DYNAMIC BALANCED GRAPH PARTITIONING*

MARCIN BIENKOWSKI†, CHEN AVIN‡, ANDREAS LOUKAS§, MACIEJ PACUT†, AND STEFAN SCHMID¶

Abstract. This paper initiates the study of the classic balanced graph partitioning problem from an online perspective: Given an arbitrary sequence of pairwise communication requests between n nodes, with patterns that may change over time, the objective is to service these requests efficiently by partitioning the nodes into ℓ clusters, each of size k, such that frequently communicating nodes are located in the same cluster. The partitioning can be updated dynamically by *migrating* nodes between clusters. The goal is to devise online algorithms which jointly minimize the amount of inter-cluster communication and migration cost.

The problem features interesting connections to other well-known online problems. For example, scenarios with $\ell = 2$ generalize online paging, and scenarios with $k = 2$ constitute a novel online variant of maximum matching. We present several lower bounds and algorithms for settings both with and without cluster-size augmentation. In particular, we prove that any deterministic online algorithm has a competitive ratio of at least k, even with *significant* augmentation. Our main algorithmic contributions are an $O(k \log k)$-competitive deterministic algorithm for the general setting with constant augmentation, and a constant competitive algorithm for maximum matching.

Key words. clustering, graph partitioning, competitive analysis, cloud computing

1. Introduction.

Graph partitioning problems, like minimum graph bisection or minimum balanced cuts, are among the most fundamental problems in theoretical computer science. They are intensively studied also due to their numerous practical applications, e.g., in communication networks, parallel processing, data mining and community discovery in social networks. Interestingly however, not much is known today about how to *dynamically* partition nodes which interact or communicate in a time-varying fashion.

This paper initiates the study of a natural model for *online graph partitioning*. We are given a set of n nodes with time-varying pairwise communication patterns, which have to be partitioned into ℓ clusters of equal size k. Intuitively, we would like to minimize inter-cluster interactions by mapping frequently communicating nodes to the same cluster. Since communication patterns change over time, partitions should be dynamically readjusted, that is, the nodes should be *repartitioned*, in an online manner, by *migrating* them between clusters. The objective is to jointly minimize inter-cluster communication and repartitioning costs, defined respectively as the number of communication requests "served remotely" and the number of times nodes are migrated from one cluster to another.

The online graph problem is a fundamental one and has many applications. For example, in the context of cloud computing, n may represent virtual machines or containers that are distributed across ℓ physical servers, each having k cores: each server can host k virtual machines. We would like to (dynamically) distribute the virtual machines across the servers such that datacenter traffic and migration costs are minimized.

1.1. The Model.

Formally, the online *Balanced RePartitioning* problem (BRP) is defined as follows. There is a set of n nodes, initially distributed arbitrarily across ℓ clusters, each of size k. We call two nodes $u, v \in V$ *collocated* if they are in the same cluster.

An input to the problem is a sequence of communication requests $\sigma = (u_1, v_1), (u_2, v_2), (u_3, v_3), \ldots$, where pair (u_t, v_t) means that nodes exchange a fixed amount of data. For succinctness of later descriptions, we assume that a request (u_t, v_t) occurs at time $t \geq 1$. At any time $t \geq 1$, an online algorithm needs to serve the communication request (u_t, v_t). Right before serving the request, the online algorithm can repartition the nodes into new clusters (i.e., we assume the usual *one-lookahead model*[1]) We assume that a communication request between two collocated nodes costs 0. The cost of a communication request between two nodes located in different clusters

---

* A preliminary version of this paper appeared as "Online Balanced Repartitioning" in the proceedings of the 30th International Symposium on DIStributed Computing (DISC 2016).

Funding: Research supported by the German-Israeli Foundation for Scientific Research (GIF) Grant I-1245-407.6/2014 and the Polish National Science Centre grants 2016/22/E/ST6/00499 and 2016/23/N/ST6/03412.

†University of Wroclaw, Poland
‡Ben Gurion University of the Negev, Israel
§TU Berlin, Germany
¶Aalborg University, Denmark

[1] It is easy to see that the cost in a zero-lookahead model is at most a constant factor larger.

is normalized to 1, and the cost of migrating a node from one cluster to another is $\alpha \geq 1$, where α is a parameter (an integer). For any algorithm $\textsc{Alg}$, we denote its total cost (consisting of communication plus migration costs) on sequence σ by $\textsc{Alg}(\sigma)$.

We are in the realm of competitive worst-case analysis and compare the performance of an online algorithm to the performance of an optimal offline algorithm. Formally, let $\textsc{Onl}(\sigma)$ resp. $\textsc{Opt}(\sigma)$ be the cost induced by σ on an online algorithm $\textsc{Onl}$ resp. on an optimal offline algorithm $\textsc{Opt}$. In contrast to $\textsc{Onl}$, which learns the requests one-by-one as it serves them, $\textsc{Opt}$ has a complete knowledge of the entire request sequence σ ahead of time. The goal is to design online repartitioning algorithms that provide worst-case guarantees. In particular, $\textsc{Onl}$ is said to be ρ-competitive if there is a constant β such that for any input sequence σ it holds that

$$\textsc{Onl}(\sigma) \leq \rho \cdot \textsc{Opt}(\sigma) + \beta \ .$$

Note that β cannot depend on input σ but can depend on other parameters of the problem, such as the number of nodes or the number of clusters. The minimum ρ for which $\textsc{Onl}$ is ρ-competitive is called the *competitive ratio* of $\textsc{Onl}$.

We consider two different settings:

Without augmentation: The nodes fit perfectly into the clusters, i.e., $n = k \cdot \ell$. Note that in this setting, due to cluster capacity constraints, a node can never be migrated alone, but it must be *swapped* with another node at a cost of 2α. We assume that also when an algorithm wants to migrate more than two nodes, this has to be done using several swaps, each involving two nodes.

With augmentation: An online algorithm has access to additional space in each cluster. We say that an algorithm is δ-augmented if the size of each cluster is $k' = \delta \cdot k$, whereas the total number of nodes remains $n = k \cdot \ell < k' \cdot \ell$. As usual, in competitive analysis, the augmented online algorithm is compared to the optimal offline algorithm with cluster capacity k.

An online repartitioning algorithm has to cope with the folowing issues:

Serve remotely or migrate ("rent or buy")? If a communication pattern is short-lived, it may not be worthwhile to collocate the nodes: the migration might be too large in comparison to communication costs.

Where to migrate, and what? If an algorithm decides to collocate nodes x and y, the question becomes how. Should x be migrated to the cluster holding y, y to the one holding x, or should both nodes be migrated to a new cluster?

Which nodes to evict? There may not exist sufficient space in the desired destination cluster. In this case, the algorithm needs to decide which nodes to "evict" (migrate to other clusters), to free up space.

1.2. Our Contributions. This paper introduces the online Balanced RePartitioning problem (BRP), a fundamental *dynamic* variant of the classic graph clustering problem. We show that BRP features some interesting connections to other well-known online online graph problems. For $\ell = 2$, BRP is able to simulate online paging problem and for for $k = 2$, BRP is a novel online version of maximum matching. We consider deterministic algorithms and make the following technical contributions:

Algorithms for General Variant: For the non-augmented variant, in Section 3, we first present a simple $O(k^2 \cdot \ell^2)$-competitive algorithm. Our main technical contribution is an $O((1 + 1/\epsilon) \cdot k \log k)$-competitive deterministic algorithm $\textsc{Crep}$ for a setting with $(2 + \epsilon)$-augmentation (Section 4). We emphasize that this bound does not depend on ℓ. This is interesting, as in many application domains of this problem, k is small: for example, in our motivating virtual machine collocation problem, a server typically hosts only a small number of virtual machines (e.g., related to the constant number of cores on the server).

Algorithms for Online Rematching: For the special case of online rematching ($k = 2$, but arbitrary ℓ), in Section 5, we prove that a variant of a greedy algorithm is 7-competitive. We also demonstrate a lower bound of 3 for any deterministic algorithm.

Lower Bounds: By a reduction to online paging, in Section 6.1, we show that for two clusters no deterministic algorithm can obtain a better bound than $k - 1$. While this shows an interesting link between BRP and paging, in Section 6.2, we present a stronger bound. Namely, we show that for $\ell \geq 2$ clusters, no deterministic algorithm can beat the bound of k even with an arbitrary amount of augmentation, as long as the algorithm cannot keep all nodes in a single cluster. In contrast, online paging is known to become constant competitive with constant augmentation [33].

1.3. A Practical Motivation. The dynamic graph clustering problem is a fundamental one and comes with many applications. To give just one example, we consider server virtualization in datacenters. Distributed cloud applications, including batch processing applications such as MapReduce, streaming applications such as Apache Flink or Apache Spark, and scale-out databases and key-value stores such as Cassandra, generate a significant amount of network traffic and a considerable fraction of their runtime is due to network activity [29]. For example, traces of jobs from a Facebook cluster reveal that network transfers on average account for 33% of the execution time [11]. In such applications, it is desirable that frequently communicating virtual machines are *collocated*, i.e., mapped to the same physical server, since communication across the network (i.e., inter-server communication) induces network load and latency. However, migrating virtual machines between servers also comes at a price: the state transfer is bandwidth intensive, and may even lead to short service interruptions. Therefore the goal is to design online algorithms that find a good trade-off between the inter-server communication cost and the migration cost.

2. Related Work. The static offline version of our problem, i.e., a problem variant where migration is not allowed, where all requests are known in advance, and where the goal is to find best node assignment to ℓ clusters, is known as the ℓ-balanced graph partitioning problem. The problem is known to be NP-complete, and cannot even be approximated within any finite factor unless P = NP [2]. The static variant where $n/\ell = 2$ corresponds to a maximum matching problem, which is polynomial-time solvable. The static variant where $\ell = 2$ corresponds to the minimum bisection problem, which is already NP-hard [21]. Its approximation was studied in a long line of work [31, 3, 17, 16, 24, 30] and the currently best approximation ratio of $O(\log n)$ was given by Räcke [30]. The $O(\log^{3/2} n)$-approximation given by Krauthgamer and Feige [24] can be extended to general ℓ, but the running time becomes exponential in ℓ.

