Tight Bounds for Online Graph Partitioning[*]

Monika Henzinger [†] Stefan Neumann[‡] Harald Räcke[§] Stefan Schmid[¶]

Abstract

We consider the following online optimization problem. We are given a graph G and each vertex of the graph is assigned to one of ℓ servers, where servers have capacity k and we assume that the graph has $\ell \cdot k$ vertices. Initially, G does not contain any edges and then the edges of G are revealed one-by-one. The goal is to design an online algorithm ONL, which always places the connected components induced by the revealed edges on the same server and never exceeds the server capacities by more than ϵk for constant $\epsilon > 0$. Whenever ONL learns about a new edge, the algorithm is allowed to move vertices from one server to another. Its objective is to minimize the number of vertex moves. More specifically, ONL should minimize the competitive ratio: the total cost ONL incurs compared to an optimal offline algorithm OPT.

The problem was recently introduced by Henzinger et al. (SIGMETRICS'2019) and is related to classic online problems such as online paging and scheduling. It finds applications in the context of resource allocation in the cloud and for optimizing distributed data structures such as union–find data structures.

Our main contribution is a polynomial-time randomized algorithm, that is asymptotically optimal: we derive an upper bound of $O(\log \ell + \log k)$ on its competitive ratio and show that no randomized online algorithm can achieve a competitive ratio of less than $\Omega(\log \ell + \log k)$. We also settle the open problem of the achievable competitive ratio by deterministic online algorithms, by deriving a competitive ratio of $\Theta(\ell \log k)$; to this end, we present an improved lower bound as well as a deterministic polynomial-time online algorithm.

Our algorithms rely on a novel technique which combines efficient integer programming with a combinatorial approach for maintaining ILP solutions. More precisely, we use an ILP to assign the connected components induced by the revealed edges to the servers; this is similar to existing approximation schemes for scheduling algorithms. However, we cannot obtain our competitive ratios if we run the ILP after each edge insertion. Instead, we identify certain types of edge insertions, after which we can manually obtain an optimal ILP solution at zero cost without resolving the ILP. We believe this technique is of independent interest and will find further applications in the future.

[*]The full version of this paper including all proofs is available on arxiv.

[†]University of Vienna, Austria. Monika Henzinger is supported by the Austrian Science Fund (FWF): 33775-N.

[‡]KTH Royal Institute of Technology, Sweden. Work was done while at University of Vienna. Stefan Neumann gratefully acknowledges the financial support from the Doctoral Programme "Vienna Graduate School on Computational Optimization" which is funded by the Austrian Science Fund (FWF, project no. W1260-N35) and from the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement No. 340506.

[§]TU Munich, Germany.

[¶]University of Vienna, Austria. Stefan Schmid is supported by the European Research Council under the European Community's Horizon 2020 Programme / ERC grant agreement No. 864228 and by the Austrian Science Fund (FWF): 33775-N.

1 Introduction

Distributed cloud applications generate a significant amount of network traffic [22]. To improve their efficiency and performance, the underlying infrastructure needs to become *demand-aware*: frequently communicating endpoints need to be allocated closer to each other, e.g., by collocating them on the same server or in the same rack. Such optimizations are enabled by the increasing resource allocation flexibilities available in modern virtualized infrastructures, and are further motivated by rich spatial and temporal structure featured by communication patterns of data-intensive applications [9].

This paper studies the algorithmic problem underlying such demand-aware resource allocation in scenarios where the communication pattern is not known ahead of time. Instead, the algorithm needs to learn a communication pattern in an online manner, dynamically collocating communication partners while minimizing reconfiguration costs. It has recently been shown that this problem can be modeled by the following *online graph partitioning problem* [18]: Let k and ℓ be known parameters. We are given a graph G which initially does not contain any edges. Each vertex of the graph is assigned to one of ℓ servers and each server has *capacity* k, i.e., stores k vertices. Next, the edges of G are revealed one-by-one in an online fashion and the algorithm has to guarantee that *every connected component is placed on the same server (cc-condition).* This is possible, as it is guaranteed that there always exists an assignment of the connected components to servers such that no connected component is split across multiple servers. Thus after each edge insertion, the online algorithm has to decide which vertices to move between servers to guarantee the cc-condition. Each vertex move incurs a cost of $1/k$. The optimal offline algorithm knows all the connected components, computes, dependent on the initial placement of the vertices, the minimum-cost assignment of these connected components to servers and moves the vertices to their

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

final servers after the first edge insertion. It requires no further vertex moves. To measure the performance of an online algorithm ONL we use the competitive ratio: the total cost of ONL divided by the total cost of the optimal offline algorithm OPT. In this setting, no deterministic online algorithm can have a competitive ratio better than $\Omega(\ell \cdot k)$ even with unbounded computational resources [18, 23]. In fact, even assigning the connected components to the servers such that the capacity constraints are obeyed is already NP-hard and this holds even in the static setting when the connected components do not change [4].

Thus, we relax the server capacity requirement for the online algorithm: Specifically, the online algorithm is allowed to place up to $(1 + \epsilon)k$ vertices on a server at any point in time. We call this problem the *online graph partitioning problem.*

Henzinger et al. [18] studied this problem and showed that the previously described demand-aware resource allocation problem reduces to the online graph partitioning problem: vertices of the graph G correspond to communication partners and edges correspond to communication requests; thus, by collocating the communication partners based on the connected components of the vertices, we minimize the network traffic (since all future communications among the revealed edges will happen locally). They also showed how to implement a distributed union–find data structure with this approach. Algorithmically, [18] presented a deterministic exponential-time algorithm with competitive ratio $O(\ell \log \ell \log k)$ and complemented their result with a lower bound of $\Omega(\log k)$ on the competitive ratio of any deterministic online algorithm. While their derived bounds are tight for $\ell = O(1)$ servers, there remains a gap of factor $O(\ell \log \ell)$ between upper and lower bound for the scenario of $\ell = \omega(1)$. Furthermore, their lower bound only applies to deterministic algorithms and thus it is a natural question to ask whether randomized algorithms can obtain better competitive ratios.

