Dynamic Balanced Graph Clustering*

Marcin Bienkowski[1], Chen Avin[2], Andreas Loukas[3], Maciej Pacut[1], and Stefan Schmid[4,3]

[1]University of Wroclaw, Poland
[2]Ben Gurion University of the Negev, Israel
[3]TU Berlin, Germany
[4]Aalborg University, Denmark

September 16, 2017

This paper initiates the study of deterministic algorithms for collocating frequently communicating nodes in a distributed networked systems in an online fashion. We introduce the *Balanced RePartitioning* (BRP) problem: Given an arbitrary sequence of pairwise communication requests between n nodes, with patterns that may change over time, the objective is to dynamically partition the nodes into ℓ clusters, each of size k, at a minimum cost. The partitioning can be updated dynamically by *migrating* nodes between clusters. The goal is to devise online algorithms which minimize the sum of the communication and the migration cost.

BRP features interesting connections to other well-known online problems. In particular, scenarios with $\ell = 2$ generalize online paging, and scenarios with $k = 2$ constitute a novel online variant of maximum matching. We consider settings both with and without cluster-size augmentation. Somewhat surprisingly (and unlike online paging), we prove that any deterministic online algorithm has a competitive ratio of at least k, even with augmentation. Our main technical contribution is an $O(k \log k)$-competitive deterministic algorithm for the setting with a constant augmentation. This is attractive as, in contrast to ℓ, k is likely to be small in practice. For the case $k = 2$, we present a constant competitive algorithm that does not rely on augmentation.

Keywords: 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,

*Research supported by the German-Israeli Foundation for Scientific Research (GIF) Grant I-1245-407.6/2014 and by the Polish National Science Centre grant 2016/22/E/ST6/00499. Preliminary version of this paper appeared as "Online Balanced Repartitioning" in the proceedings of 30th International Symposium on DIStributed Computing (DISC 2016).

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. One of the applications is the scenario, where n virtual machines are distributed across ℓ physical servers, each having k cores, i.e., capable of running k virtual machines.

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.

1.1 The Model

Formally, the *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 a time $t \geq 1$, an online algorithm needs to serve the communication request (u_t, v_t), but right before or right after serving it can also repartition the nodes into new clusters. We assume that a communication request between two nodes located in different clusters costs 1, a communication request between two collocated nodes costs 0, and migrating a node from one cluster to another costs $\alpha \geq 1$, where α is an integer. For any algorithm Alg, we denote its total cost (consisting of communication plus migration costs) on sequence σ by $\textsc{Alg}(\sigma)$. We consider two 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 typical in the competitive analysis, the augmented online algorithm is compared to the optimal offline algorithm with cluster capacity k.

Let $\textsc{Onl}(\sigma)$ and $\textsc{Opt}(\sigma)$ be the cost induced by σ on an online algorithm Onl and an optimal offline algorithm Opt, respectively. In contrast to Onl, which learns the requests one-by-one as it serves them, 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, 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 ONL is ρ-competitive is called the competitive ratio of ONL.

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). As it turns out, BRP features some interesting connections to other well-known online online graph problems. For $\ell = 2$, BRP is able to simulate online paging problem (see below) 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 present a simple $O(k^2 \cdot \ell^2)$-competitive algorithm. Our main technical contribution however is an $O((1 + 1/\epsilon) \cdot k \log k)$-competitive deterministic algorithm CREP for $(2+\epsilon)$-augmentation setting, given in Section 4. We emphasize that this bound does not depend on ℓ. This is interesting, as for example, in our motivating virtual machine collocation problem, k is likely to be small: a server typically hosts 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 [ST85].

1.3 Practical Motivation

One practical motivation for our problem arises in the context of 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 [MP12]. For example, traces of jobs from a Facebook cluster reveal that network transfers on average account for 33% of the execution time [CZM+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., without migration, where the goal is to find best node assignment to ℓ clusters and all requests are known in advance, is the ℓ-balanced graph partitioning problem. It is known to be NP-complete, and cannot even be approximated within any finite factor unless P = NP [AR06]. 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 [GJS76]. Its approximation was studied in a long line of work [SV95, AKK99, FKN00, FK02, KF06, Räc08] and the currently best approximation ratio of $O(\log n)$ was given by Räcke [Räc08]. The $O(\log^{3/2} n)$-approximation given by Krauthgamer and Feige [KF06] 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 a research of bicriteria variant, which is an offline counterpart of our cluster-size augmentation. That is, 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 when $\delta \geq 2$ was considered in [LMT90, ST97, ENRS00, ENRS99, KNS09]. So far the best result is an $O(\sqrt{\log n \cdot \log \ell})$-approximation by Krauthgamer et al. [KNS09], which builds on ideas from $O(\sqrt{\log n})$-aproximation algorithm for balanced cuts by Arora et al. [ARV09]. 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 [AR06], which was later improved to $O(\log n)$ by Feldmann and Foschini [FF15].

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 [ST85, FKL+91, MS91, ACN00], 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. In this classic variant, the problem usually boils 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* [EILNG11, EILN15, Ira02]. Nonetheless, we show that BRP is capable of emulating online paging.