The inaproximability of the static variant for general values of ℓ motivated research on the bicriteria variant, which can be seen as the offline counterpart of our cluster-size augmentation approach. Here, the goal is to develop (ℓ, δ)-balanced graph partitioning, where the graph has to be partitioned in ℓ components of size less than $\delta \cdot (n/\ell)$ and the cost of the cut is compared to the optimal (non-augmented) solution where all components are of size n/ℓ. The variant where $\delta \geq 2$ was considered in [26, 32, 15, 14, 25]. So far the best result is an $O(\sqrt{\log n \cdot \log \ell})$-approximation by Krauthgamer et al. [25], which builds on ideas from the $O(\sqrt{\log n})$-aproximation algorithm for balanced cuts by Arora et al. [4]. For smaller values of δ, i.e., when $\delta = 1 + \epsilon$ with a fixed $\epsilon > 0$, Andreev and Räcke gave an $O(\log^{1.5} n/\epsilon^2)$ approximation [2], which was later improved to $O(\log n)$ by Feldmann and Foschini [18].

The BRP problem considered in this paper was not previously studied. However, it bears some resemblance to the classic online problems; below we highlight some of them.

Our model is related to online paging [33, 20, 28, 1], sometimes also referred to as online caching, where requests for data items (nodes) arrive over time and need to be served from a cache of finite capacity, and where the number of cache misses must be minimized. Classic problem variants usually boil down to finding a smart eviction strategy, such as Least Recently Used (LRU). In our setting, requests can be served remotely (i.e., without fetching the corresponding nodes to a single cluster). In this light, our model is more reminiscent of caching models *with bypassing* [12, 13, 22]. Nonetheless, we show that BRP is capable of emulating online paging.

The BRP problem is an example of a non-uniform problem [23]: the cost of changing the state is higher than the cost of serving a single request. This requires finding a good trade-off between serving requests remotely (at a low but repeated communication cost) or migrating nodes into a single cluster (entailing a potentially high one-time cost $O(\alpha)$). Many online problems exhibit this so called *rent-or-buy* property, e.g., ski rental problem [23, 27], relaxed metrical task systems [8], file migration [8, 10], distributed data management [9, 6, 7], or rent-or-buy network design [5, 34, 19].

There are two major differences between BRP and the problems listed above. First, these problems typically maintain some configuration of servers or bought infrastructure and upon a new request (whose cost typically depends on the distance to the infrastructure), decide about its reconfiguration (e.g., server movement or purchasing additional links). In constrast, in our model, *both* end-points of a communication request are subject to optimization. Second, in the BRP problem a request reveals only very limited information about the optimal configuration to serve it: There exist relatively long sequences of requests that can be served with zero cost from a fixed configuration. Not only can the set of such configurations be very large, but such configurations may also differ significantly from each other.

3. A Simple Upper Bound. As a warm-up and to present the model, we start with a straightforward $O(k^2 \cdot \ell^2)$-competitive deterministic algorithm Det. At any time, Det serves a request, adjusts its internal structures (defined below) accordingly and then possibly migrates nodes. Det operates in phases, and each phase is analyzed separately. The first phase starts with the first request.

In a single phase, Det maintains a helper structure: a complete graph on all $\ell \cdot k$ nodes, with an edge present between each pair of nodes. We say that a communication request is *paid* (by Det) if it occurs between nodes from different clusters, and thus entails a cost for Det. For each edge between nodes x and y, we define its weight $w_{x,y}$ to be the number of paid communication requests between x and y since the beginning of the current phase.

Whenever an edge weight reaches α, it is called *saturated*. If a request causes the corresponding edge to become saturated, Det computes a new placement of nodes (potentially for all of them), so that all saturated edges are inside clusters (there is only one new saturated edge). If this is not possible, node positions are not changed, the current phase ends with the current request and the new phase begins with the next request. Note that all edge weights are reset to zero at the beginning of a phase.

Theorem 3.1. Det *is* $O(k^2 \cdot \ell^2)$*-competitive.*

Proof. We bound the costs of Det and Opt in a single phase. First, observe that whenever an edge weight reaches α, its endpoint nodes will be collocated until the end of the phase, and therefore its weight is not incremented anymore. Hence the weight of any edge is at most α.

Second, observe that the graph induced by saturated edges always constitutes a forest. For the sake of contradiction, suppose that, at a time t, two nodes u and v, which are not connected by a saturated edge, become connected by a path of saturated edges. From that time onward, they are stored by Det in a single cluster. Hence, the weight $w_{u,v}$ cannot increase at subsequent time points, and (u, v) may not become saturated. The forest property implies that the number of saturated edges is smaller than $k \cdot \ell$.

The two observations above allow us to bound the cost of Det in a single phase. The number of reorganizations is at most the number of saturated edges, i.e., at most $k \cdot \ell$. As the cost associated with a single reorganization is $O(k \cdot \ell \cdot \alpha)$, the total cost of all node migrations in a single phase is at most $O(k^2 \cdot \ell^2 \cdot \alpha)$. The communication cost itself is equal to the total weight of all edges, and by the first observation, it is at most $\binom{k \cdot \ell}{2} \cdot \alpha < k^2 \cdot \ell^2 \cdot \alpha$. Hence for any phase P (also for the last one), it holds that $\text{Det}(P) = O(k^2 \cdot \ell^2 \cdot \alpha)$.

Now we lower-bound the cost of Opt on any phase P but the last one. If Opt performs a node swap in P, it pays 2α. Otherwise its assignment of nodes to clusters is fixed throughout P. Recall that at the end of P, Det failed to reorganize the nodes. This means that for any static mapping of the nodes to clusters (in particular the one chosen by Opt), there will be a saturated intra-cluster edge. The communication cost over such an edge incurred by Opt is at least α (it can be also strictly greater than α as the edge weight only counts the communication requests paid by Det).

Therefore, the Det-to-Opt cost ratio in any phase but the last one is at most $O(k^2 \cdot \ell^2)$ and the cost of Det on the last phase is at most $O(k^2 \cdot \ell^2 \cdot \alpha)$. Therefore, $\text{Det}(\sigma) \le O(k^2 \cdot \ell^2) \cdot \text{Opt}(\sigma) + O(k^2 \cdot \ell^2 \cdot \alpha)$ for any input σ. ∎

4. Algorithm Crep. In this section, we present the main result of this paper, a *Component-based REPartitioning algorithm (*Crep*)* which achieves a competitive ratio of $O((1 + 1/\epsilon) \cdot k \log k)$ with augmentation $2 + \epsilon$, for any $\epsilon > 0$. Crep maintains a similar graph structure as the simple deterministic $O(k^2 \cdot \ell^2)$-competitive algorithm from the previous section, i.e., it keeps counters denoting how many times it paid for a communication between two nodes. Similarly, at any time t, Crep serves the current request, adjusts its internal structures, and then possibly migrates nodes. The execution of Crep is *not* partitioned into global phases: the reset of counters to zero can occur at different times.

4.1. Algorithm Definition. We describe the construction of Crep in two stages. The first stage uses an intermediate concept of *communication components*, which are groups of at most k nodes. In the second stage, we show how components are assigned to clusters, so that all nodes from any single component are always stored in a single cluster.

4.1.1. Stage 1: Maintaining Components. Roughly speaking, nodes are grouped into components if they communicated a lot recently. At the very beginning, each node is in a singleton component. Once the cumulative communication cost between nodes distributed across s components exceeds $\alpha \cdot (s-1)$, Crep merges them into a single component. If a resulting component size exceeds k, it becomes deleted and replaced by singleton components.

More precisely, the algorithm maintains a time-varying *partition of all nodes into components*. As a helper structure, Crep keeps a complete graph on all $k \cdot \ell$ nodes, with an edge present between each pair of nodes. For each edge between nodes x and y, Crep maintains its weight $w_{x,y}$. We say that a communication request is *paid* (by Crep) if it occurs between nodes from different clusters, and thus entails a cost for Crep. If x and y belong to the same component, then $w_{x,y} = 0$. Otherwise, $w_{x,y}$ is equal to the number of paid communication requests between x and y since the last time when they were placed in different components by Crep. It is worth emphasizing that during an execution of Crep, it is possible that $w_{x,y} > 0$ even when x and y belong to the same cluster.

For any subset of components $S = \{c_1, c_2, \ldots, c_{|S|}\}$ (called *component-set*), by $w(S)$ we denote the total weight of all edges between nodes of S. Note that positive weight edges occur only between different components of S. We call a component-set *trivial* if it contains only one component; $w(S) = 0$ in such a case.

Initially, all components are singleton components and all edge weights are zero. At time t, upon a communication request between a pair of nodes x and y, if x and y lie in the same cluster, the corresponding cost is 0 and Crep does nothing. Otherwise, the cost entailed to Crep is 1, nodes x and y lie in different clusters (and hence also in different components), and the following updates of weights and components are performed.

1. *Weight increment.* Weight $w_{x,y}$ is incremented.
2. *Merge actions.* We say that a non-trivial component-set $S = \{c_{i_1}, c_{i_2}, \ldots, c_{i_{|S|}}\}$ is *mergeable* if $w(S) \geq (|S|-1) \cdot \alpha$. If a mergeable component-set S exists, then all its components are merged into a single one. If multiple mergeable component-sets exist, Crep picks the one with maximum number of components, breaking ties arbitrarily. Weights of all intra-S edges are reset to zero, and thus intra-component edge weights are always zero. A mergeable set S induces a sequence of $|S| - 1$ *merge actions*: Crep iteratively replaces two arbitrary components from S by a component being their union (this constitutes a single merge action).
3. *Delete action.* If the component resulting from merge action(s) has more than k nodes, it is deleted and replaced by singleton components. Note that weights of edges between these singleton components are all zero as they have been reset by the preceding merge actions.

We say that merge actions are *real* if they are not followed by a delete action (at the same time point) and *artificial* otherwise.

