_i$, then depending on the distortion the adversary:

Option 1: picks $\sigma_{i+1} = \arg\max_{(u,v)\in E_i} d_h(u, v)$.

Option 2: reveals a new batch of edges $M \subset V \times V$.

From these two options, the adversary's strategy becomes clear; Option 1 forces the highest possible cost to ON based on E_i and h, and Option 2 introduces new communication edges to force an increase in distortion. What is left to show is how the value of $d_h(E_i)$ comes into play, and which edges the adversary commits to. The adversary reveals at most $n - 1$ edges (since the final request graph is a list), and they will be revealed in batches of size $n/2$, $n/4$, $n/8$, etc., resulting in $\log n$ batches. After each batch, for ON to remain optimal it must permute its layout at cost $\Omega(n^2)$, totaling a cost of $\Omega(n^2 \log n)$ for all batches combined. To ensure that $R(\sigma_{ON})$ is a list graph, the partial request graph E_i (i.e., the set of revealed edges) always comprises a set of disjoint *sublists*. Therefore, the adversary only reveals edges that concatenate two sublists in E_i. Initially E_i is empty and the corresponding sublists are all singleton sets of $u \in V$.



(a) Two embeddings h and h' of a set of edges. Even though both embeddings embed only a single edge suboptimally, the distortion of $d_h(E)$ is bigger than $d_{h'}(E)$ because more paths in E_i cross e_1 than e_2.

(b) A visualization of $d_h(E_i \cup M)$: the list graph N, E_i (solid) and M (dashed) are sets of edges, configured on N by h (dotted). The sum of length of the configured edges $h((E_i \cup M)^+)$ is the distortion $d_h(E_i \cup M)$.

Fig. 3: Illustrations of distortion.

To help decide which edges to reveal, we use the distortion to associate a cost to batches of edges that the adversary can commit to. Let $M \subseteq V \times V \setminus E_i$ be any set of edges such that the graph $(V, E_i \cup M)$ comprises a set of disjoint sublists. For a configuration h of ON, the set M induces a distortion of $d_h(E_i \cup M)$, as shown in Figure 3b. We show that for any embedding that ON chooses, the adversary can find a set M so that the distortion is large.

Lemma 1. *Let N be a list graph, and $E \subseteq V \times V$ a set of edges so that the graph $G = (V, E)$ induces k disjoint sublists. For every $h \in C_{V \hookrightarrow N}$, there exists a set $M \subseteq V \times V$ of at most $k/2$ edges such that* $\mathrm{d}_h(E \cup M) = \Omega(\frac{n^3}{k})$ *and $(V, E \cup M)$ comprises a set of disjoint lists.*

To prove this lemma, we use the following fact (with proof in the full paper[6]):

Theorem 3. *Let $x_1, \ldots, x_k$ and $y_1, \ldots, y_k$ be sequences of k nonnegative numbers, and let x (resp. y) denote $\sum_{i=1}^{k} x_i$. Let the* weight *of an* involution[4] *over the indices $1, \ldots, k$ be defined as $w(f) = \sum_{i=1}^{k} x_i y_{f(i)}$.*

The average weight over all involutions is $\Omega(\frac{xy}{k})$.

Proof (Lemma 1). Let $L_1, ..., L_k \subseteq E$ be the sublists in G. For all pairs (i, j), let (L_i, L_j) denote any edge so that $L_i \cup L_j \cup \{(L_i, L_j)\} = L_i \oplus L_j$ is connected. For any involution f on the sublists we have:

$$2d_h(E \cup \{(L_i, L_{f(i)}) \mid i \neq f(i)\}) \geq \sum_{i=1}^{k} d_h(L_i \oplus L_{f(i)}). \tag{1}$$