The BRP problem is an example of a non-uniform problem [KMMO94]: 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 [KMMO94, LPSR08], relaxed metrical

task systems [BCI01], file migration [BCI01, BBM17], distributed data management [BFR95, ABF93, ABF98], rent-or-buy network design [AAB04, Umb15, FWL16].

There are two major differences between BRP and problems listed above. First, these problems typically maintain some configuration of servers or bought infrastructure and upon a new request (whose cost typically depend 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 the set of such configurations can be very large, but such configurations may differ a lot 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 od nodes. We say that a communication request is *paid* (by Det) if it occurs between nodes from different clusters, and thus incurs a cost to 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. 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 on phase beginning.

Theorem 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 are kept within a single cluster till 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. 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 on, 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 edge incurred on 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, $\textsc{Det}(\sigma) \leq O(k^2 \cdot \ell^2) \cdot \textsc{Opt}(\sigma) + O(k^2 \cdot \ell^2 \cdot \alpha)$ 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. However, the runtime of Crep is not partitioned into phases and the reset of counters to zero does not occur globally.

4.1 Algorithm definition

We state 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 od 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 incurs a cost to 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 component-set *trivial* if it contains only one component; $w(S) = 0$ in such case.

Initially, all components are singleton components and all edge weights are zero. At time t, upon a communication request between 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 the average 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 2. *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 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 reseting 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.

As only one request is given at a time, all weights and α are integers, Lemma 2 immediately implies the following result.

Corollary 3. *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 $S(c)$. 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 to belong to some previous component b till 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 3, the number of these requests that incurred 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 on Opt by requests contained in $F(c)$ plus the cost of Opt migrations contained in $F(c)$. The total cost of Opt can be then 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. *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 3, $w(S(c)) = (|S(c)| - 1) \cdot \alpha$ 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 5. *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, 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 bundles definition, it involves a communication between two nodes stored in different clusters by Opt. Additionally, Opt(c) involves also $\sum_{b \in S(c)}$ 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 6. *For any input σ, let* $\textsc{del}(\sigma)$ *be the set of components that are eventually deleted by* Crep. *Then* $\textsc{Opt}(\sigma) \geq \sum_{c \in \textsc{del}(\sigma)} |c|/(2k) \cdot \alpha$.

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 5 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 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 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 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 Crep. Let $\textsc{final}(\sigma)$ be the set of all components that exist when 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, $\textsc{Crep}^{\text{req}}(\sigma)$, and the cost of node migrations, $\textsc{Crep}^{\text{mig}}(\sigma)$.

Lemma 7. *For any input σ,* $\textsc{Crep}^{\text{req}}(\sigma) = |M(\sigma)| \cdot \alpha + w(\textsc{final}(\sigma))$.

Proof. The proof follows by an induction over all requests of σ. Whenever 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 8. *For any input σ, with $(2+\epsilon)$-augmentation,* $\textsc{Crep}^{\text{mig}}(\sigma) \leq (1 + 4/\epsilon) \cdot \alpha \cdot \sum_{m \in M(\sigma)} \textsc{size}(m)$.

Proof. Let V be any cluster and n be the number of nodes Crep stores at this cluster. We define *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:

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

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 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.,

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

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, $\Delta\textsc{ovr}(V_x) \leq 0$, $\Delta\textsc{ovr}(V_y) \leq |c_x|$, $V_z = V_y$, and thus (2) follows.

2. V_y had less than $|c_x|$ empty slots and $|c_y| \leq (2/\epsilon) \cdot |c_x|$. Crep migrates both c_x and c_y to component V_z and the incurred cost is $\textsc{Crep}^{\text{mig}}(r) = (|c_x| + |c_y|) \cdot \alpha \leq (1 + 2/\epsilon) \cdot |c_x| \cdot \alpha < (1 + 4/\epsilon) \cdot |c_x| \cdot \alpha$. It remains to show that the second summand of (2) is at most 0. Clearly, $\Delta\textsc{ovr}(V_x) \leq 0$ and $\Delta\textsc{ovr}(V_y) \leq 0$. 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 $k + |c_x| + |c_y| \leq 2k$ after the migration. This implies that $\Delta\textsc{ovr}(V_z) = 0 - 0 = 0$.

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 $\textsc{Crep}^{\text{mig}}(r) = (|c_x| + |c_y|) \cdot \alpha < 2 \cdot |c_y| \cdot \alpha$. This time, $\textsc{Crep}^{\text{mig}}(r)$ 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, $\Delta\textsc{ovr}(V_x) \leq 0$ and $\Delta\textsc{ovr}(V_z) = 0$. Observe that $|c_x| < (\epsilon/2) \cdot |c_y| \leq (\epsilon/2) \cdot k$. As the migration of $|c_x|$ to V_y was not possible, the initial number of nodes in V_y was greater than $(2 + \epsilon) \cdot k - |c_x| \geq (2 + \epsilon/2) \cdot k$, i.e., $\textsc{ovr}^{\sigma}(V_y) \geq (\epsilon/2) \cdot k \geq (\epsilon/2) \cdot |c_y|$. As component c_y was migrated out of V_y, the number of overflow nodes in V_y changes by

   $$\Delta\textsc{ovr}(V_y) = -\min\left\{ \textsc{ovr}^{\sigma}(V_y),\, |c_y| \right\} \leq -(\epsilon/2) \cdot |c_y| \;\; .$$

   Therefore, the second summand of (2) is at most $(4/\epsilon) \cdot \alpha \cdot \Delta\textsc{ovr}(V_y) \leq -(4/\epsilon) \cdot \alpha \cdot (\epsilon/2) \cdot |c_y| = -2 \cdot |c_y| \cdot \alpha$ as desired.