4.1.2. Stage 2: Assigning Components to Clusters. At time t, Crep processes a communication request and recomputes components as described in the first stage. Recall that we require that nodes of a single component are always stored in a single cluster. To maintain this property for artificial merge actions, no actual migration is necessary. The property may however be violated by real merge actions. Hence, in the following, we assume that in the first stage Crep found a mergeable component set $S = \{c_1, c_2, \ldots, c_{|S|}\}$ that triggers $|S| - 1$ merge actions not followed by a delete action.

Crep consecutively processes each real merge action by migrating some nodes. We describe this process for a single real merge action involving two components c_x and c_y. As a delete action was not executed, $|c_x| + |c_y| \leq k$, where $|c|$ denotes the number of component c nodes. Without loss of generality, $|c_x| \leq |c_y|$.

We may assume that c_x and c_y are in different clusters as otherwise Crep does nothing. If the cluster containing c_y has $|c_x|$ free space, then c_x is migrated to this cluster. Otherwise, Crep finds a cluster that has at most k nodes, and moves both c_x and c_y there. We call the corresponding actions *component migrations*. By an averaging argument, there always exists a cluster that has at most k nodes, and hence, with $(2+\epsilon)$-augmentation, component migrations are always feasible.

4.2. Analysis: Structural Properties. We start with a structural property of components and edge weights. It states that immediately after Crep merges (and possibly deletes) a component-set, no other component-set is mergeable. This property holds independently of the actual placement of components in particular clusters.

Lemma 4.1. *At any time t, after* Crep *performs its merge and delete actions (if any), $w(S) < \alpha \cdot (|S|-1)$ for any non-trivial component-set S.*

Proof. We prove the lemma by induction on steps. Clearly, the lemma holds before an input sequence starts as then $w(S) = 0 \leq \alpha - 1 < \alpha \cdot (|S|-1)$ for any non-trivial set S. We assume that it holds at time $t-1$ and show it for time t.

At time t, only a single weight, say $w_{x,y}$, may be incremented. If after the increment, Crep does not merge any component, then clearly $w(S) < \alpha \cdot (|S|-1)$ for any non-trivial set S. Otherwise, at time t, Crep merges a component-set A into a new component c_A, and then possibly deletes c_A and creates singleton components from its nodes. We show that the lemma statement holds then for any non-trivial component-set S. We consider three cases.

1. Component-sets A and S do not share any common node. Then, A and S consist only of components that were present already right before time t and they are all disjoint. The edge (x, y) involved in communication at time t is contained in A, and hence does not contribute to the weight of $w(S)$. By the inductive assumption, $w(S) < \alpha \cdot (|S|-1)$ holds right before time t. As $w(S)$ is not affected by Crep actions at step t, the inequality holds also right after time t.
2. Crep does not delete c_A and $c_A \in S$. Let $X = S \setminus \{c_A\}$. Let $w(A, X)$ denote the total weight of all edges with one endpoint in A and another in X. As Crep merged component-set A and did not merge component-set $A \uplus X$, A was mergeable ($w(A) \geq \alpha \cdot (|A|-1)$), while $A \uplus X$ was not, i.e., $w(A) + w(A, X) + w(X) = w(A \uplus X) < \alpha \cdot (|A| + |X| - 1)$. Therefore, $w(A, X) + w(X) < \alpha \cdot |X|$ right after weight $w_{x,y}$ is incremented at time t. Observe that neither $w(A, X)$ nor $w(X)$ is affected by the merge of component-set A and (Stefan: marcin: unclear:) resetting weights of all intra-A edges to zero. Therefore after Crep merges A into c_A, it holds that $w(S) = w(A, X) + w(X) < \alpha \cdot |X| = \alpha \cdot (|S|-1)$.
3. Crep deletes c_A creating singleton components $d_1, d_2, \ldots, d_r$ and some of these components belong to set S. This time, we define X to be the set S without these components (X might be also an empty set). In the same way as in the previous case, we may show that $w(A, X) + w(X) < \alpha \cdot |X|$ after Crep performs merge and delete operations. Hence, at this time $w(S) \leq w(A, X) + w(X) < \alpha \cdot |X| \leq \alpha \cdot (|S|-1)$. The last inequality follows as S has strictly more components than X.

Since only one request is given at a time, and since all weights and α are integers, Lemma 4.1 immediately implies the following result.

Corollary 4.2. *Fix any time t and consider weights right after they are updated by* Crep, *but before* Crep *performs merge actions. Then, $w(S) \leq (|S|-1) \cdot \alpha$ for any component-set S. In particular, $w(S) = (|S|-1) \cdot \alpha$ for a mergeable component-set S.*

4.3. Analysis: Lower Bound on OPT. For estimating the cost of Opt, we pick any input sequence σ and we execute Crep on it. Then we execute Opt on σ and we analyze its cost in terms of the number of merges and deletions performed by Crep. We split any swap of two nodes performed by Opt into two migrations of the corresponding nodes.

For any component c maintained by Crep, let $\tau(c)$ be the time of its creation. A non-singleton component c is created at $\tau(c)$ by the merge of a component-set, henceforth denoted by $S(c)$. For a singleton component, $\tau(c)$ is the time when the component that previously contained the sole node of c was deleted; $\tau(c) = 0$ if c existed at the beginning of input σ. We will use time 0 as an artificial time point that occurred before an actual input sequence.

For a non-singleton component c, we define $F(c)$ as the set of the following (node, time) pairs:

$$F(c) = \biguplus_{b \in S(c)} \{b\} \times \{\tau(b)+1, \ldots, \tau(c)\} \quad .$$

Intuitively, $F(c)$ tracks the history of all nodes of c from the time (exclusively) they started belonging to some previous component b, until the time (inclusively) they become members of c. Note that sets F are disjoint and they cover all possible node-time pairs (except for time zero).

For a given component c, we say that a communication request between nodes x and y at time t *is contained* in $F(c)$ if both $(x,t) \in F(c)$ and $(y,t) \in F(c)$. Note that only the requests contained in $F(c)$ could contribute towards later creation of c by Crep. In fact, by Corollary 4.2, the number of these requests that entailed an actual cost to Crep is exactly $(|S(c)| - 1) \cdot \alpha$.

We say that a migration of node x performed by Opt at time t *is contained* in $F(c)$ if $(x,t) \in F(c)$. For any component c, we define Opt(c) as the cost incurred by Opt due to requests contained in $F(c)$, plus the cost of Opt migrations contained in $F(c)$. The total cost of Opt can then be lower-bounded by the sum of Opt(c) over all components c. (The cost of Opt can be larger as $\sum_c$ Opt(c) does not account for communication requests not contained in $F(c)$ for any component c.)

Lemma 4.3. *Fix any component c and partition $S(c)$ into a set of $g \geq 2$ disjoint component-sets $S_1, S_2, \ldots, S_g$. The number of communication requests in $F(c)$ that are between sets S_i is at least $(g-1) \cdot \alpha$.*

Proof. Let w be the weight measured right after its increment at time $\tau(c)$. Observe that the number of all communication requests from $F(c)$ that were between sets S_i and that *were paid* by Crep is $w(S(c)) - \sum_{i=1}^{g} w(S_i)$. It suffices to show that this amount is at least $(g-1) \cdot \alpha$. By Corollary 4.2, $w(S(c)) = (|S(c)| - 1) \cdot \alpha$ and $w(S_i) \leq (|S_i| - 1) \cdot \alpha$. Therefore, $w(S(c)) - \sum_{i=1}^{g} w(S_i) \geq (|S(c)| - 1) \cdot \alpha - \sum_{i=1}^{g} (|S_i| - 1) \cdot \alpha = (g-1) \cdot \alpha$. □

For any component c maintained by Crep, let Y_c denote set of clusters containing nodes of c in the solution of Opt after Opt performs its migrations (if any) at time $\tau(c)$. If particular, if $\tau(c) = 0$, then Y_c consists of only one cluster that contained the sole node of c at the beginning of an input sequence.

Lemma 4.4. *For any non-trivial component c, it holds that* Opt$(c) \geq (|Y_c| - 1) \cdot \alpha - \sum_{b \in S(c)} (|Y_b| - 1) \cdot \alpha$.

Proof. Fix a component $b \in S(c)$ and any node $x \in b$. Let opt-mig(x) be the number of Opt migrations of node x at times $t \in \{\tau(b)+1, \ldots, \tau(c)\}$ (recall that Opt may perform migrations both before and after serving the request at time t). Furthermore, let Y'_x be the set of clusters that contained x at some moment of a time $t \in \{\tau(b)+1, \ldots, \tau(c)\}$. We extend these notions to components: opt-mig$(b) = \sum_{x \in b}$ opt-mig(x) and $Y'_b = \bigcup_{x \in b} Y'_x$. Observe that $|Y'_b| \leq |Y_b| +$ opt-mig(b).

We aggregate components of $S(c)$ into component-sets called *bundles*, so that any two bundles have their nodes always in disjoint clusters. To this end, we construct a hypergraph, whose nodes correspond to clusters from $\bigcup_{b \in S(c)} Y'_b$. Each component $b \in S(c)$ defines a hyperedge that connects all nodes (clusters) that are in Y'_b. Now we look at connected hypergraph components (called *hypergraph parts* to avoid ambiguity). There are

$$\begin{aligned} B &\geq |\textstyle\bigcup_{b \in S(c)} Y'_b| - \sum_{b \in S(c)} (|Y'_b| - 1) \\ &\geq |Y_c| - \textstyle\sum_{b \in S(c)} (|Y_b| - 1) - \sum_{b \in S(c)} \text{opt-mig}(b) \end{aligned}$$

hypergraph parts. Each hypergraph part corresponds to a bundle consisting of components contained in clusters belonging to this part, i.e., the number of bundles is also B.

By Lemma 4.3, the number of communication requests in $F(c)$ that are between different bundles is at least $(B-1) \cdot \alpha$. Each such request is paid by Opt because, by the definition of bundles, it involves a communication between two nodes which Opt stored in different clusters. Additionally, Opt(c) involves $\sum_{b \in S(c)}$ opt-mig(b) node migrations in $F(c)$, and therefore Opt$(c) \geq (B-1) \cdot \alpha + \sum_{b \in S(c)}$ opt-mig$(b) \cdot \alpha \geq (|Y_c| - 1) \cdot \alpha - \sum_{b \in S(c)} (|Y_b| - 1) \cdot \alpha$. □

Lemma 4.5. *For any input σ, let* del(σ) *be the set of components that were eventually deleted by* Crep. *Then* Opt$(\sigma) \geq \sum_{c \in \text{del}(\sigma)} |c|/(2k) \cdot \alpha$.