The factor 2 is necessary because for i such that $i \neq f(i)$, the term $d_h(L_i \oplus L_{f(i)})$ appears twice in the sum. Now partition N into three sublists: a left part $X = \{1, \ldots, \lceil n/3 \rceil\}$, a right part $Y = \{\lfloor 2n/3 \rfloor, \ldots, n\}$, and the centre part $C = N \setminus (X \cup Y)$. Let $h_X(L_i)$ (resp. $h_Y(L_i)$) denote the number of elements of L_i that h maps onto X (resp. Y). Every two vertices u, v so that $h(u) \in X$ and $h(v) \in Y$ are by construction at least $|C| = \Theta(n)$ apart on N, and therefore we can lower bound $d_h(L_i \oplus L_j)$ by:

$$d_h(L_i \oplus L_j) \geq |C| \cdot h_X(L_i) h_Y(L_j) \tag{2}$$

For an involution f drawn uniformly at random, Theorem 3 gives us a bound on the expected value of the following:

$$\mathbf{E}\left(\sum_{i=1}^{k} h_X(L_i) h_Y(L_{f(i)})\right) = \Omega\left(\frac{\lceil n/3 \rceil^2}{k}\right) \tag{3}$$

Therefore, there exists an involution f for which we have:

$$\begin{aligned} 2d_h(E \cup \{(L_i, L_{f(i)}) \mid i \neq f(i)\}) &\overset{(1)}{\geq} \sum_{i=1}^{k} d_h(L_i \oplus L_{f(i)}) \\ &\overset{(2)}{\geq} |C| \cdot \sum_{i=1}^{k} h_X(L_i) h_Y(L_{f(i)}) \\ &\overset{(3)}{=} \Theta(n) \cdot \Omega(n^2/k) = \Omega\left(\frac{n^3}{k}\right) \end{aligned}$$

[4] A function f such that $f(f(x)) = x$ for all x.

Since this holds for any choice of (L_i, L_j), we can pick them so that $(V, E \cup \{(L_i, L_{f(i)}) \mid i \neq f(i)\})$ comprises a set of disjoint lists. □

This lemma (and the proof) reveals how the adversary commits to a new batch of edges in Option 2 (essentially a random matching will do). Observe that the number of edges is at most half the number of sublists in E_i. In the worst case we have to assume it is exactly half, and thus that the number of sublists is halved after every new batch of edges is selected. Next we show the precondition for the adversary to opt for Option 1, including a lower bound on the corresponding cost imposed on *ON*.

Lemma 2. *Let N be a list graph, $h \in C_{V \hookrightarrow N}$ a configuration, and $E \subseteq V \times V$ a set of edges so that the graph $G = (V, E)$ has n/ℓ disjoint sublists of size ℓ. If $\mathrm{d}_h(E) = \Omega(\ell n^2)$, then there exists an edge $(u, v) \in E$ such that $\mathrm{d}_h(u, v) = \Omega(n/\ell)$.*

Proof. There are at most $n/\ell \cdot \binom{\ell}{2} = \mathcal{O}(\ell n)$ distinct simple paths in G, meaning that the average distortion of these paths is $\frac{\Omega(\ell n^2)}{\mathcal{O}(\ell n)} = \Omega(n)$. The highest distortion is at least the average, and every path in G has length at most ℓ. On this path, there must exist an edge with distortion $\Omega(n/\ell)$, since if all edges have a distortion of $o(n/\ell)$, the total would be $o(n)$. □

Combined, Lemma 1 and Lemma 2 imply that the adversary can either request an edge at cost $\Omega(n/\ell)$, or increase the distortion to $\Omega(\ell n^2)$ by revealing a new batch of edges. The final ingredient is a lower bound on how much cost the adversary can impose on *ON* in between these batches.

Lemma 3. *Let N be a list graph, $E \subset V \times V$ a set of communication edges. If $h, h' \in C_{V \hookrightarrow N}$ are two embeddings that differ only in the order of two adjacent elements u and v, then $\mathrm{d}_h(E) \leq \mathrm{d}_{h'}(E) + 2\ell$, where ℓ is the size of the largest sublist in E.*

Proof. Consider all simple paths in E that end in u. At most ℓ paths ending in u (or v) are reduced by 1, and therefore $d_h(E)) - d_{h'}(E) \leq 2\ell$.

Combining the previous lemmata, we can prove the main technical result.

Lemma 4. *For every online algorithm A, there is a sequence σ_{ON} of length $\mathcal{O}(\varepsilon n^{1+\varepsilon} \log n)$ such that cost$(\mathrm{ON}(\sigma_{\mathrm{ON}})) = \Omega(\varepsilon n^2 \log n)$, for $0 < \varepsilon \leq 1$. Furthermore, the resulting request graph $R(\sigma_{\mathrm{ON}})$ is a list graph.*

Proof. W.l.o.g. assume that $n = 2^p$ for some integer p. This implies that the number of edges in every new batch is a power of 2; consequently, the sublists in any set E_i of revealed edges have size $2^k = \ell$ for some integer k.

