# Dynamically Optimal Self-Adjusting Single-Source Tree Networks

Chen Avin[1], Kaushik Mondal[2], and Stefan Schmid[3]

[1] Ben Gurion University of the Negev, Israel
[2] Indian Institute of Technology Ropar, India
[3] Faculty of Computer Science, University of Vienna, Austria

**Abstract.** This paper studies a fundamental algorithmic problem related to the design of demand-aware networks: networks whose topologies adjust toward the traffic patterns they serve, in an online manner. The goal is to strike a tradeoff between the benefits of such adjustments (shorter routes) and their costs (reconfigurations). In particular, we consider the problem of designing a self-adjusting tree network which serves single-source, multi-destination communication. The problem has interesting connections to self-adjusting datastructures. We present two constant-competitive online algorithms for this problem, one randomized and one deterministic. Our approach is based on a natural notion of *Most Recently Used (MRU)* tree, maintaining a *working set*. We prove that the working set is a cost lower bound for any online algorithm, and then present a randomized algorithm Random-Push which *approximates* such an MRU tree at low cost, by pushing less recently used communication partners down the tree, along a random walk. Our deterministic algorithm Move-Half does not directly maintain an MRU tree, but its cost is still proportional to the cost of an MRU tree, and also matches the working set lower bound.

## 1 Introduction

While datacenter networks traditionally rely on a *fixed* topology, recent optical technologies enable *reconfigurable* topologies which can adjust to the demand (i.e., traffic pattern) they serve *in an online manner*, e.g. [1,2,3,4]. Indeed, the physical topology is emerging as the next frontier in an ongoing effort to render networked systems more flexible.

In principle, such topological reconfigurations can be used to provide shorter routes between frequently communicating nodes, exploiting structure in traffic patterns [2,5,6], and hence to improve performance. However, the design of self-adjusting networks which dynamically optimize themselves toward the demand introduces an algorithmic challenge: an online algorithm needs to be devised which guarantees an efficient tradeoff between the benefits (i.e., shorter route lengths) and costs (in terms of reconfigurations) of topological optimizations.

This paper focuses on the design of a self-adjusting *complete tree (*CT*)* network: a network of nodes (e.g., servers or racks) that forms a complete tree, and

we measure the routing cost in terms of the length of the shortest path between two nodes. Trees are not only a most fundamental topological structure of their own merit, but also a crucial building block for more general self-adjusting network designs: Avin et al. [7,8] recently showed that multiple tree networks (optimized individually for a single source node) can be combined to build general networks which provide low degree and low distortion. The design of a dynamic single-source multi-destination communication tree, as studied in this paper, is hence a stepping stone.

The focus on trees is further motivated by a relationship of our problem to problems arising in self-adjusting datastructures [9]: self-adjusting datastructures such as self-adjusting search trees [10] have the appealing property that they optimize themselves to the workload, leveraging temporal locality, but without knowing the future. Ideally, self-adjusting datastructures store items which will be accessed (frequently) *in the future*, in a way that they can be accessed quickly (e.g., close to the root, in case of a binary search tree), while also accounting for reconfiguration costs. However, in contrast to most datastructures, in a *network*, the search property is not required: the network supports *routing*. Accordingly our model can be seen as a novel flavor of such self-adjusting binary search trees where lookup is supported by a *map*, enabling shortest path routing (more details will follow).

We present a formal model for this problem later, but a few observations are easy to make. If we restrict ourselves to the special case of a *line* network (a "linear tree"), the problem of optimally arranging the destinations of a given single communication source is equivalent to the well-known *dynamic list update* problem: for such self-adjusting (unordered) lists, dynamically optimal online algorithms have been known for a long time [11]. In particular, the simple move-to-front algorithm which immediately promotes the accessed item to the front of the list, fulfills the *Most-Recently Used (MRU)* property: the $i^{th}$ furthest away item from the front of the list is the $i^{th}$ most recently used item. In the list (and hence on the line), this property is enough to guarantee optimality. The MRU property is related to the so called *working set property*: the cost of accessing item $x$ at time $t$ depends on the number of distinct items accessed since the last access of $x$ prior to time $t$, including $x$. Naturally, we wonder whether the *MRU* property is enough to guarantee optimality also in our case. The answer turns out to be non-trivial.