Proof. Fix any component $c \in \textsc{del}(\sigma)$. Consider a tree $\mathcal{T}(c)$ which describes how component c was created: the leaves of $\mathcal{T}(c)$ are singleton components containing nodes of c, the root is c itself, and each internal node corresponds to a component created at a single time by merging its children.

We now sum $\textsc{Opt}(b)$ over all components b from $\mathcal{T}(c)$, including the root c and the leaves $L(\mathcal{T}(c))$. The lower bound given by Lemma 4.4 sums telescopically, i.e.,

$$\begin{aligned} \textstyle\sum_{b\in\mathcal{T}(c)} \textsc{Opt}(b) &\geq (|Y_c|-1)\cdot\alpha - \textstyle\sum_{b\in L(\mathcal{T}(c))}(|Y_b|-1)\cdot\alpha \\ &= (|Y_c|-1)\cdot\alpha \ , \end{aligned}$$

where the equality follows as any $b \in L(\mathcal{T}(c))$ is a singleton component, and therefore $|Y_b| = 1$. As c has $|c|$ nodes, it has to span at least $\lceil |c|/k \rceil$ clusters of $\textsc{Opt}$, and therefore $\sum_{b\in\mathcal{T}(c)} \textsc{Opt}(b) \geq (\lceil |c|/k\rceil - 1)\cdot\alpha \geq |c|/(2k)\cdot\alpha$, where the second inequality follows because $c \in \textsc{del}(\sigma)$ and thus $|c| > k$.

The proof is concluded by observing that, for deleted components c, the corresponding trees $\mathcal{T}(c)$ do not share common components, and thus $\textsc{Opt}(\sigma) \geq \sum_{c\in\textsc{del}(\sigma)} \sum_{b\in\mathcal{T}(c)} \textsc{Opt}(b) \geq \sum_{c\in\textsc{del}(\sigma)} |c|/(2k)$. ∎

4.4. Analysis: Upper Bound on CREP. To upper bound the cost of $\textsc{Crep}$, we fix any input σ and introduce the following notions. Let $M(\sigma)$ be the sequence of merge actions (real and artificial ones) performed by $\textsc{Crep}$. For any real merge action $m \in M(\sigma)$, by $\textsc{size}(m)$ we denote the size of the smaller component that was merged. For an artificial merge action, we set $\textsc{size}(m) = 0$.

Recall that $\textsc{del}(\sigma)$ denotes the set of all components that become eventually deleted by $\textsc{Crep}$. Let $\textsc{final}(\sigma)$ be the set of all components that exist when $\textsc{Crep}$ finishes sequence σ. Note that $w(\textsc{final}(\sigma))$ is the total weight of all edges after processing σ.

We split $\textsc{Crep}(\sigma)$ into two parts: the cost of serving requests, $\textsc{Crep}^{\mathrm{req}}(\sigma)$, and the cost of node migrations, $\textsc{Crep}^{\mathrm{mig}}(\sigma)$.

Lemma 4.6. *For any input σ, $\textsc{Crep}^{\mathrm{req}}(\sigma) = |M(\sigma)|\cdot\alpha + w(\textsc{final}(\sigma))$.*

Proof. The proof follows by an induction over all requests of σ. Whenever $\textsc{Crep}$ pays for the communication request, the corresponding edge weight is incremented and both sides increase by 1. At a time in which s components are merged, $s-1$ merge actions are executed and the sum of all edge weights decreases exactly by $(s-1)\cdot\alpha$. Then, the value of both sides remain unchanged. ∎

Lemma 4.7. *For any input σ, with $(2+\epsilon)$-augmentation, $\textsc{Crep}^{\mathrm{mig}}(\sigma) \leq (1+4/\epsilon)\cdot\alpha\cdot\sum_{m\in M(\sigma)} \textsc{size}(m)$.*

Proof. (Stefan: good idea to call it n?:) Let V be any cluster and n be the number of nodes $\textsc{Crep}$ places in this cluster. We define the *overflow* of V as $\max\{n-2k, 0\}$. We denote the overflow of cluster V_i (for $i \in \{1,\ldots,\ell\}$) after processing sequence σ by $\textsc{ovr}^{\sigma}(V_i)$. We will show the following relation:

$$\textsc{Crep}^{\mathrm{mig}}(\sigma) \ + \ \sum_{j=1}^{\ell}(4/\epsilon)\cdot\alpha\cdot\textsc{ovr}^{\sigma}(V_j) \ \leq \ (1+4/\epsilon)\cdot\alpha\cdot\sum_{m\in M(\sigma)}\textsc{size}(m) \ . \tag{1}$$

The proof will follow by an induction on requests in σ. Clearly, (1) holds trivially at the beginning, as there are no overflows, and thus both sides of (1) are zero. Assume that (1) holds for a sequence σ and we show it for sequence $\sigma \cup \{r\}$, where r is some request.

We may focus on request r that triggers component(s) migration as otherwise (1) holds trivially. Such a migration is triggered by a real merge action m of two components c_x and c_y. We assume that $|c_x| \leq |c_y|$, and hence $\textsc{size}(m) = |c_x|$. Note that $|c_x| + |c_y| \leq k$, as otherwise the resulting component would be deleted and no migration would be performed.

Let V_x and V_y denote the cluster that held components c_x and c_y, respectively, and V_z be the destination cluster for c_x and c_y (it is possible that $V_z = V_y$). For any cluster V, we denote the change in the number of its overflow nodes by $\Delta\textsc{ovr}(V)$. It suffices to show that the change of the left hand side of (1) is at most the increase of its right hand side, i.e.,

$$\textsc{Crep}^{\mathrm{mig}}(r) \ + \sum_{V\in\{V_x,V_y,V_z\}} (4/\epsilon)\cdot\alpha\cdot\Delta\textsc{ovr}(V) \ \leq \ (1+4/\epsilon)\cdot|c_x|\cdot\alpha \ . \tag{2}$$

For the proof, we consider three cases.

1. V_y had at least $|c_x|$ empty slots. In this case, Crep simply migrates c_x to V_y paying $|c_x| \cdot \alpha$. Then, $\Delta\text{ovr}(V_x) \le 0$, $\Delta\text{ovr}(V_y) \le |c_x|$, $V_z = V_y$, and thus (2) follows.
2. V_y had less than $|c_x|$ empty slots and $|c_y| \le (2/\epsilon) \cdot |c_x|$. Crep migrates both c_x and c_y to component V_z and the incurred cost is $\text{Crep}^{\text{mig}}(r) = (|c_x| + |c_y|) \cdot \alpha \le (1 + 2/\epsilon) \cdot |c_x| \cdot \alpha < (1 + 4/\epsilon) \cdot |c_x| \cdot \alpha$. It remains to show that the second summand of (2) is at most 0. Clearly, $\Delta\text{ovr}(V_x) \le 0$ and $\Delta\text{ovr}(V_y) \le 0$. Furthermore, the number of nodes in V_z was at most k before the migration by the definition of Crep, and thus is at most $k + |c_x| + |c_y| \le 2k$ after the migration. This implies that $\Delta\text{ovr}(V_z) = 0 - 0 = 0$.
3. V_y had less than $|c_x|$ empty slots and $|c_y| > (2/\epsilon) \cdot |c_x|$. As in the previous case, Crep migrates c_x and c_y to component V_z, paying $\text{Crep}^{\text{mig}}(r) = (|c_x| + |c_y|) \cdot \alpha < 2 \cdot |c_y| \cdot \alpha$. This time, $\text{Crep}^{\text{mig}}(r)$ can be much larger than the right hand side of (2), and thus we will resort to showing that the second summand of (2) is at most $-2 \cdot |c_y| \cdot \alpha$.
   As in the previous case, $\Delta\text{ovr}(V_x) \le 0$ and $\Delta\text{ovr}(V_z) = 0$. Observe that $|c_x| < (\epsilon/2) \cdot |c_y| \le (\epsilon/2) \cdot k$. As the migration of $|c_x|$ to V_y was not possible, the initial number of nodes in V_y was greater than $(2 + \epsilon) \cdot k - |c_x| \ge (2 + \epsilon/2) \cdot k$, i.e., $\text{ovr}^{\sigma}(V_y) \ge (\epsilon/2) \cdot k \ge (\epsilon/2) \cdot |c_y|$. As component c_y was migrated out of V_y, the number of overflow nodes in V_y changes by
$$\Delta\text{ovr}(V_y) = -\min\left\{\text{ovr}^{\sigma}(V_y), |c_y|\right\} \le -(\epsilon/2) \cdot |c_y| \ .$$
   Therefore, the second summand of (2) is at most $(4/\epsilon) \cdot \alpha \cdot \Delta\text{ovr}(V_y) \le -(4/\epsilon) \cdot \alpha \cdot (\epsilon/2) \cdot |c_y| = -2 \cdot |c_y| \cdot \alpha$ as desired.

As relation (1) holds for any sequence σ and the second summand of (1) is always non-negative, $\text{Crep}^{\text{mig}}(\sigma) \le (1 + 4/\epsilon) \cdot \alpha \cdot \sum_{m \in M(\sigma)} \text{size}(m)$. ∎

4.5. Analysis: Competitive Ratio. In the previous two subsections, we related $\text{Opt}(\sigma)$ to the total size of components that are deleted by Crep (cf. Lemma 4.5) and $\text{Crep}(\sigma)$ to $\sum_{m \in M(\sigma)} \text{size}(m)$, where the latter amount is related to the merging actions performed by Crep (cf. Lemma 4.7). Now we will link these two amounts. Note that each delete action corresponds to preceding real merge actions that led to the creation of the eventually deleted component.