Consider the situation right after a batch of edges is revealed, where all sublists have size ℓ. By Lemma 1 this implies that the distortion is $\Omega(\ell n^2)$. Let $\sigma = \sigma_i, \sigma_{i+1}, ..., \sigma_{i+\ell n}$ be the requests obtained by repeatedly requesting the edge in E_i with largest distortion. There are two situations:

- Throughout serving σ, the distortion is always at least $\Omega(\ell n^2)$. Then by Lemma 2 each σ_j, $i \leq j \leq i + \ell n$ incurred a cost of $\Omega(n/\ell)$, at total cost $\Omega(n^2)$.
- By serving σ, *ON* halves the distortion, thus reducing it by at least $\Omega(\ell n^2)$. Then, since by Lemma 3 every swap reduces the distortion by at most 2ℓ, *ON* must have used at least $\Omega(n^2)$ swaps.

This argument holds for each batch of edges revealed. The adversary stops when the sublists have size $2^{\varepsilon \log n}$, yielding a sequence σ_{ON} with

$$|\sigma_{ON}| = \sum_{\ell \in \{2^0, ..., 2^{\varepsilon \log n}\}} \ell n = \mathcal{O}(n^{1+\varepsilon})$$

and $cost(\sigma_{ON}) = \Omega(\varepsilon n^2 \log n)$. By Lemma 2, the adversary only requests edges that are introduced using the matching from Lemma 1. Any edge introduced by the latter Lemma concatenates two already existing sublists, hence $R(\sigma_{ON})$ is a list graph. □

To wrap up the proof for Theorem 2, we conclude by showing that for any online algorithm *ON*, the sequence σ_{ON} can be solved in $\mathcal{O}(n^2)$ by an optimal offline algorithm.

Proof (Proof of Theorem 2). Let *ON* be any online algorithm solving Distributed List Update. Apply Lemma 4 with $\varepsilon = 1/2$, yielding $\text{cost}(ON(\sigma_{ON})) = \Omega(n^2 \log n)$. Since σ_{ON} is a list graph, an offline algorithm can embed this graph at (worst case optimal) cost $\Theta(n^2)$, and serve every request at optimal cost $\mathcal{O}(1)$. This yields $\text{cost}(OFF(\sigma_{ON})) = \Theta(n^2)$, and thus

$$\rho = \frac{\text{cost}(ON(\sigma))}{\text{cost}(OFF(\sigma))} = \Omega(\log n)$$

In order to make this bound hold for arbitrary long sequences, we slightly modify the adversary. After every $\mathcal{O}(n^2)$ requests it serves, it can reconfigure to a new list at cost $\mathcal{O}(n^2)$, and repeat the argument to force cost of $\Omega(n^2 \log n)$ to *ON* for the subsequent $\mathcal{O}(n^2)$ requests.

Remark. We can generalise the model for Distributed List Update to include cases where both the request graph and the host graph G are a d-dimensional grid, for constant d; we dub this problem Distributed Grid Update. On a request (u, v), the cursor is placed at u and the request is served when it touches v. The same operations are allowed: **moving** the cursor, or **swapping** with on of its 2^d neighbors (also moving the cursor).

Lemma 5. *For every online algorithm* ON *for* Distributed Grid Update, *there is a sequence* σ_{ON} *of length* $\mathcal{O}(\varepsilon n^{1+\varepsilon}\log n)$ *such that cost*$(\text{ON}(\sigma_{\text{ON}})) = \Omega(\varepsilon n^{1+1/d}\log n)$, *for* $0 < \varepsilon \leq 1$. *The resulting request graph* $R(\sigma_{\text{ON}})$ *is a d-dimensional grid graph.*

The proof of Lemma 5 is essentially identical to that of Lemma 4. An overview of the necessary modifications are given in the full paper[6].