A first contribution of this paper is the observation that if we count only *access* cost (ignoring any rearrangement cost, see Definition 1 for details), the answer is affirmative: the most-recently used tree is what is called *access optimal*. Furthermore, we show that the corresponding access cost is a lower bound for any algorithm which is dynamically optimal. But securing this property, i.e., maintaining the most-recently used items close to the root in the tree, introduces a new challenge: how to achieve this *at low cost*? In particular, assuming that *swapping* the locations of items comes at a *unit cost*, can the property be maintained at cost proportional to the *access* cost? As we show, *strictly* enforcing the most-recently used property in a tree is too costly to achieve optimality. But,

as we will show, when turning to an *approximate* most-recently used property, we are able to show two important properties: *i)* such an approximation is good enough to guarantee access optimality; and *ii)* it can be maintained in expectation using a *randomized* algorithm: less recently used communication partners are pushed down the tree along a random walk.

While the most-recently used property is *sufficient*, it is not necessary: we provide a deterministic algorithm which is dynamically optimal but does not even maintain the MRU property approximately. However, its cost is still proportional to the cost of an MRU tree (Definition 4).

Succinctly, we make the following ***contributions***. First we show a working set lower bound for our problem. We do so by proving that an MRU tree is *access optimal*. In the following theorem, let $WS(\sigma)$ denote the working set of $\sigma$ (a formal definition will follow later).

**Theorem 1.** *Consider a request sequence $\sigma$. Any algorithm* Alg *serving $\sigma$ using a self-adjusting complete tree, has cost at least* $\text{cost}(\text{Alg}(\sigma)) \geq WS(\sigma)/4$*, where $WS(\sigma)$ is the working set of $\sigma$.*

Our main contribution is a deterministic online algorithm Move-Half which maintains a constant competitive self-adjusting Complete Tree (CT) network.

**Theorem 2.** Move-Half *algorithm is dynamically optimal.*

Interestingly, Move-Half does not require the MRU property and hence does not need to maintain MRU tree. This implies that maintaining a working set on CTs is not a necessary condition for dynamic optimality, although it is a sufficient one.

Furthermore, we present a dynamically optimal, i.e., constant competitive (on expectation) randomized algorithm for self-adjusting CTs called Random-Push. Random-Push relies on maintaining an approximate MRU tree.

**Theorem 3.** *The* Random-Push *algorithm is dynamically optimal on expectation.*

Due to space constraints, proofs and longer discussions appear in a technical report [12].

## 2 Model and Preliminaries

Our problem can be formalized using the following simple model. We consider a single *source* that needs to communicate with a set of $n$ nodes $V = \{v_1, \ldots, v_n\}$. The nodes are arranged in a complete binary tree and the source is connected the the root of the tree. While the tree describes a reconfigurable *network*, we will use terminology from datastructures, to highlight this relationship and avoid the need to introduce new terms.

We consider a complete tree $T$ connecting $n$ servers $S = \{s_1, \ldots, s_n\}$. We will denote by $s_1(T)$ the root of the tree $T$, or $s_1$ when $T$ is clear from the context,

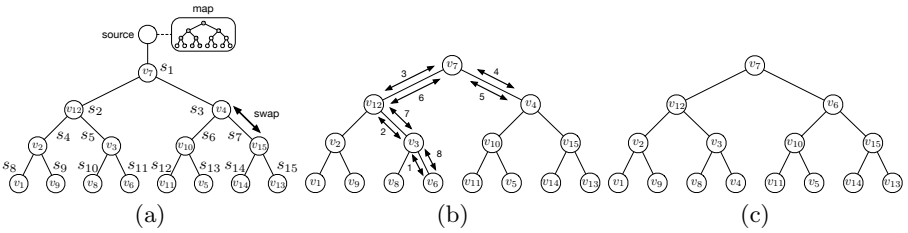


Fig. 1: (a) Our *complete tree* model: a source with a map, a tree of servers that host items (nodes) and a *swap* operation between neigboring items. (b) The node's tree network implied by the tree $T$ from (a) and the set of swaps needed to interchange the location of $v_6$ and $v_4$. (c) The tree network after the interchange and swap operations of (b).

and by $s_i$.left (resp. $s_i$.right) the left (resp. right) child of server $s_i$. We assume that the $n$ servers store $n$ items (nodes) $V = \{v_1, \ldots, v_n\}$, one item per server. For any $i \in [1, n]$ and any time $t$, we will denote by $s_i.\text{guest}^{(t)} \in V$ the item mapped to $s_i$ at time $t$. Similarly, $v_i.\text{host}^{(t)} \in S$ denotes the server hosting item $v_i$. Note that if $v_i.\text{host}^{(t)} = s_j$ then $s_j.\text{guest}^{(t)} = v_i$.