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

4.5 Analysis: Competitive Ratio

In the previous two subsections, we related $\textsc{Opt}(\sigma)$ to the total size of components that are deleted by Crep (cf. Lemma 6) and $\textsc{Crep}(\sigma)$ to $\sum_{m \in M(\sigma)} \textsc{size}(m)$, where the latter amount is related to the merging actions performed by Crep (cf. Lemma 8). 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 9. *For any input σ, $\sum_{m \in M(\sigma)} \textsc{size}(m) \leq \sum_{c \in \textsc{del}(\sigma)} |c| \cdot \log k + \sum_{c \in \textsc{final}(\sigma)} |c| \cdot \log |c|$, where all logarithms are binary.*

Proof. We prove the lemma by the 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 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 $p \leq q$ without loss of generality. Due to such action, the right hand side of the lemma inequality increases by

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

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 $-\sum_{i=1}^{g}|c_i|\cdot\log|c_i| + |c|\cdot\log k \geq -\sum_{i=1}^{g}|c_i|\cdot\log k + |c|\cdot\log k = 0$. As the left hand side of the lemma inequality is unaffected by artificial merge actions, the inductive hypothesis follows also in this case. □

Theorem 10. *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 7 and Lemma 8,

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

In order to bound $|M(\sigma)|$, 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 $|c|-1 \leq |c|$, 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 $\textsc{size}(m) \geq 1$. Combining these two bounds yields $|M(\sigma)| \leq \sum_{m\in M(\sigma)}\textsc{size}(m) + \sum_{c\in\textsc{del}(\sigma)}|c|$. We use this inequality and later apply Lemma 9 to bound $\sum_{m\in M(\sigma)}\textsc{size}(m)$ obtaining,

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

Finally, Lemma 6 can be used to bound $\sum_{c\in\textsc{del}(\sigma)}|c|\cdot\alpha$ by $2k\cdot\textsc{Opt}(\sigma)$ yielding

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

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 2, $w(\textsc{final}(\sigma)) < k\cdot\ell\cdot\alpha$. 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, $\sum_{c\in\textsc{final}(\sigma)}|c|\cdot\log|c| \leq \ell\cdot((2+\epsilon)\cdot k)\cdot\log((2+\epsilon)\cdot k) = O(\ell\cdot k\cdot\log k)$.

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 is viewed as an online maximal (re)matching problem: clusters of size two host (resp. "match") exactly one pair of nodes, and maximizing pair-wise communication within each cluster is equivalent to minimizing inter-cluster communication.

5.1 Generalizing Requests Costs

Our approach will be to migrate nodes a accumulated cost pertained to certain pair reaches some threshold. It appears however that the best choice of such threshold is $(4/5) \cdot \alpha$, which might not be an integer multiple of requests costs (i.e., the treshold might not be reached in one step, but exceeded in the next one).

We deal with this problem in a standard way, by considering a *generalized cost model*, in which each communication request between node in two different clusters incurs cost $\epsilon = 1/5$. To avoid ambiguity, we will call such communication attempt *ϵ-request*. Note that a swap of two nodes still costs 2α. We define algorithm Greedy for this model, and show that it is 7-competitive. Afterwards, using the fact that the algorithm may migrate nodes before serving the communication request, we show how to construct an algorithm Greedy' for the standard cost model (where communication between two clusters costs 1), whose total cost is at most that of Greedy on any input.

5.2 Greedy Algorithm for Generalized Cost Model

A natural greedy online algorithm Greedy, parameterized by a real positive number λ, solves this problem as follows. Roughly speaking, for each pair of nodes that are in different clusters, we keep track of the total communication cost caused by ϵ-requests since the nodes are put in different clusters. Once the sum of some of these costs (defined precisely below) reaches the threshold of $\lambda \cdot \alpha$, we put these nodes in the same cluster. We will ensure that $\lambda \cdot \alpha$ is an integer multiplicity of ϵ, i.e., once some costs are greater or equal than $\lambda \cdot \alpha$, they are in fact equal $\lambda \cdot \alpha$.