4 An Upper Bound

This section presents a $\mathcal{O}(\log n)$-competitive online algorithm for Distributed List Update. Our main technical lemma shows that the total cost spent on learning the optimal embedding never exceeds $\mathcal{O}(n^2 \log n)$. We propose a simple greedy algorithm that identifies a *locally optimal* embedding, and always moves towards this embedding. Observe that a set of k sublists can be embedded perfectly on a line graph in at most $2^k k!$ ways (they are permuted in some order, and every list has at most two orientations). Given a configuration $h \in C_{V \hookrightarrow N}$ of the lists, we define the locally optimal embedding to be an optimal embedding one that takes the fewest number of reconfigurations to reach, starting at h. Formally, if $opt(E)$ is the set of optimal embeddings of a set edges, then the h-optimal embedding of E is

$$h[E] = \underset{h' \in opt(E)}{\arg\min} \sum_{v \in V} |h(v) - h'(v)|$$

With such a configuration we associate the cost:

$$\Phi_h[E] = \sum_{v \in V} |h(v) - h[E](v)|$$

Let Gread be the algorithm (it GREedily ADjoins sublists), that upon seeing a new edge σ_i, *immediately* moves to the embedding $h[E_i \cup \{\sigma_{i+1}\}]$.

For each E_i, let $\mathcal{V}(E_i)$ be the connected components of (V, E_i), so that $\mathcal{V}_\sigma = \cup_{1 \leq i \leq m} \mathcal{V}(E_i)$ is the set of all sublists induced by σ. This naturally defines a binary tree $T_\sigma = (\mathcal{V}_\sigma, E_\sigma)$: for every first occurence σ_i of $(u, w) \in E_m$ connecting two sublists U, W in $R(E_i)$, there are two corresponding edges $(U, U \cup W)$ and $(W, U \cup W)$ in E_σ.For every $\sigma_i \in E_m$, Gread incurs some cost for reconfiguring, and the following lemma bounds this cost.

Lemma 6. *Let h be an optimal embedding of E_i, and let σ_{i+1} be an edge connecting two sublists U and W of E_i. It holds that*

$$\Phi_h[E_i \cup \{\sigma_{i+1}\}] \leq n \cdot \min(|U|, |W|)$$

Proof. Since E_i is optimally embedded by h, we simply need to move the smaller of U and W into its correct location so that $E_i \cup \{\sigma_{i+1}\}$ is optimally embedded. This requires every element in the smaller list to be moved at most n locations, therefore $\Phi_h[E_i \cup \{\sigma_{i+1}\}] \leq n \min(|U|, |W|)$.

For a node $U \in \mathcal{V}_\sigma$, let left(U) and right(U) denote U's left and right child respectively. Further, let $w(U)$ denote the number of nodes in the subtree rooted at U. Observe that for any binary tree with nodes N, it holds that

$$\sum_{v \in N} \min(w(\text{left}(v)), w(\text{right}(v))) \leq |N| \log |N|$$

Theorem 4. *For any σ, with $|\sigma| = m$, such that $|E_m| = k$ and $R(\sigma)$ is a list,*

$$\textit{cost}(\textsc{Gread}(\sigma)) = \mathcal{O}(m + nk \log k)$$

Proof. Let h_i denote the configuration after request σ_1, and let h_0 denote the trivial optimal initial embedding. Then the total cost of Gread is the sum of reconfiguring after every σ_i plus accessing every request at cost 1:

$$\begin{aligned}
\text{cost}(\textsc{Gread}(\sigma)) - m &= \sum_{i=0}^{m} \Phi_{h_i}[E_i \cup \{\sigma_{i+1}\}] \\
&\leq \sum_{U \in \mathcal{V}_\sigma} n \min(w(\text{left}(U)), w(\text{right}(U))) \\
&\leq nk \log k
\end{aligned}$$

As a corollary, it is not hard to show that Gread achieves optimal $\log n$ competitiveness for the worst case sequence constructed in Section 3. Additionally, in the full paper [6] we show a distributed implementation of this algorithm using message passing.

5 Conclusion

We presented a first and asymptotically tight, i.e., $\Theta(\log n)$-competitive online algorithm for self-adjusting reconfigurable linear networks with linear demand. Both our lower and upper bounds are non-trivial, and we believe that our work opens several interesting directions for future research. In particular, it would be very interesting to shed light on the competitive ratio achievable in more general network topologies, and to study randomized algorithms.