The *depth* of a server $s_i$ is fixed and describes the distance from the root; it is denoted by $s_i$.dep, and $s_1.\text{dep} = 0$. The depth of an item $v_i$ *at time* $t$ is denoted by $v_i.\text{dep}^{(t)}$, and is given by the depth of the server to which $v_i$ is mapped at time $t$. Note that $v_i.\text{dep}^{(t)} = v_i.\text{host}.\text{dep}^{(t)}$.

To this end, we interpret communication requests from the source as *accesses* to *items* stored in the (unordered) tree. All access requests (resp. communication requests) to items (resp. nodes) originate from the root $s_1$. If an item (resp. node) is frequently requested, it can make sense to move this item (node) closer to the root of $T$: this is achieved by *swapping* items which are neighboring in the tree (resp. by performing local topological swaps).

Access requests occur over time, forming a (finite or infinite) sequence $\sigma = (\sigma^{(1)}, \sigma^{(2)}, \ldots)$, where $\sigma^{(t)} = v_i \in V$ denotes that item $v_i$ is requested, and needs to be accessed at time $t$. The sequence $\sigma$ (henceforth also called the *workload*) is revealed one-by-one to an online algorithm On. The *working set* of an item $v_i$ at time $t$ is the set of distinct elements accessed since the last access of $v_i$ prior to time $t$, including $v_i$. We define the *rank* of item $v_i$ at time $t$ to be the size of the working set of $v_i$ at time $t$ and denote it as $v_i.\text{rank}^{(t)}$. When $t$ is clear of context, we simply write $v_i$.rank. The working set bound of sequence $\sigma$ of $m$ requests is defined as $WS(\sigma) = \sum_{t=1}^{m} \log(\sigma^{(t)}.\text{rank})$.

Both serving (i.e., *routing*) the request and adjusting the configuration comes at a cost. We will discuss the two cost components in turn. Upon a request, i.e., whenever the source wants to communicate to a partner, it routes to it via the tree $T$. To this end, a message passed between nodes can include, for each node it passes, a bit indicating which child to forward the message next (requires $O(\log n)$ bits). Such a *source routing* header can be built based on a dynamic

global *map* of the tree that is maintained at the source node. As mentioned, the source node is a direct neighbor of the root of the tree, aware of all requests, and therefore it can maintain the map. The *access cost* is hence given by the distance between the root and the requested item, which is basically the depth of the item in the tree.

The *reconfiguration cost* is due to the adjustments that an algorithm performs on the tree. We define the unit cost of reconfiguration as a *swap*: a swap means changing position of an item with its parent. Note that, any two items $u, v$ in the tree can be *interchanged* using a number of swaps equal to twice the distance between them. This can be achieved by $u$ first swapping along the path to $v$ and then $v$ swapping along the same path to initial location of $u$. This interchange operation results in the tree staying the same, but only $u$ and $v$ changing locations. We assume that to interchange items, we first need to access one of them. See Figure 1 for an example of our model and interchange operation.

**Definition 1 (Cost).** *The cost incurred by an algorithm* Alg *to serve a request* $\sigma^{(t)} = v_i$ *is denoted by* $\text{cost}(\text{Alg}(\sigma^{(t)}))$*, short* $\text{cost}^{(t)}$*. It consists of two parts,* access *cost, denoted* $\text{acc-cost}^{(t)}$*, and* adjustment *cost, denoted* $\text{adj-cost}^{(t)}$*. We define access cost simply as* $\text{acc-cost}^{(t)} = v_i.\text{dep}^{(t)}$ *since* Alg *can maintain a global* map *and access* $v_i$ *via the shortest path. Adjustment cost,* $\text{adj-cost}^{(t)}$*, is the total number of swaps, where a single swap means changing position of an item with its parent or a child. The total cost, incurred by* Alg *is then*

$$\text{cost}(\text{Alg}(\sigma)) = \sum_t \text{cost}(\text{Alg}(\sigma^{(t)})) = \sum_t \text{cost}^{(t)} = \sum_t (\text{acc-cost}^{(t)} + \text{adj-cost}^{(t)})$$