Lemma 4.8. *For any input σ, $\sum_{m \in M(\sigma)} \text{size}(m) \le \sum_{c \in \text{del}(\sigma)} |c| \cdot \log k + \sum_{c \in \text{final}(\sigma)} |c| \cdot \log |c|$, where all logarithms are binary.*

Proof. We prove the lemma by induction on the sequence of merge and delete actions induced by input σ. At the very beginning, both sides of the lemma inequality are zero, and hence the induction basis holds trivially. We show that the lemma inequality is preserved at any integer time; we need to consider only times at which some merge actions occur.

First consider a time with a sequence of real merge actions. We show that the lemma inequality is preserved after processing each merge action. Let c_x and c_y be merged components, with sizes $p = |c_x|$ and $q = |c_y|$, where $p \le q$ without loss of generality. Due to such action, the right hand side of the lemma inequality increases by

$$\begin{aligned}(p + q) \cdot \log(p + q) - p \cdot \log p - q \cdot \log q &= p \cdot (\log(p + q) - \log p) + q \cdot (\log(p + q) - \log q) \\ &\ge p \cdot \log \frac{p + q}{p} \\ &\ge p \cdot \log 2 = p \ .\end{aligned}$$

As the left hand side of the inequality changes exactly by p, the inductive hypothesis holds.

Now consider a sequence of artificial merge actions (i.e., followed by a delete action) and let $c_1, c_2, \ldots, c_g$ denote components that were merged to create component c that was immediately deleted. Then, the right hand side of the lemma inequality changes by $-\sum_{i=1}^{g} |c_i| \cdot \log |c_i| + |c| \cdot \log k \ge -\sum_{i=1}^{g} |c_i| \cdot \log k + |c| \cdot \log k = 0$. As the left hand side of the lemma inequality is unaffected by artificial merge actions, the inductive hypothesis follows also in this case. ∎

THEOREM 4.9. *With augmentation at least* $2+\epsilon$, CREP *is* $O((1+1/\epsilon)\cdot k\cdot \log k)$*-competitive.*

Proof. Fix any input sequence σ. By Lemma 4.6 and Lemma 4.7,

$$\begin{aligned}\textsc{Crep}(\sigma) &= \textsc{Crep}^{\mathrm{mig}}(\sigma) + \textsc{Crep}^{\mathrm{req}}(\sigma)\\ &\le (1+4/\epsilon)\cdot\alpha\cdot \textstyle\sum_{m\in M(\sigma)} \textsc{size}(m) + |M(\sigma)|\cdot\alpha + w(\textsc{final}(\sigma))\ .\end{aligned}$$

Regarding a bound for $|M(\sigma)|$, we observe the following. First, if CREP executes artificial merge actions, then they are immediately followed by a delete action of the resulting component c. The number of artificial merge actions is clearly at most $|c|-1 \le |c|$, and thus the total number of all artificial actions in $M(\sigma)$ is at most $\sum_{c\in\textsc{del}(\sigma)} |c|$. Second, if CREP executes a real merge action m at some time t, then $\textsc{size}(m) \ge 1$. Combining these two bounds yields $|M(\sigma)| \le \sum_{m\in M(\sigma)} \textsc{size}(m) + \sum_{c\in\textsc{del}(\sigma)} |c|$. We use this inequality and later apply Lemma 4.8 to bound $\sum_{m\in M(\sigma)} \textsc{size}(m)$ obtaining,

$$\begin{aligned}\textsc{Crep}(\sigma) &\le (2+4/\epsilon)\cdot\alpha\cdot \textstyle\sum_{m\in M(\sigma)} \textsc{size}(m) + \alpha\cdot\sum_{c\in\textsc{del}(\sigma)}|c| + w(\textsc{final}(\sigma))\\ &\le (2+4/\epsilon)\cdot\alpha\cdot\Big(\alpha\cdot\textstyle\sum_{c\in\textsc{del}(\sigma)}|c|\cdot\log k + \sum_{c\in\textsc{final}(\sigma)}|c|\cdot\log|c|\Big) + \sum_{c\in\textsc{del}(\sigma)}|c| + w(\textsc{final}(\sigma))\\ &\le (4+8/\epsilon)\cdot\alpha\cdot\textstyle\sum_{c\in\textsc{del}(\sigma)}|c|\cdot\log k + (2+4/\epsilon)\cdot\alpha\cdot\sum_{c\in\textsc{final}(\sigma)}|c|\cdot\log|c| + w(\textsc{final}(\sigma))\ .\end{aligned}$$

Finally, Lemma 4.5 can be used to bound $\sum_{c\in\textsc{del}(\sigma)}|c|\cdot\alpha$ by $2k\cdot\textsc{Opt}(\sigma)$ yielding

$$\textsc{Crep}(\sigma) \le O(1+1/\epsilon)\cdot k\cdot\log k\cdot\textsc{Opt}(\sigma) + O(1+1/\epsilon)\cdot\alpha\cdot\textstyle\sum_{c\in\textsc{final}(\sigma)}|c|\cdot\log|c| + w(\textsc{final}(\sigma))\ .$$

To bound $w(\textsc{final}(\sigma))$, observe that the component-set $\textsc{final}(\sigma)$ contains at most $k\cdot\ell$ components, and hence by Lemma 4.1, $w(\textsc{final}(\sigma)) < k\cdot\ell\cdot\alpha$. Furthermore, the maximum of $\sum_{c\in\textsc{final}(\sigma)}|c|\cdot\log|c|$ is achieved when all nodes in a specific cluster constitute a single component. Thus, $\sum_{c\in\textsc{final}(\sigma)}|c|\cdot\log|c| \le \ell\cdot((2+\epsilon)\cdot k)\cdot\log((2+\epsilon)\cdot k) = O(\ell\cdot k\cdot\log k)$.

In total, $(1+1/\epsilon)\cdot\alpha\cdot\sum_{c\in\textsc{final}(\sigma)}|c|\cdot\log|c| + w(\textsc{final}(\sigma)) = O((1+1/\epsilon)\cdot\alpha\cdot\ell\cdot k\cdot\log k)$, i.e., it can be upper-bounded by a constant independent of input sequence σ, which concludes the proof. ∎

5. Online Rematching. Let us now consider the special case where clusters are of size two ($k=2$, arbitrary ℓ). This can be viewed as an online maximal (re)matching problem: clusters of size two contain ("match") exactly one pair of nodes, and maximizing pairwise communication within each cluster is equivalent to minimizing inter-cluster communication.

5.1. Greedy Algorithm. We define a natural greedy online algorithm GREEDY, parameterized by a real positive number λ. Similarly to our other algorithms, GREEDY maintains an edge weight for each pair of nodes. The weights of all edges are initially zero. Weights of intra-cluster edges are always zero and weights of inter-cluster edges are related to the number of paid communication requests between edge endpoints. However, GREEDY resets weights to zero not only when their endpoints become collocated.

More precisely, when facing an inter-cluster request between nodes x and y, GREEDY increments the weight $w(e)$, where $e=(x,y)$. Let x' and y' be the nodes collocated with x and y, respectively. If after the weight increase, it holds that $w(x,y)+w(x',y') \ge \lambda\cdot\alpha$, GREEDY performs a swap: it places x and y in one cluster and x' and y' in another; afterwards it resets the weights of edges (x,y) and (x',y') to 0. Finally, GREEDY pays for the request between x and y. Note that if the request triggered a migration, then GREEDY does not pay its communication cost.

5.2. Analysis. We use E to denote the set of all edges. Let M^{GR} (M^{OPT}) denote the set of all edges $e=(u,v)$, such that u and v are collocated by GREEDY (OPT). Note that M^{GR} and M^{OPT} are perfect matchings on the set of all nodes.

For the analysis, we associate the following edge-potential with any edge e:

$$\Phi(e) = \begin{cases} 0 & \text{if } e\in M^{\mathrm{GR}},\\ -w(e) & \text{if } e\in M^{\mathrm{OPT}}\setminus M^{\mathrm{GR}},\\ f\cdot w(e) & \text{if } e\notin M^{\mathrm{OPT}} \text{ and } e\notin M^{\mathrm{GR}},\end{cases}$$

where $f \geq 0$ is a constant that will be defined later.

The union of M^{GR} and M^{OPT} constitutes a set of alternating cycles: an alternating cycle of length $2j$ (for some $j \geq 1$) consists of $2j$ nodes, j edges from M^{GR} and j edges from M^{OPT}, interleaved. The case $j = 1$ is degenerate: such a cycle consists of a single edge from $M^{\mathrm{GR}} \cap M^{\mathrm{OPT}}$, but we still count it as a cycle of length 2. We define the cycle-potential as

$$\Psi = -C \cdot g \cdot \alpha,$$

where C is the number of all cycles and $g \geq 0$ is a constant that will be defined later.

To simplify the analysis, we slightly modify the way weights are increased by Greedy. The modification is applied only when the weight increment triggers a node migration. Recall that this happens when there is an inter-cluster request between nodes x and y. The corresponding weight $w(x, y)$ is then increased by 1. After the increase it holds that $w(x, y) + w(x', y') \geq \lambda \cdot \alpha$. (Nodes x' and y' are those collocated with x and y, respectively.) Instead, we increase $w(x, y)$ possibly by a smaller amount, so that $w(x, y) + w(x', y')$ becomes *equal* to $\lambda \cdot \alpha$. This modification allows for a more streamlined analysis and is local: before and after the modification, Greedy performs a migration and right after that resets weight $w(x, y)$ to zero.

We split processing of each request into three stages. In the first stage, Opt may perform an arbitrary number of migrations. (We may assume that Opt always migrates nodes before serving the requests.) In the second stage, both Greedy and Opt serve a request. In the third stage, Greedy may perform a node swap. By the modification described above, we may assume that at the beginning of the third stage, $w(x, y) + w(x', y') = \lambda \cdot \alpha$. Furthermore, within the first stage, all edge weights are at most $\lambda \cdot \alpha$.

We will show that for an appropriate choice of λ, f and g, for all three stages described above the following inequality holds:

$$\Delta\textsc{Greedy} + \Delta\Psi + \textstyle\sum_{e \in E} \Delta\Phi(e) \leq 7 \cdot \Delta\textsc{Opt}\ . \tag{3}$$

Here, ΔGreedy and ΔOpt denote the increases of Greedy's and Opt's cost, respectively. $\Delta\Psi$ and $\Delta\Phi(e)$ are the changes of the potentials Ψ and $\Phi(e)$. The 7-competitiveness then immediately follows from summing (3) and bounding the initial and final values of the potentials.