More precisely, upon seeing and serving an inter-cluster ϵ-request over an edge $e = (x, y)$, Greedy adds ϵ to the counter $\mathrm{cnt}(e)$. If after counter increase there exist four nodes v_1, v_2, u_1, u_2, such that (i) pairs (v_1, v_2) and (u_1, u_2) are kept by Greedy in single clusters and (ii) $\mathrm{cnt}((v_1, u_1)) + \mathrm{cnt}((v_2, u_2)) \geq \lambda \cdot \alpha$, then Greedy performs a swap: it puts v_1 and u_1 in one cluster and v_2 and u_2 in another. Afterwards, Greedy resets counters $\mathrm{cnt}((v_1, u_1))$ and $\mathrm{cnt}((v_2, u_2))$ to zero.

5.3 Analysis

Observe that by the definition of Greedy, a counters related to a pair of nodes kept in a single cluster is always zero. For the analysis, we assume that there is an edge between each pair of nodes and the set of all edges is denoted by E. Let M^{GR} (M^{OPT}) denotes the set of all edges $e = (u, v)$, such that u and v are kept in the same cluster by Greedy (Opt).

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

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

where f is a constant that will be defined later.

Note that both M^{GR} and M^{OPT} are perfect matchings on the set of all edges. Their union constitutes a set of alternating cycles: each alternating cycle consists of $2j$ nodes, j edges from M^{GR} and j edges from M^{OPT}, interleaved, for some $j \geq 1$. The case $j = 1$ is degenerate: the cycle is then actually a single edge from $M^{\text{GR}} \cap M^{\text{OPT}}$, but we still count it as a cycle. Note that each node is adjacent to exactly one edge from M^{GR} and exactly one edge from M^{OPT} (it can be the same edge). We define the cycle-potential function as

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

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

We will show that for appropriate choice of λ, f and g, for any of three events that may occur at a single time: serving ϵ-request by Greedy and Opt (Lemma 11), a swap performed by Greedy (Lemma 13) and a swap performed by Opt (Lemma 14), the following inequality hold:

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

ΔGreedy and ΔOpt are increases of Greedy and Opt cost, respectively. $\Delta\Psi$ and $\Delta\Phi(e)$ are the changes of the values of Ψ and $\Phi(e)$. The 7-competitiveness (Theorem 15) will then immediately follow by summing (3) and bounding the initial and final values of potentials.

Lemma 11. *If $f \leq 6$, then (3) holds for the event of serving an ϵ-request by* Greedy *and* Opt.

Proof. Let $e_c = (v, u)$ be the communicating pair. For the considered event, neither Greedy nor Opt change their clusters. Therefore the structure of alternating cycles remains unchanged and $\Delta\Psi = 0$. Futhermore, the counter value may change only for edge e_c and therefore, among all edges, only the edge-potential of e_c may change. We consider three cases.

1. If $e_c \in M^{\text{GR}}$, then $\Delta\text{Greedy} = 0$, $\Delta\text{Opt} \geq 0$. As $\text{cnt}(e_c)$ is unchanged, $\Delta\Phi(e_c) = 0$.
2. If $e_c \in M^{\text{OPT}} \setminus M^{\text{GR}}$, $\Delta\text{Greedy} = \epsilon$ and $\Delta\text{Opt} = 0$. The value of $\text{cnt}(e_c)$ increases by ϵ, and hence $\Delta\Phi(e_c) = -\epsilon$.
3. If $e_c \notin M^{\text{OPT}}$ and $e_c \notin M^{\text{GR}}$, then $\Delta\text{Greedy} = \Delta\text{Opt} = \epsilon$. The value of $\text{cnt}(e_c)$ increases by ϵ, and hence $\Delta\Phi(e_c) = f \cdot \epsilon$.

Therefore, for the considered event $\Delta\text{Greedy} + \Delta\Psi + \sum_{e \in E} \Delta\Phi(e) = \Delta\text{Greedy} + \Delta\Phi(e_c) \leq (f + 1) \cdot \Delta\text{Opt}$, which implies (3) as we assumed $f \leq 6$. □

Before we proceed with two remaining lemmas, we state the following simple observation.
TODO: this observation is false... it is possible that it does not change

Observation 12. *Consider a swap (performed either by* Opt *or* Greedy*). If the two swapped edges belonged to a single cycle, then the number of cycles increases by one, otherwise it decreases by one.*

Lemma 13. *If $2 + \lambda \leq g \leq f \cdot \lambda - 2$, then (3) holds for the event of* Greedy *performing a swap.*

Proof. Clearly, for such event $\Delta\text{Greedy} = 2\alpha$ and $\Delta\text{Opt} = 0$. There are four edges involved in a swap: let e_1^{old} and e_2^{old} be the edges that were in M^{GR} before the swap and let e_1^{new} and e_2^{new} be the new edges in M^{GR}after the swap. By the definition of Greedy, $\text{cnt}(e_1^{\text{old}}) = \text{cnt}(e_2^{\text{old}}) = 0$ before and after the swap.

By the bounds above, $\Delta\text{Greedy} + \Delta\Phi + \Delta\Psi = 2\alpha + \Delta\Phi(e_1^{\text{new}}) + \Delta\Phi(e_2^{\text{new}}) + \Delta\Psi$, and hence to show (3) it suffices to show that the latter amount is at most $7 \cdot \Delta\text{Opt} = 0$. Recall that, by the definition of Greedy, $\text{cnt}(e_1^{\text{new}}) + \text{cnt}(e_2^{\text{new}}) = \lambda \cdot \alpha$ before the swap, and after the swap these counters are reset to zero. We use Observation 12, to split the analysis of potential change into two cases.