Our main objective is to design online algorithms that perform almost as well as optimal offline algorithms (which know $\sigma$ ahead of time), even in the worst-case. In other words, we want to devise online algorithms which minimize the competitive ratio:

**Definition 2 (Competitive Ratio $\rho$).** *We consider the standard definition of (strict) competitive ratio* $\rho$*, i.e.,* $\rho = \max_\sigma \text{cost}(\text{On})/\text{cost}(\text{Opt})$ *where* $\sigma$ *is any input sequence and where* Opt *denotes the optimal offline algorithm.*

If an online algorithm is constant competitive, independently of the problem input, it is called *dynamically optimal*.

**Definition 3 (Dynamic Optimality).** *An (online) algorithm* On *achieves* dynamic optimality *if it asymptotically matches the offline optimum on every access sequence. In other words, the algorithm* On *is* $O(1)$*-competitive.*

We also consider a weaker form of competitivity (similarly to the notion of *search-optimality* in related work [13]), and say that On is *access-competitive* if we consider only the access cost of On (and ignore any adjustment cost) when comparing it to Opt (which needs to pay both for access and adjustment). For a randomized algorithm, we consider an oblivious online adversary which does not know the random bits of the online algorithm a priori.

The ***Self-adjusting Complete Tree Problem*** considered in this paper can then be formulated as follows: Find an online algorithm which serves any (finite or infinite) online request sequence $\sigma$ with minimum cost (including both access and rearrangement costs), on a self-adjusting complete binary tree.

## 3 Access Optimality: A Working Set Lower Bound

For *fixed* trees, it is easy to see that keeping frequent items close to the root, i.e., using a *Most-Frequently Used* (MFU) policy, is optimal (cf. the technical report [12]). The design of online algorithms for *adjusting* trees is more involved. In particular, it is known that MFU is not optimal for lists [11]. A natural strategy could be to try and keep items close to the root which have been frequent "recently". However, this raises the question over which time interval to compute the frequencies. Moreover, changing from one MFU tree to another one may entail high adjustment costs.

This section introduces a natural *pendant* to the MFU tree for a dynamic setting: the *Most Recently Used (MRU) tree*. Intuitively, the MRU tree tries to keep the "working set" resp. *recently* accessed items close to the root. In this section we show a working set lower bound for any self-adjusting complete binary tree.

While the move-to-front algorithm, known to be dynamically optimal for self-adjusting lists [11], naturally provides such a "most recently used" property, generalizing move-to-front to the tree is non-trivial. We therefore first show that any algorithm that maintains an MRU tree is *access-competitive*. With this in mind, let us first formally define the MRU tree.

**Definition 4 (MRU Tree).** *For a given time $t$, a tree $T$ is an* MRU *tree if and only if,*

$$v_i.\text{dep} = \lfloor \log v_i.\text{rank} \rfloor \tag{1}$$

Accordingly the root of the tree (level zero) will always host an item of rank one. More generally, servers in level $i$ will host items that have a rank between $(2^i, 2^{i+1} - 1)$. Upon a request of an item, say $v_j$ with rank $r$, the rank of $v_j$ is updated to one, and only the ranks of items with rank smaller than $r$ are increased, each by 1. Therefore, the rank of items with rank higher than $r$ do not change and their level (i.e., depth) in the MRU tree remains the same (but they may switch location within the same level).

**Definition 5 (MRU algorithm).** *An online algorithm* On *has the* MRU *property (or the working set property) if for each time $t$, the tree $T^{(t)}$ that* On *maintains, is an* MRU *tree.*

The working set lower bound will follow from the following theorem which states that any algorithm that has the *MRU* property is *access competitive*.

**Theorem 4.** *Any online algorithm* On *that has the* MRU *property is 4 access-competitive.*

Recall that an analogous statement of Theorem 4 is known to be true for a *list* [11]. As such, one would hope to find a simple proof that holds for complete trees, but it turns out that this is not trivial, since OPT has more freedom in trees. We therefore present a direct proof based on a potential function, similar in spirit to the list case.