Lemma 5.1. *If $2 \cdot (f + 1) \cdot \lambda + g \leq 14$, then (3) holds for the first stage.*

Proof. We consider any node swap performed by Opt. Clearly, for such an event ΔGreedy $= 0$ and ΔOpt $= 2 \cdot \alpha$. The number of cycles decreases at most by one, and thus $\Delta\Psi \leq g \cdot \alpha$.

We will now upper-bound the change in the edge-potentials. Let e_1^{old} and e_2^{old} be the edges that were removed from M^{OPT} by the swap and let e_1^{new} and e_2^{new} be the edges added to M^{OPT}. For any $i \in \{1, 2\}$, $\Delta\Phi(e_i^{\mathrm{new}}) \leq 0$ as the initial value of $\Phi(e_i^{\mathrm{new}})$ is at least 0 and the final value of $\Phi(e_i^{\mathrm{new}})$ is at most 0. Similarly, $\Delta\Phi(e_i^{\mathrm{old}}) \leq (f + 1) \cdot w(e_i^{\mathrm{old}})$ as the initial value of $\Phi(e_i^{\mathrm{old}})$ is at least $-w(e_i^{\mathrm{old}})$ and the final value of $\Phi(e_i^{\mathrm{old}})$ is at most $f \cdot w(e_i^{\mathrm{old}})$.

Summing up, $\sum_{e \in E} \Delta\Phi \leq (f + 1) \cdot (w(e_1^{\mathrm{old}}) + w(e_2^{\mathrm{old}})) \leq 2 \cdot (f + 1) \cdot \lambda \cdot \alpha$ as the weight of each edge is at most $\lambda \cdot \alpha$. By combining the bounds above and using the lemma assumption, we obtain ΔGreedy $+ \sum_{e \in E} \Delta\Phi(e) + \Delta\Psi \leq 0 + 2 \cdot (f + 1) \cdot \lambda \cdot \alpha + g \cdot \alpha \leq 14 \cdot \alpha = 7 \cdot \Delta$Opt. ∎

Lemma 5.2. *If $f \leq 6$, then (3) holds for the second stage.*

Proof. In this stage, both Greedy and Opt serve a communication request between nodes x and y. Let $e_c = (x, y)$. Neither Greedy nor Opt migrates any nodes in this stage. Hence, the structure of alternating cycles remains unchanged, i.e., $\Delta\Psi = 0$. Futhermore, only edge e_c may change its weight, and therefore, among all edges, only the edge-potential of e_c may change. We consider three cases.

1. If $e_c \in M^{\mathrm{GR}}$, then ΔGreedy $= 0$ and ΔOpt ≥ 0. As $w(e_c)$ is unchanged, $\Delta\Phi(e_c) = 0$, and therefore ΔGreedy $+ \Delta\Phi(e_c) = 0 = \Delta$Opt.
2. If $e_c \notin M^{\mathrm{GR}}$, then let $\Delta w(e_c) \leq 1$ denote the increase of the weight of edge e_c. Note that ΔGreedy $\leq \Delta w(e_c)$: either no migration is triggered and ΔGreedy $= \Delta w(e_c) = 1$ or a migration is triggered and then Greedy does not pay for the request.

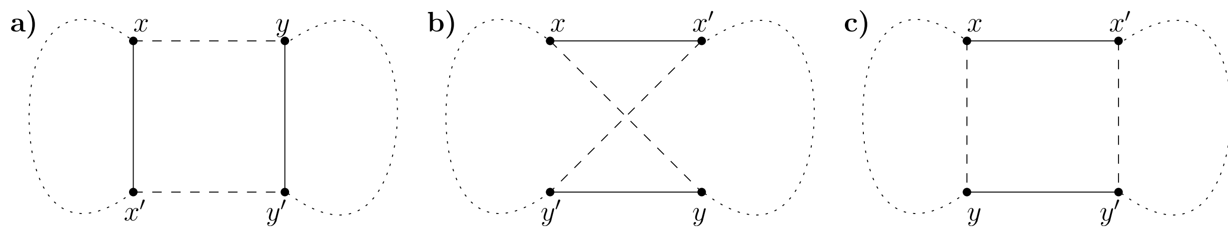

FIG. 1. *Three cases in the analysis of the third stage (a swap performed by* GREEDY*). Solid edges denote edges that were removed from M^{GR} because of the swap, dashed ones denote the ones that were added to M^{GR}. Dotted paths denote the remaining parts of the involved alternating cycle(s).*

If $e_c \in M^{\mathrm{OPT}}$, then $\Delta\text{OPT} = 0$ and $\Delta\Phi(e_c) = -w(e_c)$. Thus, $\Delta\text{GREEDY} + \Delta\Phi(e_c) \leq 0 = \Delta\text{OPT}$. Otherwise, $e_c \notin M^{\mathrm{OPT}}$, in which case $\Delta\text{OPT} = 1$. Furthermore $\Delta\Phi(e_c) = f \cdot \Delta w(e_c)$, and thus $\Delta\text{GREEDY} + \Delta\Phi(e_c) = (f+1)\cdot w(e_c) \leq f + 1 = (f+1)\cdot\Delta\text{OPT}$.

Therefore, in the second stage, $\Delta\text{GREEDY} + \Delta\Psi + \sum_{e\in E}\Delta\Phi(e) \leq (f+1)\cdot\Delta\text{OPT}$, which implies (3) as we assumed $f \leq 6$. ∎

LEMMA 5.3. *If $2 + \lambda \leq g \leq f\cdot\lambda - 2$, then (3) holds for the third stage.*

Proof. In the third stage (if it is present), GREEDY performs a swap. Clearly, for such an event $\Delta\text{GREEDY} = 2\cdot\alpha$ and $\Delta\text{OPT} = 0$.

There are four edges involved in a swap: let (x, x') and (y, y') be the edges that were in M^{GR} before the swap and let (x, y) and (y, y') be the new edges in M^{GR} after the swap. Note that $w(x, x') = w(y, y') = 0$ before and after the swap. By the definition of GREEDY and our modification of weight updates, $w(x, y) + w(x', y') = \lambda\cdot\alpha$ before the swap, and after the swap these weights are reset to zero.

For any edge e, let $w^{\mathrm{S}}(e)$ and $\Phi^{\mathrm{S}}(e)$ denote the weight and the edge-potential of e right before the swap. By the bounds above, $\Delta\text{GREEDY} + \sum_{e\in E}\Delta\Phi(e) + \Delta\Psi = 2\cdot\alpha - \Phi^{\mathrm{S}}(x, y) - \Phi^{\mathrm{S}}(x', y') + \Delta\Psi$, and hence to show (3) it suffices to show that the latter amount is at most $7\cdot\Delta\text{OPT} = 0$. We consider three cases.

1. Assume that edges (x, x') and (y, y') were in different alternating cycles before the swap, see Figure 1a. Then the number of alternating cycles decreases by one, and hence $\Delta\Psi = g\cdot\alpha$. Let C be the cycle that contained edge (x, x'). Node x is adjacent to an edge from C that belongs to M^{OPT}. (It is possible that this edge is (x, x'); this occurs in the degenerate case when C is of length 2.) As M^{OPT} is a matching, $(x, y) \notin M^{\mathrm{OPT}}$. Analogously, $(x', y') \notin M^{\mathrm{OPT}}$. Therefore, $\Phi^{\mathrm{S}}(x, y) + \Phi^{\mathrm{S}}(x', y') = f\cdot w(x, y) + f\cdot w(x', y') = f\cdot\lambda\cdot\alpha$. Using the lemma assumption, $\Delta\text{GREEDY} + \sum_{e\in E}\Delta\Phi(e) + \Delta\Psi = (2 + g - f\cdot\lambda)\cdot\alpha \leq 0$.
2. Assume that edges (x, x') and (y, y') belonged to the same cycle and it contained the nodes in the order $x, x', \ldots, y, y', \ldots$, see Figure 1b. In this case it holds that $\Delta\Psi = 0$, since the number of alternating cycles is unaffected by the swap. By similar reasoning as in the previous case, neither (x, y) nor (x', y') belong to M^{OPT}, and thus again, $\Phi^{\mathrm{S}}(x, y) + \Phi^{\mathrm{S}}(x', y') = f\cdot w(x, y) + f\cdot w(x', y') = f\cdot\lambda\cdot\alpha$. In this case, $\Delta\text{GREEDY} + \sum_{e\in E}\Delta\Phi(e) + \Delta\Psi = (2 - f\cdot\lambda)\cdot\alpha \leq (2 + g - f\cdot\lambda)\cdot\alpha \leq 0$.
3. Assume that edges (x, x') and (y, y') belonged to the same cycle and this cycle contained the nodes in the order $x, x', \ldots, y', y, \ldots$, see Figure 1c. When the swap is performed, the number of alternating cycles decreases, and thus $\Delta\Psi = -g\cdot\alpha$. Unlike the previous cases, here it is possible that (x, y) and (x', y') belong to M^{OPT}. But even in such a case, we may lower-bound the initial values of the corresponding edge-potentials: $\Phi^{\mathrm{S}}(x, y) + \Phi^{\mathrm{S}}(x', y') \geq -w^{\mathrm{S}}(x, y) - w^{\mathrm{S}}(x', y') = -\lambda\cdot\alpha$. Using the lemma assumption, $\Delta\text{GREEDY} + \sum_{e\in E}\Delta\Phi(e) + \Delta\Psi = (2 - g + \lambda)\cdot\alpha \leq 0$. ∎