1. In the first case, edges e_1^{old} and e_2^{old} were in different clusters before the swap. Then, by Observation 12, the number of cycles decrements, and hence $\Delta\Psi = g \cdot \alpha$. Furthermore, e_1^{new} and e_2^{new} do not belong to M^{OPT} (otherwise e_1^{old} and e_2^{old} would lie on a common alternating cycle). Therefore, $\Delta\Phi(e_1^{\text{new}}) + \Delta\Phi(e_2^{\text{new}}) = 0 + 0 - f \cdot \text{cnt}(e_1^{\text{old}}) - f \cdot \text{cnt}(e_2^{\text{old}}) = -f \cdot \lambda \cdot \alpha$ and the claim folows as $2 + g - f \cdot \lambda \leq 0$ by the lemma assumption.

2. In the second case, edges e_1^{old} and e_2^{old} were in the same cluster before the swap. Then, by Observation 12, the number of cycles increments, and hence $\Delta\Psi = -g \cdot \alpha$. The edge-potential of any edge e_1^{new} after the swap is zero and at least $-\text{cnt}(e_1^{\text{new}})$ before the swap. Therefore, $\Delta\Phi(e_1^{\text{new}}) + \Delta\Phi(e_2^{\text{new}}) = \text{cnt}(e_1^{\text{new}}) + \text{cnt}(e_2^{\text{new}}) = \lambda \cdot \alpha$ and the claim follows as $2 + \lambda - g \leq 0$ by the lemma assumption. □

Lemma 14. *If* $2 \cdot (f+1) \cdot \lambda + g \leq 14$, *then* (3) *holds for the event of* Opt *performing a swap.*

Proof. Clearly, for such event $\Delta\textsc{Greedy} = 0$ and $\Delta\textsc{Opt} = 2\alpha$. By Observation 12, the number of cycles decreases at most by one, and thus $\Delta\Psi \leq g \cdot \alpha$.

Now, we will upper-bound the change in the edge-potentials. This time, 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, $\Delta\Phi(e_i^{\text{new}}) \leq 0$ as the initial value of $\Phi(e_i^{\text{new}})$ is at least 0 and the final value of $\Phi(e_i^{\text{new}})$ is at most 0. Similarly, for any i, $\Delta\Phi(e_i^{\text{old}}) \leq (f+1) \cdot \text{cnt}(e_i^{\text{old}})$ as the initial value of $\Phi(e_i^{\text{old}})$ is at least $-\text{cnt}(e_i^{\text{old}})$ and the final value of $\Phi(e_i^{\text{old}})$ is at most $f \cdot \text{cnt}(e_i^{\text{old}})$. Summing, up we obtain that $\Delta\Phi \leq (f+1) \cdot \text{cnt}(e_i^{\text{old}}) \leq 2 \cdot (f+1) \cdot \lambda \cdot \alpha$, as the counter of each edge is at most $\lambda \cdot \alpha$.

Finally, by putting the achieved bounds together, and using the lemma assumption, we obtain $\Delta\textsc{Greedy} + \Delta\Phi + \Delta\Psi \leq 0 + 2(f+1) \cdot \lambda \cdot \alpha + g \cdot \alpha \leq 14\alpha \leq 7 \cdot \Delta\textsc{Opt}$. □

Theorem 15. *Assume the generalized cost model with ϵ-requests, where $\epsilon = 1/5$. For $\lambda = 4/5$, $f = 6$ and $g = 14/5$,* Greedy *is 7-competitive.*

Proof. For the chosen value of λ, and integer α, $\lambda \cdot \alpha$ is a multiplicity of ϵ and therefore Greedy is well defined. Furthermore, the chosen values of λ, f and g satisfy the conditions of Lemmas 11, 13 and 14. Summing these inequalities over all events occuring in the runtime on an input sequence σ (serving requests and swaps of Greedy and Opt) yields

$$\textsc{Greedy}(\sigma) + (\Psi_{\text{final}} - \Psi_{\text{initial}}) + \textstyle\sum_{e \in E} (\Phi_{\text{final}}(e) - \Phi_{\text{initial}}(e)) \leq 7 \cdot \textsc{Opt}(\sigma) \ , \tag{4}$$

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, $\Psi_{\text{final}} - \Psi_{\text{initial}} \leq g \cdot \ell \cdot \alpha$ (the number of alternating cycles is at most ℓ) and $\Phi_{\text{final}}(e) - \Phi_{\text{initial}}(e) \leq (f+1) \cdot \text{cnt}(e) \leq (f+1) \cdot \lambda \cdot \alpha$ (by the definition of Greedy, all edge counters are always at most $\lambda \cdot \alpha$). The number of edges is exactly $\binom{2\cdot\ell}{2}$, and therefore