Based on Theorem 4 we can now show our working set lower bound:

**Theorem 1.** *Consider a request sequence $\sigma$. Any algorithm* ALG *serving $\sigma$ using a self-adjusting complete tree, has cost at least* $\text{cost}(\text{ALG}(\sigma)) \geq WS(\sigma)/4$*, where $WS(\sigma)$ is the working set of $\sigma$.*

*Proof.* The sum of the access costs of items from an MRU tree is exactly $WS(\sigma)$. For the sake of contradiction assume that there is an algorithm ALG with cost $\text{cost}(\text{ALG}(\sigma)) < WS(\sigma)/4$. If follows that Theorem 4 is not true. A contradiction. □

## 4 Deterministic Algorithm

### 4.1 Efficiently Maintaining an MRU Tree

It follows from the previous section that if we can maintain an MRU tree at the cost of *accessing* an MRU tree, we will have a dynamically optimal algorithm. So we now turn our attention to the problem of efficiently maintaining an MRU tree. To achieve optimality, we need that the tree adjustment cost will be proportional to the access cost. In particular, we aim to design a tree which on one hand achieves a good approximation of the MRU property to capture temporal locality, by providing fast *access* (resp. *routing*) to items; and on the other hand is also adjustable at low cost over time.

Let us now assume that a certain item $\sigma^{(t)} = u$ is accessed at some time $t$. In order to re-establish the (strict) MRU property, $u$ needs to be promoted to the root. This however raises the question of where to move the item currently located at the root, let us call it $v$. A natural idea to make space for $u$ at the root while preserving locality, is to *push down* items from the root, including item $v$. However, note that simply pushing items down along the path between $u$ and $v$ (as done in lists) will result in a poor performance in the tree. To see this, let us denote the sequence of items along the path from $u$ to $v$ by $P = (u, w_1, w_2, \ldots, w_\ell, v)$, where $\ell = u.\text{dep}$, *before* the adjustment. Now assume that the access sequence $\sigma$ is such that it repeatedly cycles through the sequence $P$, in this order. The resulting cost per request is in the order of $\Theta(\ell)$, i.e., could reach $\Theta(\log n)$ for $\ell = \Theta(\log n)$. However, an algorithm which assigns (and then fixes) the items in $P$ to the top $\log \ell$ levels of the tree, will converge to a cost of only $\Theta(\log \ell) \in O(\log \log n)$ per request: an exponential improvement.

Another basic idea is to try and keep the MRU property at every step. Let us call this strategy MAX-PUSH. Consider a request to item $u$ which is at depth $u.\text{dep} = k$. Initially $u$ is moved to the root. Then the MAX-PUSH strategy chooses for each depth $i < u.\text{dep}$, the *least* recently accessed (and with maximum rank)

**Algorithm 1:** Upon request to $u$ in Move-Half's Tree

1: **access** $u = s$.guest along the tree branches (cost: $u$.dep)
2: let $v$ be the item with the highest rank at depth $\lfloor u.\text{dep}/2 \rfloor$
3: **swap** $u$ along tree branches to node $v$ (cost: $\frac{3}{2}u$.dep)
4: **swap** $v$ along tree branches to server $s$ (cost: $\frac{3}{2}u$.dep)

item from level $i$: formally, $w_i = \arg\max_{v \in V : v.\text{dep}=i} v.\text{rank}$. We then push $w_i$ to the host of $w_{i+1}$. It is not hard to see that this strategy will actually maintain a perfect MRU tree. However, items with the maximum rank in different levels, i.e., $w_i$.host and $w_{i+1}$.host, may not be in a parent-child relation. So to push $w_i$ to $w_{i+1}$.host, we may need to travel all the way from $w_i$.host to the root and then from the root to $w_{i+1}$.host, resulting in a cost proportional to $i$ per level $i$. This accumulates a rearrangement cost of $\sum_{i=1}^{k} i > k^2/2$ to push all the items with maximum rank at each layer, up to layer $k$. This is not proportional to the original access cost $k$ of the requested item and therefore, leads to a non-constant competitive ratio as high as $\Omega(\log n)$.

Later, in Section 5, we will present a randomized algorithm that maintains a tree that approximates an MRU tree at a low cost. But first, we will present a simple deterministic algorithm that does not directly maintain an MRU tree, but has cost that is proportional to the MRU cost and is hence dynamically optimal.