References

1. Albers, S.: A competitive analysis of the list update problem with lookahead. Theoretical Computer Science **197**(1-2), 95–109 (1998)

2. Albers, S.: Improved randomized on-line algorithms for the list update problem. SIAM Journal on Computing **27**(3), 682–693 (1998)
3. Albers, S., Von Stengel, B., Werchner, R.: A combined bit and timestamp algorithm for the list update problem. Information Processing Letters **56**(3), 135–139 (1995)
4. Albers, S., Westbrook, J.: Self-organizing data structures. In: Online algorithms, pp. 13–51. Springer (1998)
5. Ambühl, C.: Offline list update is np-hard. In: European Symposium on Algorithms. pp. 42–51. Springer (2000)
6. Avin, C., van Duijn, I., Schmid, S.: Self-adjusting linear networks. arXiv preprint arXiv:1905.02472 (2019)
7. Avin, C., Haeupler, B., Lotker, Z., Scheideler, C., Schmid, S.: Locally self-adjusting tree networks. In: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing. pp. 395–406. IEEE (2013)
8. Avin, C., Hercules, A., Loukas, A., Schmid, S.: Towards communication-aware robust topologies. ArXiv Technical Report (2017)
9. Avin, C., Loukas, A., Pacut, M., Schmid, S.: Online balanced repartitioning. In: Proc. 30th International Symposium on Distributed Computing (DISC) (2016)
10. Avin, C., Mondal, K., Schmid, S.: Demand-aware network designs of bounded degree. In: Proc. International Symposium on Distributed Computing (DISC) (2017)
11. Avin, C., Mondal, K., Schmid, S.: Demand-aware network design with minimal congestion and route lengths. In: Proc. IEE INFOCOM (2019)
12. Avin, C., Schmid, S.: Toward demand-aware networking: A theory for self-adjusting networks. In: ACM SIGCOMM Computer Communication Review (CCR) (2018)
13. Bubeck, S., Cesa-Bianchi, N., et al.: Regret analysis of stochastic and nonstochastic multi-armed bandit problems. Foundations and Trends® in Machine Learning **5**(1), 1–122 (2012)
14. Díaz, J., Petit, J., Serna, M.: A survey of graph layout problems. ACM Computing Surveys (CSUR) **34**(3), 313–356 (2002)
15. Fenz, T., Foerster, K.T., Schmid, S., Villedieu, A.: Efficient non-segregated routing for reconfigurable demand-aware networks. In: Proc. IFIP Networking (2019)
16. Huq, S., Ghosh, S.: Locally self-adjusting skip graphs. In: Proc. IEEE 37th International Conference on Distributed Computing Systems (ICDCS). pp. 805–815 (2017)
17. M. Ghobadi et al.: Projector: Agile reconfigurable data center interconnect. In: Proc. ACM SIGCOMM. pp. 216–229 (2016)
18. Olver, N., Pruhs, K., Schewior, K., Sitters, R., Stougie, L.: The itinerant list update problem. In: 13th Workshop on Models and Algorithms for Planning and Scheduling Problems. p. 163 (2017)
19. Peres, B., Souza, O., Goussevskaia, O., Schmid, S., Avin, C.: Distributed self-adjusting tree networks. In: Proc. IEE INFOCOM (2019)
20. Reingold, N., Westbrook, J.: Off-line algorithms for the list update problem. Information Processing Letters **60**(2), 75–80 (1996)
21. Reingold, N., Westbrook, J., Sleator, D.D.: Randomized competitive algorithms for the list update problem. Algorithmica **11**(1), 15–32 (1994)
22. Schmid, S., Avin, C., Scheideler, C., Borokhovich, M., Haeupler, B., Lotker, Z.: Splaynet: Towards locally self-adjusting networks. IEEE/ACM Transactions on Networking (ToN) (2016)
23. Sleator, D.D., Tarjan, R.E.: Amortized efficiency of list update and paging rules. Communications of the ACM **28**(2), 202–208 (1985)
24. Teia, B.: A lower bound for randomized list update algorithms. Information Processing Letters **47**(1), 5–9 (1993)