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

which concludes the proof. □

5.4 Back to the standard cost model

TODO: finish here

To this end, upon seeing a request (of cost 1) ALG′ feeds $1/\epsilon$ requests of cost ϵ to ALG. If ALG decides to perform migration, then costs $\epsilon = 1/5$, and the thresholds are integer multiplicities of ϵ. Now, given an actual request (in the standard model), we give an algorithm $1/\epsilon$ requests, each of cost ϵ and translate its actions into actions in the standard model as follows: if the algorithm decides to swap after any of

Recall however that our model allows to migrate nodes before serving the request. Therefore, we use a standard trick of splitting each request into $1/\epsilon$ virtual "microrequests", each incurring an infinitisimal cost ϵ. Then if the algorithm decides to migrate

It is convenient to define our algorithm so that it performs node migration after some

Our algorithm will make a migration decision after

6 Lower Bounds

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 would be required to solve the partitioning problem trivially, by putting all nodes into the same cluster). The proof uses average argument. A similar argument can be used for a special case online rematching ($k = 2$ without agumentation), where we present a lower bound of 3.

6.1 Reduction to Online Paging

Theorem 16. *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 where the cache size is $k - 1$ and the number of different pages is k.*

Proof. The pages are denoted $p_1, p_2, \ldots, p_k$ and 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 $\sigma = \sigma_1, \sigma_2, \sigma_3, \ldots$ for the paging problem, where σ_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. Upon seeing a request to a page σ_t, it creates a subsequence of requests for the BRP problem, runs B on them, and on this basis serves the request σ_t.

We use the following 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 configuration of an algorithm for the BRP problem 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: 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, by 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 $p_1, p_2, \ldots, p_{k-1}$ for the paging problem).

For any paging node p, let $\textsc{comm}(p)$ be a subsequence of communication requests in the BRP instance, consisting ot the request (p, s) followed by $\binom{k-1}{2}$ requests to all pairs of auxiliary nodes. Given an input sequence σ, we construct the input sequence σ^B for the BRP problem in the following way: For a request σ_t, we repeat a subsequence $\textsc{comm}(\sigma_t)$ till the σ_t becomes collocated with s and the resulting node placement is well aligned. As there exists a fixed node placement for which serving subsequence $\textsc{comm}(\sigma_t)$ is free, the algorithm B ends up in such configuration at some point (as otherwise its cost would be arbitrarily large and B would not be competitive). We denote the resulting sequence of $\textsc{comm}(\sigma_t)$ subsequences by $\textsc{comm}_t(\sigma_t)$.

By $cnf(t)$ we denote the configuration of $\textsc{Det}$ at the end of the last $\textsc{comm}(p^t)$ subsequence. In addition, by $cnf(0)$ we denote the initial configuration and we note that it is well aligned.

To construct the response of the algorithm P to the paging request σ_t, it runs P on $\textsc{comm}_t(\sigma_t)$ and read its configuration $cnf(t)$. As $cnf(t)$ is well aligned, $\textsc{cache}(cnf(t))$ is well defined. As a response, we change the cache configuration of P to a cache corresponding to configuration $cnf(t)$. Observe that the solution returned by P is feasible: σ_t is collocated with s in $cnf(t)$, and thus included by the cache. Furthermore, we may relate the cost of P to the cost of B. Assume that P needs to modify the cache contents. The cost of such change is at most 1 as exactly one page has be be fetched. On the other hand, such situation occurs only if B changed some node placement in clusters, resulting in the cost of at least $2 \cdot \alpha$. Therefore, $2 \cdot \alpha \cdot P(\sigma_t) \leq B(\textsc{comm}_t(\sigma_t))$, which summed over all requests from sequence σ^P yields the relation $2 \cdot \alpha \cdot P(\sigma) \leq \textsc{Det}^P(\sigma^B)$.

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

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

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

and therefore $\textsc{Det}^P$ is γ-competitive. □

As any deterministic algorithm for the paging problem with cache size $k-1$ has competitive ratio at least $k-1$ [ST85], we obtain the following result.

Corollary 17. *The competitive ratio of the BRP problem on two clusters is at least $k-1$.*

6.2 Remaining Lower Bounds

Theorem 18. *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 will be numbered as follows: $v_0, v_1, \ldots, v_{n-1}$. All the requests presented in our construction will be edges in a ring graph on these nodes, where the edge are defined as $e_i = \{v_i, v_{(i+1) \mod n}\}$ for $i = 0, \ldots, n-1$. At all integer times, the adversary gives a request

between an arbitrary pair of nodes that are kept by ONL in different clusters. As $\delta < \ell$, ONL cannot fit the entire ring in a single cluster, and hence such pair always exists.

This way, we may define an input sequence σ of an arbitrary length. Note ONL has to serve every request remotely, and hence it pays 1 for each request. In this proof, we will ignore migration costs paid by ONL.