### 4.2 The Move-Half Algorithm

In this section we propose a simple deterministic algorithm, Move-Half, that is proven to be dynamically optimal. Interestingly Move-Half does not maintain the MRU property but its cost is shown to be competitive to the *access cost* on an MRU tree, and therefore, to the working set lower bound.

Move-Half is described in Algorithm 1. Initially, Move-Half and Opt start from the same tree (which is assumed w.l.o.g. to be an MRU tree). Then, upon a request to an item $u$, Move-Half first accesses $u$ and then interchanges its position with node $v$ that is the highest ranked item positioned at half of the depth of $u$ in the tree. After the interchange the tree remains the same, only $u$ and $v$ changed locations. See Figure 1 (b) for an example of Move-Half operation where $v_6$ at depth 3 is requested and is then interchanged with $v_4$ at depth 1 (assuming it is the highest rank node in level 1).

The *access cost* of Move-Half is proportional to the access cost of an MRU tree.

**Theorem 5.** *Algorithm* Move-Half *is 4 access-competitive to an MRU algorithm.*

**Theorem 2.** Move-Half *algorithm is dynamically optimal.*

*Proof (Proof of Theorem 2).* Using Theorem 4 and Theorem 5, MOVE-HALF is 16-access competitive. It is easy to see from Algorithm 1 that total cost of MOVE-HALF's tree is 4 times the access cost. Considering these, MOVE-HALF is 64-competitive. □

In the coming section we show techniques to maintain MRU trees cheaply. This is another way to maintain dynamic optimality.

## 5 Randomized MRU Trees

The question of how, and if at all possible, to maintain an MRU tree deterministically (where for each request $\sigma^{(t)}$, $\sigma^{(t)}.depth = \lfloor \log \sigma^{(t)}.\text{rank} \rfloor$) at low cost is still an open problem. But, in this section we show that the answer is affirmative with two relaxations: namely by using randomization and approximation. We believe that the properties of the algorithm we describe next may also find applications in other settings, and in particular data structures like skip lists [14].

At the heart of our approach lies an algorithm to maintain a constant approximation of the MRU tree at any time. First we define MRU($\beta$) trees for any constant $\beta$.

**Definition 6 (MRU($\beta$) Tree).** *A tree $T$ is called an* MRU($\beta$) *tree if it holds for any item $u$ and any time that,* $u.\text{dep} = \lfloor \log u.\text{rank} \rfloor + \beta$.

Note that, any MRU(0) tree is also an MRU tree. In particular, we prove in the following that a constant additive approximation is sufficient to obtain dynamic optimality.

**Theorem 6.** *Any online MRU($\beta$) algorithm is* $4(1 + \lceil \frac{\beta}{2} \rceil)$ *access-competitive.*

To efficiently achieve an MRU($\beta$) tree, we propose the RANDOM-PUSH strategy (see Algorithm 2). This is a simple randomized strategy which selects a random path starting at the root, and then steps down the tree to depth $k = u.\text{dep}$ (the accessed item depth), by choosing uniformly at random between the two children of each server at each step. This can be seen as a simple $k$-step random walk in a directed version of the tree, starting from the root of the tree. Clearly, the adjustment cost of RANDOM-PUSH is also proportional to $k$ and its actions are independent of any oblivious online adversary. The main technical challenge of this section is proving the following theorem.

**Theorem 7.** RANDOM-PUSH *maintains an MRU*(4) *(Definition 6) tree in expectation, i.e., the expected depth of the item with rank $r$ is less than* $\log r + 3 < \lfloor \log r \rfloor + 4$ *for any sequence $\sigma$ and any time $t$.*

It now follows almost directly from Theorems 6 and 7 that RANDOM-PUSH is dynamically optimal.