THEOREM 5.4. *For $\lambda = 4/5$,* GREEDY *is 7-competitive.*

Proof. We choose $f = 6$ and $g = 14/5$. The chosen values of λ, f and g satisfy the conditions of Lemma 5.2, Lemma 5.3 and Lemma 5.1. Summing these inequalities over all stages occuring while serving an input sequence σ yields

$$\textsc{Greedy}(\sigma) + (\Psi_{\text{final}} - \Psi_{\text{initial}}) + \textstyle\sum_{e \in E} (\Phi_{\text{final}}(e) - \Phi_{\text{initial}}(e)) \leq 7 \cdot \textsc{Opt}(\sigma) \ ,$$

where Ψ_{final} and $\Phi_{\text{final}}(e)$ denote the final values of the potentials and Ψ_{initial} and $\Phi_{\text{initial}}(e)$ their initial values. We observe that all the potentials occuring in the inequality above are lower-bounded and upper-bounded by values that are independent of the input sequence σ. That is, $\Psi_{\text{final}} - \Psi_{\text{initial}} \geq -g \cdot \ell \cdot \alpha$ (as the number of alternating cycles is at most ℓ) and $\Phi_{\text{final}}(e) - \Phi_{\text{initial}}(e) \geq -(f+1) \cdot w(e) \geq -(f+1) \cdot \lambda \cdot \alpha$ (as all edge weights are always at most $\lambda \cdot \alpha$). The number of edges is exactly $\binom{2 \cdot \ell}{2}$, and therefore

$$\begin{aligned} \textsc{Greedy}(\sigma) &\leq 7 \cdot \textsc{Opt}(\sigma) + g \cdot \ell \cdot \alpha + \tbinom{2 \cdot \ell}{2} \cdot (f+1) \cdot \lambda \cdot \alpha \\ &\leq 7 \cdot \textsc{Opt}(\sigma) + O(\ell^2 \cdot \alpha) \ . \end{aligned}$$

This concludes the proof. ∎

6. Lower Bounds.

In order to shed light on the optimality of the presented online algorithm, we next investigate lower bounds on the competitive ratio achievable by any (deterministic) online algorithm. We start by showing a reduction of the BRP problem to online paging, which will imply that already for two clusters the competitive ratio of the problem is at least $k-1$. We strengthen this bound, providing a lower bound of k that holds for any amount of augmentation, as long as the augmentation is less than what would be required to solve the partitioning problem trivially: by putting all nodes into the same cluster. The proof uses the averaging argument. We refine this approach for a special case of online rematching ($k = 2$ without agumentation), for which we present a lower bound of 3.

6.1. Lower Bound by Reduction to Online Paging.

Theorem 6.1. *Fix any k. If there exist a γ-competitive deterministic algorithm B for BRP for two clusters, each of size k, then there exists a γ-competitive deterministic algorithm P for the paging problem with cache size $k-1$ and where the number of different pages is k.*

Proof. The pages are denoted by $p_1, p_2, \ldots, p_k$. Without loss of generality, we assume that the initial cache is equal to $p_1, p_2, \ldots, p_{k-1}$. We fix any input sequence $\sigma^P = \sigma^P_1, \sigma^P_2, \sigma^P_3, \ldots$ for the paging problem, where σ^P_t denotes the t-th accessed page. We show how to construct, in an online manner, an online algorithm P for the paging problem that operates in the following way. It internally runs the algorithm B, starting on the initial assignment of nodes to clusters that will be defined below. For a requested page σ^P_t, it creates a subsequence of communication requests for the BRP problem, runs B on them, and serves σ^P_t on the basis of B's responses.

We use the following $2k$ nodes for the BRP problem: paging nodes $p_1, p_2, \ldots, p_k$, auxiliary nodes $a_1, a_2, \ldots, a_{k-1}$, and a special node s. We say that the node clustering is *well aligned* if one cluster contains the node s and $k-1$ paging nodes, and the other cluster contains one paging node and all auxiliary nodes. There is a natural bijection between possible cache contents and well aligned configurations: the cache consists of the $k-1$ paging nodes that are in the same cluster as node s. (Without loss of generality, we may assume that the cache of any paging algorithm is always full, i.e., consists of $k-1$ pages.) If the configuration c of a BRP algorithm is well aligned, $\textsc{cache}(c)$ denotes the corresponding cache contents.

The initial configuration for the BRP problem is the well aligned configuration corresponding to the initial cache (pages $p_1, p_2, \ldots, p_{k-1}$ in the cache).

For any paging node p, let $\textsc{comm}(p)$ be a subsequence of communication requests for the BRP problem, consisting ot the request (p, s), followed by $\binom{k-1}{2}$ requests to all pairs of auxiliary nodes. Given an input sequence σ^P, we construct the input sequence σ^B for the BRP problem in the following way: For a request σ^P_t, we repeat a subsequence $\textsc{comm}(\sigma^P_t)$ till the node clustering maintained by B becomes well aligned and σ^P_t becomes collocated with s. Note that B must eventually achieve such a node configuration: otherwise its cost would be arbitrarily large while a sequence of repeated $\textsc{comm}(\sigma^P_t)$ subsequences can be served at a constant cost—the competitive ratio of B would be unbounded. We denote the resulting sequence of $\textsc{comm}(\sigma^P_t)$ subsequences by $\textsc{comm}_t(\sigma^P_t)$.

To construct the response to the paging request σ_t^P, the algorithm P runs B on $\textsc{comm}_t(\sigma_t^P)$. Right after processing $\textsc{comm}_t(\sigma_t^P)$, node configuration c of B is well alligned and σ_t^P is collocated with s. Hence, P may change its cache configuration to $\textsc{cache}(c)$: such a response is feasible because since σ_t^P is collocated with s, it is included by P in the cache. Furthermore, we may relate the cost of P to the cost of B: If P modifies the cache contents, the corresponding cost is 1, as exactly one page has to be fetched. Such a change occurs only if B changed node placement in clusters (at a cost of at least $2 \cdot \alpha$). Therefore, $2 \cdot \alpha \cdot P(\sigma_t^P) \leq B(\textsc{comm}_t(\sigma_t^P))$, which summed over all requests from sequence σ^P yields $2 \cdot \alpha \cdot P(\sigma^P) \leq B(\sigma^B)$.

Now we show that there exists an (offline) solution $\textsc{Off}$ to σ^B, whose cost is exactly $2 \cdot \alpha \cdot \textsc{Opt}(\sigma^P)$. Recall that, for a paging request σ_t^P, σ^B contains the corresponding sequence $\textsc{comm}_t(\sigma_t^P)$. Before serving the first request of $\textsc{comm}_t(\sigma_t^P)$, $\textsc{Off}$ changes its state to a well aligned configuration corresponding to the cache of $\textsc{Opt}$ right after serving paging request σ_t^P. This ensures that the subsequence $\textsc{comm}_t(\sigma_t^P)$ is free for $\textsc{Off}$. Furthermore, the cost of node migration of $\textsc{Off}$ is 2α (two paging nodes are swapped) if $\textsc{Opt}$ performs a fetch, and 0 if $\textsc{Opt}$ does not change its cache contents. Therefore, $\textsc{Off}(\textsc{comm}_t(\sigma_t^P)) = 2 \cdot \alpha \cdot \textsc{Opt}(\sigma_t^P)$, which summed over the entire sequence σ^P yields $\textsc{Off}(\sigma^B) = 2 \cdot \alpha \cdot \textsc{Opt}(\sigma^P)$.

As B is ρ-competitive for the BRP problem, there exists a constant β, such that for any sequence σ^P and the corresponding sequence σ^B, it holds that $B(\sigma^B) \leq \gamma \cdot \textsc{Opt}(\sigma^B) + \beta$. Combining this inequality with proven relations between P and B and between $\textsc{Off}$ and $\textsc{Opt}$,

$$2 \cdot \alpha \cdot P(\sigma^P) \leq B(\sigma^B) \leq \gamma \cdot \textsc{Opt}(\sigma^B) + \beta \leq \gamma \cdot \textsc{Off}(\sigma^B) + \beta = \gamma \cdot 2 \cdot \alpha \cdot \textsc{Opt}(\sigma^P) + \beta \ ,$$

and therefore P is γ-competitive. ∎

As any deterministic algorithm for the paging problem with cache size $k-1$ has a competitive ratio of at least $k-1$ [33], we obtain the following result.

Corollary 6.2. *The competitive ratio of the BRP problem on two clusters is at least $k-1$.*

6.2. Additional Lower Bounds.

Theorem 6.3. *No δ-augmented deterministic online algorithm* $\textsc{Onl}$ *can achieve a competitive ratio smaller than k, as long as $\delta < \ell$.*

Proof. In our construction, all nodes are numbered from v_0 to v_{n-1}. All presented requests are edges in a ring graph on these nodes with edge e_i defined as $(v_i, v_{(i+1) \bmod n})$ for $i = 0, \ldots, n-1$. At any time, the adversary gives a communication request between an arbitrary pair of nodes not collocated by $\textsc{Onl}$. As $\delta < \ell$, $\textsc{Onl}$ cannot fit the entire ring in a single cluster, and hence such pair always exists, and such a request entails a cost of at least 1 for $\textsc{Onl}$. This way, we may define an input sequence σ of an arbitrary length, such that $\textsc{Onl}(\sigma) \geq |\sigma|$.

Now we present k offline algorithms $\textsc{Off}_1, \textsc{Off}_2, \ldots, \textsc{Off}_k$, such that, neglecting an initial node reorganization that they are allowed to perform before the input sequence starts, the sum of their total costs on σ is exactly $|\sigma|$. Toward this end, for any $j \in \{0, \ldots, k-1\}$, we define a set $\textsc{cut}(j) = \{e_j, e_{j+k}, e_{j+2k}, \ldots, e_{j+(\ell-1)\cdot k}\}$. For any j, set $\textsc{cut}(j)$ defines a natural partitioning of ring nodes (and hence also all nodes) into clusters, each containing k nodes. Before processing σ, the algorithm $\textsc{Off}_j$ first migrates its nodes (paying at most $n \cdot \alpha$) to the clustering defined by $\textsc{cut}(j)$ and then never changes the node placement.