TODO: time 0 not defined here

Now we present k offline algorithms $\text{OFF}_1, \text{OFF}_2, \ldots, \text{OFF}_k$, such that, neglecting an initial node reorganization which they perform at time 0, the sum of their total costs on σ is exactly $\text{ONL}(\sigma)$. To this end, for any $j \in \{0, \ldots, k-1\}$, we define a *cut-edge* set $\text{cut}(j) = \{e_j, e_{j+k}, e_{j+2k}, \ldots, e_{j+(\ell-1)\cdot k}\}$. Note that the cut-edge set $\text{cut}(j)$ defines a natural partitioning of ring nodes (and hence also all nodes) into clusters, each containing k nodes. At time 0, the algorithm OFF_j first migrates its nodes (for a total cost of at most $n \cdot \alpha$) to maintain clustering defined by $\text{cut}(j)$ and then does not change the node placement throughout the whole σ.

As all cut-edge sets are pairwise disjoint, at any request time t, exactly one algorithm OFF_j pays for the request, and thus $\sum_{j=1}^{k} \text{OFF}_j(t) = \text{ONL}(t)$. Therefore, taking the initial node reorganization into account, we obtain that $\sum_{j=1}^{k} \text{OFF}_j(\sigma) \leq k \cdot n \cdot \alpha + \text{ONL}(\sigma)$. By the average argument, there exists offline algorithm OFF_j, such that $\text{OFF}_j(\sigma) \leq \frac{1}{k} \cdot \sum_{j=1}^{k} \text{OFF}_j(\sigma) \leq n \cdot \alpha + \text{ONL}(\sigma)/k$ and therefore $\text{ONL}(\sigma) \geq k \cdot \text{OFF}_j(\sigma) - k \cdot n \cdot \alpha \geq k \cdot \text{OPT}(\sigma) - k \cdot n \cdot \alpha$. The theorem follows because the additive constant $k \cdot n \cdot \alpha$ becomes negligible as the length of σ grows. □

Theorem 19. *No deterministic online algorithm* ONL *can achieve a competitive ratio smaller than* 3 *for the case* $k = 2$.

Proof. We distinguish three assignments of nodes to clusters: configuration A: v_0 collocated with v_1 and v_2 collocated with v_3, configuration B: v_1 collocated with v_2 and v_3 collocated with v_0. configuration C: other placements.

Similarly to the proof of Theorem 18, the adversary always asks for an edge whose two endpoints are kept by ONL in different clusters. This time the exact choice of such edge 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 now 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 (they will never change the position of the remaining nodes). More concretely, OFF_1 keeps its nodes $\{v_0, \ldots, v_3\}$ always in configuration B and OFF_2 always in configuration A (note that this is exactly what OFF_1 and OFF_2 did in the previous proof). Furthermore, we define the third algorithm OFF_3 that is always in configuration A if ONL is in configuration B or C and is in configuration B if ONL is in configuration A.

Now we split the cost of ONL into the cost for serving requests ONL^{req} and the cost paid for its migrations, ONL^{mig}. By the same argument as in the previous proof, $\text{OFF}_1(t) + \text{OFF}_2(t) = \text{ONL}^{\text{req}}(t)$ for any $t \geq 1$. As OFF_3 does not pay for any request and migrates only if ONL migrates, $\text{OFF}_3(t) \geq \text{ONL}^{\text{mig}}(t)$ for any $t \geq 1$. Summing up, $\sum_{j=1}^{3} \text{OFF}_j(t) = \text{ONL}(t)$ for any $t \geq 1$. Taking into account the initial reconfiguration of nodes in OFF_j solutions (which incurs a cost of at most 4α), we obtain that $\sum_{j=1}^{3} \text{OFF}(\sigma) \leq 4\alpha + \text{ONL}(\sigma)$ and hence by the average argument, there exists $j \in \{1, 2, 3\}$, such that $\text{ONL}(\sigma) \geq 3 \cdot \text{OFF}_j(\sigma) - 4\alpha \geq 3 \cdot \text{OPT}(\sigma) + 4\alpha$. This concludes the proof as 4α 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

[AAB04] Baruch Awerbuch, Yossi Azar, and Yair Bartal. On-line generalized Steiner problem. *Theoretical Computer Science*, 324(2–3):313–324, 2004.

[ABF93] Baruch Awerbuch, Yair Bartal, and Amos Fiat. Competitive distributed file allocation. In *Proc. 25th ACM Symp. on Theory of Computing (STOC)*, pages 164–173, 1993.

[ABF98] Baruch Awerbuch, Yair Bartal, and Amos Fiat. Distributed paging for general networks. *Journal of Algorithms*, 28(1):67–104, 1998.

[ACN00] Dimitris Achlioptas, Marek Chrobak, and John Noga. Competitive analysis of randomized paging algorithms. *Theoretical Computer Science*, 234(1–2):203–218, 2000.

[AKK99] Sanjeev Arora, David R. Karger, and Marek Karpinski. Polynomial time approximation schemes for dense instances of NP-hard problems. *Journal of Computer and System Sciences*, 58(1):193–210, 1999.

[AR06] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. *Theory of Computing Systems*, 39(6):929–939, 2006.

[ARV09] Sanjeev Arora, Satish Rao, and Umesh Vazirani. Expander flows, geometric embeddings and graph partitioning. *Journal of the ACM*, 56(2):5, 2009.