**Theorem 3.** *The* RANDOM-PUSH *algorithm is dynamically optimal on expectation.*

**Algorithm 2:** Upon access to $u$ in PUSH-DOWN TREE

1: **access** $s = u$.host along tree branches (cost: $u$.dep)
2: let $v = s_1$.guest be the item at the current root
3: **move** $u$ to the root server $s_1$, setting $s_1$.guest $= u$ (cost: $u$.dep)
4: employ RANDOM-PUSH to **shift** down $v$ to depth $s$.dep (cost: $u$.dep)
5: let $w$ be the item at the end of the push-down path, where $w$.dep $= s$.dep
6: **move** $w$ to $s$, i.e., setting $s$.guest $= w$ (cost: $u$.dep $\times$ 2)

*Proof.* Let the $t$-th requested item have rank $r_t$, then the access cost is $D(r_t)$. According to the RANDOM-PUSH (Algorithm 2), the total cost is $5D(r_t)$ which is five times the access cost on the MRU(4) tree. Formally, using Theorem 6 and Theorem 7, the expected total cost is:

$$\begin{aligned}\mathbb{E}\left[\text{cost}(\text{RANDOM-PUSH})\right] = \mathbb{E}\left[\sum_{i=1}^{t} 5D(r_i)\right] &= 5\sum_{i=1}^{t}\mathbb{E}\left[D(r_i)\right] \leq 5\sum_{i=1}^{t}(\log(r_i)+3)\\ &\leq 5\sum_{i=1}^{t}(\lfloor\log(r_i)\rfloor+4) \leq 5\sum_{i=1}^{t}\text{cost}^{(t)}(\text{MRU}(4))\\ &\leq 5\cdot\text{cost}(\text{MRU}(4)) = 60\cdot\text{cost}(\text{OPT})\end{aligned}$$

□

## 6 Related Work

The work most closely related to ours arises in the context of self-adjusting datastructures. However, while datastructures need to be *searchable*, networks come with *routing* protocols: the presence of a *map* allows us to trivially access a node (or item) at distance $k$ from the front at a cost $k$. Interestingly, while we have shown in this paper that dynamically optimal algorithms for tree networks exist, the quest for constant competitive online algorithms for binary search trees remains a major open problem [10]. Nevertheless, there are self-adjusting binary search trees that are known to be *access optimal* [13], but their rearrangement cost it too high.

In the following, we first review most related work on datastructures and then discuss literature in the context of networks. A more detailed discussion appears in the technical report[12]. In contrast to CTs, self-adjustments in Binary Search Trees (BSTs) are based on *rotations* (which are assumed to have unit cost). While BSTs have the working set property, we are missing a matching lower bound: the *Dynamic Optimality Conjecture*, the question whether splay trees [10] are dynamically optimal, continues to puzzle researchers. We are also not the first to consider *Unordered* Trees (UTs) and it is known that existing lower bounds for (offline) algorithms on BSTs also apply to UTs that use rotations [15]. However, it is also known that this correspondance between ordered and unordered trees

no longer holds under weaker measures such as *key independent processing costs* and in particular *Iacono's measure* [16]: the expected cost of the sequence which results from a random assignment of keys from the search tree to the items specified in an access request sequence. Iacono's work is also one example of prior work which shows that for specific scenarios, working set and dynamic optimality properties are equivalent. Regarding the current work, we note that the reconfiguration operations in UTs are more powerful than the swapping operations considered in our paper: a rotation allows to move entire subtrees at unit costs, while the corresponding cost in CTs is linear in the subtree size. We also note that in our model, we cannot move freely between levels, but moves can only occur between parent and child. In contrast to UTs, CTs are bound to be balanced.

Intriguingly, although Skip Lists (SLs) and BSTs can be transformed to each other, Bose et al. [17] were able to prove dynamic optimality for (a restricted kind of) SLs as well as B-Trees (BTs). However, the quest for proving dynamic optimality for general skip lists remains an open problem: two restricted types of models were considered in [17], bounded and weakly bounded. Due to the relationship between SLs and BSTs, a dynamically optimal SL would imply a working set lower bound for BST. Moreover, while both in their model and ours, proving the working set property is key, the problems turn out to be fundamentally different. In contrast to SLs, CTs revolve around *unordered* (and balanced) trees (that do not provide a simple search mechanism), rely on a different reconfiguration operation (i.e., swapping or *pushing* an item to its parent comes at unit cost), and, as we show in this paper, actually provide dynamic optimality for their general form. Finally, we note that [17] (somewhat implicitly) also showed that a random walk approach can achieve the working set property; in our paper, we show that the working set property can even be achieved deterministically and without maintaining MRU.

Finally, little is known about self-adjusting *networks*. While there exist several algorithms for the design of *static* demand-aware networks, e.g. [7,8,18,19], online algorithms which also minimize reconfiguration costs are less explored. The most closely related work to ours are *SplayNets* [20,21], which are also based on a tree topology (but a searchable one). However, *SplayNets* do not provide any formal guarantees over time, besides convergence properties in case of certain fixed demands.