As all sets $\textsc{cut}(j)$ are pairwise disjoint, for any request σ_t, exactly one algorithm $\textsc{Off}_j$ pays for the request, and thus $\sum_{j=1}^{k} \textsc{Off}_j(\sigma_t) = 1$. Therefore, taking the initial node reorganization into account, we obtain that $\sum_{j=1}^{k} \textsc{Off}_j(\sigma) \leq k \cdot n \cdot \alpha + \textsc{Onl}(\sigma)$. By the averaging argument, there exists offline algorithm $\textsc{Off}_j$, such that $\textsc{Off}_j(\sigma) \leq \frac{1}{k} \cdot \sum_{j=1}^{k} \textsc{Off}_j(\sigma) \leq n \cdot \alpha + \textsc{Onl}(\sigma)/k$. Therefore, $\textsc{Onl}(\sigma) \geq k \cdot \textsc{Off}_j(\sigma) - k \cdot n \cdot \alpha \geq k \cdot \textsc{Opt}(\sigma) - k \cdot n \cdot \alpha$. The theorem follows because the additive constant $k \cdot n \cdot \alpha$ becomes negligible as the length of σ grows. ∎

Theorem 6.4. *No deterministic online algorithm* $\textsc{Onl}$ *can achieve a competitive ratio smaller than* 3 *for the case $k = 2$ (without augmentation).*

Proof. As in the previous proof, we number the nodes from v_0 to v_{n-1}. We distinguish three types of node clusterings. Configuration A: v_0 collocated with v_1, v_2 collocated with v_3, other nodes collocated arbitrarily;

configuration B: v_1 collocated with v_2, v_3 collocated with v_0, other nodes collocated arbitrarily; configuration C: all remaining clusterings.

Similarly to the proof of Theorem 6.3, the adversary always requests a communication between two nodes not collocated by Onl. This time the exact choice of such nodes is relevant: Onl receives request to (v_1, v_2) in configuration A, and to (v_0, v_1) in configurations B and C.

We define three offline algorithms. They will keep nodes $\{v_0, \ldots, v_3\}$ in the first two clusters and the remaining nodes in the remaining clusters (the remaining nodes will never change their clusters). More concretely, Off_1 keeps nodes $\{v_0, \ldots, v_3\}$ always in configuration A and Off_2 always in configuration B. Furthermore, we define the third algorithm Off_3 that is in configuration B if Onl is in configuration A, and is in configuration A if Onl is in configuration B or C.

We split the cost of Onl into the cost for serving requests, Onl^{req}, and the cost paid for its migrations, Onl^{mig}. Observe that, for any request σ_t, $\text{Off}_1(\sigma_t) + \text{Off}_2(\sigma_t) = \text{Onl}^{\text{req}}(\sigma_t)$. Moreover, as Off_3 does not pay for any request and migrates at the same time as Onl does, $\text{Off}_3(\sigma_t) = \text{Onl}^{\text{mig}}(\sigma_t)$. Summing up, $\sum_{j=1}^{3} \text{Off}_j(\sigma_t) = \text{Onl}(\sigma_t)$ for any request σ_t. Taking into account the initial reconfiguration of nodes in Off_j solutions (which involves at most one swap of cost $2 \cdot \alpha$), we obtain that $\sum_{j=1}^{3} \text{Off}(\sigma) \leq 2 \cdot \alpha + \text{Onl}(\sigma)$. Hence, by the averaging argument, there exists $j \in \{1, 2, 3\}$, such that $\text{Onl}(\sigma) \geq 3 \cdot \text{Off}_j(\sigma) - 2 \cdot \alpha \geq 3 \cdot \text{Opt}(\sigma) - 2 \cdot \alpha$. This concludes the proof, as $2 \cdot \alpha$ becomes negligible as the length of σ grows. ∎

7. Conclusion.

This paper initiated the study of a natural dynamic partitioning problem which finds applications, e.g., in the context of virtualized distributed systems subject to changing communication patterns. We derived upper and lower bounds, both for the general case as well as for a special case related to a dynamic matching problem. The natural research direction is to develop better deterministic algorithms for the non-augmented variant of the general case, improving over the straightforward $O(k^2 \cdot \ell^2)$-competitive algorithm given in Section 3. While the linear dependency on k is inevitable (cf. Section 6), it is not known whether an algorithm whose competitive ratio is independent of ℓ is possible. We resolved this issue for the $O(1)$-augmented variant, for which we gave an $O(k \log k)$-competitive algorithm.

REFERENCES

[1] D. Achlioptas, M. Chrobak, and J. Noga, *Competitive analysis of randomized paging algorithms*, Theoretical Computer Science, 234 (2000), pp. 203–218.
[2] K. Andreev and H. Räcke, *Balanced graph partitioning*, Theory of Computing Systems, 39 (2006), pp. 929–939.
[3] S. Arora, D. R. Karger, and M. Karpinski, *Polynomial time approximation schemes for dense instances of NP-hard problems*, Journal of Computer and System Sciences, 58 (1999), pp. 193–210.
[4] S. Arora, S. Rao, and U. Vazirani, *Expander flows, geometric embeddings and graph partitioning*, Journal of the ACM, 56 (2009), p. 5.
[5] B. Awerbuch, Y. Azar, and Y. Bartal, *On-line generalized Steiner problem*, Theoretical Computer Science, 324 (2004), pp. 313–324.
[6] B. Awerbuch, Y. Bartal, and A. Fiat, *Competitive distributed file allocation*, in Proc. 25th ACM Symp. on Theory of Computing (STOC), 1993, pp. 164–173.
[7] B. Awerbuch, Y. Bartal, and A. Fiat, *Distributed paging for general networks*, Journal of Algorithms, 28 (1998), pp. 67–104.
[8] Y. Bartal, M. Charikar, and P. Indyk, *On page migration and other relaxed task systems*, Theoretical Computer Science, 268 (2001), pp. 43–66.
[9] Y. Bartal, A. Fiat, and Y. Rabani, *Competitive algorithms for distributed data management*, Journal of Computer and System Sciences, 51 (1995), pp. 341–358.
[10] M. Bienkowski, J. Byrka, and M. Mucha, *Dynamic beats fixed: On phase-based algorithms for file migration*, in Proc. 44th Int. Colloq. on Automata, Languages and Programming (ICALP), 2017, pp. 13:1–13:14.
[11] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, *Managing data transfers in computer clusters with Orchestra*, in ACM SIGCOMM, 2011, pp. 98–109.
[12] L. Epstein, C. Imreh, A. Levin, and J. Nagy-György, *On variants of file caching*, in Proc. 38th Int. Colloq. on Automata, Languages and Programming (ICALP), 2011, pp. 195–206.
[13] L. Epstein, C. Imreh, A. Levin, and J. Nagy-György, *Online file caching with rejection penalties*, Algorithmica, 71 (2015), pp. 279–306.
[14] G. Even, J. Naor, S. Rao, and B. Schieber, *Fast approximate graph partitioning algorithms*, SIAM Journal on Computing, 28 (1999), pp. 2187–2214.
[15] G. Even, J. Naor, S. Rao, and B. Schieber, *Divide-and-conquer approximation algorithms via spreading metrics*, Journal of the ACM, 47 (2000), pp. 585–616.
[16] U. Feige and R. Krauthgamer, *A polylogarithmic approximation of the minimum bisection*, SIAM Journal on Computing, 31 (2002), pp. 1090–1118.

[17] U. Feige, R. Krauthgamer, and K. Nissim, *Approximating the minimum bisection size (extended abstract)*, in Proc. 32nd ACM Symp. on Theory of Computing (STOC), 2000, pp. 530–536.
[18] A. E. Feldmann and L. Foschini, *Balanced partitions of trees and applications*, Algorithmica, 71 (2015), pp. 354–376.
[19] M. C. S. Felice, D. P. Williamson, and O. Lee, *A randomized O(log n)-competitive algorithm for the online connected facility location problem*, Algorithmica, 76 (2016), pp. 1139–1157.
[20] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young, *Competitive paging algorithms*, Journal of Algorithms, 12 (1991), pp. 685–699.
[21] M. R. Garey, D. S. Johnson, and L. J. Stockmeyer, *Some simplified np-complete graph problems*, tcs, 1 (1976), pp. 237–267.
[22] S. Irani, *Page replacement with multi-size pages and applications to web caching*, Algorithmica, 33 (2002), pp. 384–409.
[23] A. R. Karlin, M. S. Manasse, L. A. McGeoch, and S. Owicki, *Competitive randomized algorithms for non-uniform problems*, Algorithmica, 11 (1994), pp. 542–571.
[24] R. Krauthgamer and U. Feige, *A polylogarithmic approximation of the minimum bisection*, SIAM Review, 48 (2006), pp. 99–130.
[25] R. Krauthgamer, J. Naor, and R. Schwartz, *Partitioning graphs into balanced components*, in Proc. 20th ACM-SIAM Symp. on Discrete Algorithms (SODA), 2009, pp. 942–949.
[26] T. Leighton, F. Makedon, and S. G. Tragoudas, *Approximation algorithms for vlsi partition problems*, in IEEE International Symposium on Circuits and Systems, vol. 4, 1990, pp. 2865–2868.
[27] Z. Lotker, B. Patt-Shamir, and D. Rawitz, *Rent, lease or buy: Randomized algorithms for multislope ski rental*, in Proc. 25th Symp. on Theoretical Aspects of Computer Science (STACS), 2008, pp. 503–514.
[28] L. A. McGeoch and D. D. Sleator, *A strongly competitive randomized paging algorithm*, Algorithmica, 6 (1991), pp. 816–825.
[29] J. C. Mogul and L. Popa, *What we talk about when we talk about cloud network performance*, ACM SIGCOMM Computer Communication Review, 42 (2012), pp. 44–48.
[30] H. Räcke, *Optimal hierarchica decompositions for congestion minimization in networks*, in Proc. 40th ACM Symp. on Theory of Computing (STOC), 2008, pp. 255–264.
[31] H. Saran and V. V. Vazirani, *Finding k cuts within twice the optimal*, SIAM Journal on Computing, 24 (1995), pp. 101–108.
[32] H. D. Simon and S. Teng, *How good is recursive bisection?*, SIAM Journal on Computing, 18 (1997), pp. 1436–1445.
[33] D. D. Sleator and R. E. Tarjan, *Amortized efficiency of list update and paging rules*, Communications of the ACM, 28 (1985), pp. 202–208.
[34] S. Umboh, *Online network design algorithms via hierarchical decompositions*, in Proc. 26th ACM-SIAM Symp. on Discrete Algorithms (SODA), 2015, pp. 1373–1387.