[BBM17] Marcin Bienkowski, Jaroslaw Byrka, and Marcin Mucha. Dynamic beats fixed: On phase-based algorithms for file migration. In *Proc. 44th Int. Colloq. on Automata, Languages and Programming (ICALP)*, pages 13:1–13:14, 2017.

[BCI01] Yair Bartal, Moses Charikar, and Piotr Indyk. On page migration and other relaxed task systems. *Theoretical Computer Science*, 268(1):43–66, 2001.

[BFR95] Yair Bartal, Amos Fiat, and Yuval Rabani. Competitive algorithms for distributed data management. *Journal of Computer and System Sciences*, 51(3):341–358, 1995.

[CZM+11] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. Managing data transfers in computer clusters with Orchestra. In *ACM SIGCOMM*, pages 98–109, 2011.

[EILN15] Leah Epstein, Csanád Imreh, Asaf Levin, and Judit Nagy-György. Online file caching with rejection penalties. *Algorithmica*, 71(2):279–306, 2015.

[EILNG11] Leah Epstein, Csanád Imreh, Asaf Levin, and Judit Nagy-György. On variants of file caching. In *Proc. 38th Int. Colloq. on Automata, Languages and Programming (ICALP)*, pages 195–206, 2011.

[ENRS99] Guy Even, Joseph Naor, Satish Rao, and Baruch Schieber. Fast approximate graph partitioning algorithms. *SIAM Journal on Computing*, 28(6):2187–2214, 1999.

[ENRS00] Guy Even, Joseph Naor, Satish Rao, and Baruch Schieber. Divide-and-conquer approximation algorithms via spreading metrics. *Journal of the ACM*, 47(4):585–616, 2000.

[FF15] Andreas Emil Feldmann and Luca Foschini. Balanced partitions of trees and applications. *Algorithmica*, 71(2):354–376, 2015.

[FK02] Uriel Feige and Robert Krauthgamer. A polylogarithmic approximation of the minimum bisection. *SIAM Journal on Computing*, 31(4):1090–1118, 2002.

[FKL+91] Amos Fiat, Richard M. Karp, Michael Luby, Lyle A. McGeoch, Daniel D. Sleator, and Neal E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, 1991.

[FKN00] Uriel Feige, Robert Krauthgamer, and Kobbi Nissim. Approximating the minimum bisection size (extended abstract). In *Proc. 32nd ACM Symp. on Theory of Computing (STOC)*, pages 530–536, 2000.

[FWL16] Mário César San Felice, David P. Williamson, and Orlando Lee. A randomized o(log n)-competitive algorithm for the online connected facility location problem. *Algorithmica*, 76(4):1139–1157, 2016.

[GJS76] M. R. Garey, David S. Johnson, and Larry J. Stockmeyer. Some simplified np-complete graph problems. *tcs*, 1(3):237–267, 1976.

[Ira02] Sandy Irani. Page replacement with multi-size pages and applications to web caching. *Algorithmica*, 33(3):384–409, 2002.

[KF06] Robert Krauthgamer and Uriel Feige. A polylogarithmic approximation of the minimum bisection. *SIAM Review*, 48(1):99–130, 2006.

[KMMO94] Anna R. Karlin, Mark S. Manasse, Lyle A. McGeoch, and Susan Owicki. Competitive randomized algorithms for non-uniform problems. *Algorithmica*, 11(6):542–571, 1994.

[KNS09] Robert Krauthgamer, Joseph Naor, and Roy Schwartz. Partitioning graphs into balanced components. In *Proc. 20th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 942–949, 2009.

[LMT90] T. Leighton, F. Makedon, and S. G. Tragoudas. Approximation algorithms for vlsi partition problems. In *IEEE International Symposium on Circuits and Systems*, volume 4, pages 2865–2868, 1990.

[LPSR08] Zvi Lotker, Boaz Patt-Shamir, and Dror Rawitz. Rent, lease or buy: Randomized algorithms for multislope ski rental. In *Proc. 25th Symp. on Theoretical Aspects of Computer Science (STACS)*, pages 503–514, 2008.

[MP12] Jeffrey C Mogul and Lucian Popa. What we talk about when we talk about cloud network performance. *ACM SIGCOMM Computer Communication Review*, 42(5):44–48, 2012.

[MS91] Lyle A. McGeoch and Daniel D. Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6(6):816–825, 1991.

[Räc08] Harald Räcke. Optimal hierarchica decompositions for congestion minimization in networks. In *Proc. 40th ACM Symp. on Theory of Computing (STOC)*, pages 255–264, 2008.

[ST85] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[ST97] Horst D. Simon and Shang-Hua Teng. How good is recursive bisection? *SIAM Journal on Computing*, 18(5):1436–1445, 1997.

[SV95] Huzur Saran and Vijay V. Vazirani. Finding k cuts within twice the optimal. *SIAM Journal on Computing*, 24(1):101–108, 1995.

[Umb15] Seeun Umboh. Online network design algorithms via hierarchical decompositions. In *Proc. 26th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 1373–1387, 2015.