**Acknowledgments.** Research supported by the ERC Consolidator grant *AdjustNet* (agreement no. 864228).

## References

1. K.-T. Foerster and S. Schmid, "Survey of reconfigurable data center networks: Enablers, algorithms, complexity," in *SIGACT News*, 2019.
2. M. Ghobadi et al., "Projector: Agile reconfigurable data center interconnect," in *Proc. ACM SIGCOMM*, pp. 216–229, 2016.
3. N. Hamedazimi, Z. Qazi, H. Gupta, V. Sekar, S. R. Das, J. P. Longtin, H. Shah, and A. Tanwer, "Firefly: A reconfigurable wireless data center fabric using free-

space optics," in *Proc. ACM SIGCOMM Computer Communication Review (CCR)*, vol. 44, pp. 319–330, 2014.
4. S. Bojja Venkatakrishnan, M. Alizadeh, and P. Viswanath, "Costly circuits, submodular schedules and approximate carathéodory theorems," in *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, pp. 75–88, 2016.
5. C. Avin, M. Ghobadi, C. Griner, and S. Schmid, "On the complexity of traffic traces and implications," in *Proc. ACM SIGMETRICS*, 2020.
6. S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: measurements & analysis," in *Proc. 9th ACM Internet Measurement Conference (IMC)*, pp. 202–208, 2009.
7. C. Avin, K. Mondal, and S. Schmid, "Demand-aware network designs of bounded degree," *Distributed Computing*, 2017.
8. C. Avin, K. Mondal, and S. Schmid, "Demand-aware network design with minimal congestion and route lengths," in *Proc. IEEE INFOCOM*, pp. 1351–1359, 2019.
9. C. Avin and S. Schmid, "Toward demand-aware networking: A theory for self-adjusting networks," *ACM SIGCOMM Computer Communication Review*, vol. 48, no. 5, pp. 31–40, 2019.
10. D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *J. ACM*, vol. 32, pp. 652–686, July 1985.
11. D. D. Sleator and R. E. Tarjan, "Amortized efficiency of list update and paging rules," *Commun. ACM*, vol. 28, pp. 202–208, Feb. 1985.
12. C. Avin, K. Mondal, and S. Schmid, "Push-down trees: Optimal self-adjusting complete trees," in *Technical Report arXiv 1807.04613*, 2020.
13. A. Blum, S. Chawla, and A. Kalai, "Static optimality and dynamic search-optimality in lists and trees," in *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2002.
14. B. C. Dean and Z. H. Jones, "Exploring the duality between skip lists and binary search trees," in *Proceedings of the 45th Annual Southeast Regional Conference*, ACM-SE 45, (New York, NY, USA), pp. 395–399, ACM, 2007.
15. M. L. Fredman, "Generalizing a theorem of wilber on rotations in binary search trees to encompass unordered binary trees," *Algorithmica*, vol. 62, no. 3-4, pp. 863–878, 2012.
16. J. Iacono, "Key-independent optimality," *Algorithmica*, vol. 42, no. 1, pp. 3–10, 2005.
17. P. Bose, K. Douïeb, and S. Langerman, "Dynamic optimality for skip lists and b-trees," in *Proc. 19th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 1106–1114, 2008.
18. K.-T. Foerster, M. Ghobadi, and S. Schmid, "Characterizing the algorithmic complexity of reconfigurable data center architectures," in *Proc. ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2018.
19. A. Singla, A. Singh, K. Ramachandran, L. Xu, and Y. Zhang, "Proteus: a topology malleable data center network," in *Proc. ACM Workshop on Hot Topics in Networks (HotNets)*, 2010.
20. B. Peres, A. d. O. Otavio, O. Goussevskaia, C. Avin, and S. Schmid, "Distributed self-adjusting tree networks," in *Proc. IEEE INFOCOM*, pp. 145–153, 2019.
21. S. Schmid, C. Avin, C. Scheideler, M. Borokhovich, B. Haeupler, and Z. Lotker, "Splaynet: Towards locally self-adjusting networks," *IEEE/ACM Trans. Netw.*, vol. 24, pp. 1421–1433, June 2016.