1.1 Our Contributions

Our main contribution is a polynomial-time randomized algorithm for online graph partitioning which achieves a polylogarithmic competitive ratio. In particular, we derive an $O(\log \ell + \log k)$ upper bound on the competitive ratio of our algorithm, where ℓ is the number of servers and k is the server capacity. We also show that no randomized online algorithm can achieve a competitive ratio of less than $\Omega(\log \ell + \log k)$. The achieved competitive ratio is hence asymptotically optimal.

We further settle the open problem of the competitive ratio achievable by deterministic online algorithms. To this end, we derive an improved lower bound of $\Omega(\ell \log k)$, and present a polynomial-time deterministic online algorithm which achieves a competitive ratio of $O(\ell \log k)$. Thus, also our deterministic algorithm is optimal up to constant factors in the competitive ratio.

These results improve upon the results of [18] in three respects: First, our deterministic online algorithm has competitive ratio $O(\ell \log k)$ and polynomial runtime, while the algorithm in [18] has competitive ratio $O(\ell \log \ell \log k)$ and requires exponential time. Second, we present a significantly higher and matching lower bound of $\Omega(\ell \log k)$ on the competitive ratio of any deterministic online algorithm. Third, we initiate the study of randomized algorithms for the online graph partitioning problem and show that it is possible to achieve a competitive ratio of $O(\log \ell + \log k)$ and we complement this result with a matching lower bound. Note that the competitive ratio obtained by our randomized algorithm provides an exponential improvement over what any deterministic algorithm can achieve in terms of the dependency on the parameter ℓ.

We further show that for $\epsilon > 1$ (i.e., the servers can store at least $(2 + \epsilon')k$ vertices for some $\epsilon' > 0$), our deterministic algorithm is $O(\log k)$-competitive.

Technical Novelty. We will now provide a brief overview of our approach and its technical novelty. Since our deterministic and our randomized algorithms are based on the same algorithmic framework, we will say *our algorithm* in the following.

Our algorithm keeps track of the set of connected components induced by the revealed edges. We will denote the connected components as *pieces* and when two connected components become connected due to an edge insertion, we say that the corresponding pieces are *merged.*

The algorithm maintains an assignment of the pieces onto the servers which we call a *schedule.* We will make sure that the schedule is *valid*, i.e., that every piece is assigned to some server and that the capacities of the servers are never exceeded by more than the allowed additive ϵk. To compute valid schedules, we solve an integer linear program (ILP) using a generic ILP solver and show how the solution of the ILP can be transformed into a valid schedule. We ensure that the ILP is of constant size and can, hence, be solved in polynomial time. Next, we show that when two pieces are merged due to an edge insertion, the schedule does not change much, i.e., we do not have to move "too many" pieces between the servers. We do this using a *sensitivity analysis* of the ILP, which guarantees that when two pieces are merged, the solution of the ILP does not change by much. Furthermore, we prove that this change in the ILP solution corresponds to only slightly adjusting the schedules, and thus only moving a few pieces.

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

However, the sensitivity analysis alone is not enough to obtain the desired competitive ratio. Indeed, we identify certain types of merge-operations for which the optimal offline algorithm OPT might have very small or even zero cost. In this case, adjusting the schedules based on the ILP sensitivity would be too costly: the generic ILP solver from above could potentially move to an optimal ILP solution which is very different from the current solution and, thus, incur much more cost than OPT. Hence, to keep the cost paid by our algorithm low, we make sure that for these special types of merge-operations, our algorithm sticks extremely close to the previous ILP solution, incurs zero cost for moving pieces and still obtains an optimal ILP solution. The optimality of the algorithm's solution after such merge-operations is crucial as otherwise, we could not apply the sensitivity analysis after the subsequent merge-operations. To the best of our knowledge, our algorithm is the first to *interleave ILP sensitivity analysis with manual maintenance of optimal ILP solutions.*

More specifically, we assume that each server has a unique color and consider each vertex as being colored by the color of its initial server. In our analysis we identify two different types of pieces: *monochromatic* and *non-monochromatic* ones. In the monochromatic pieces, "almost all" of the vertices have the same color, i.e., were initially assigned to the same server, while the non-monochromatic pieces contain "many" vertices which started on different servers. We show that we have to treat the monochromatic pieces very carefully because these are the pieces for which OPT might have very small or even no cost. Hence, it is desirable to always schedule monochromatic pieces on the server of the majority color. Unfortunately, we show that this is not always possible. Indeed, the hard instances in our lower bounds show that an adversary can force any deterministic algorithm to create schedules with *extraordinary* servers. Informally, a server s is extraordinary if there exists a monochromatic piece p for which almost all of its vertices have the color of s but p is not scheduled on s (see Section 4 for the formal definition). All other servers are *ordinary* servers. Note that we have to deal with extraordinary servers carefully: we might have to pay much more than OPT for their monochromatic pieces that are scheduled on other servers.

Thus, to obtain a competitive algorithm, we need to minimize the number of extraordinary servers. We achieve this with the following idea: We equip our ILP with an objective function that minimizes the number of extraordinary servers and we show that the number of extraordinary servers created by the ILP gives a lower bound on the cost paid by OPT. We use this fact to argue that we can charge the algorithm's cost when creating extraordinary servers to OPT to obtain competitive results.

The previously described ideas provide a deterministic algorithm with competitive ratio $O(\ell \log k)$. We also provide a matching lower bound of $\Omega(\ell \log k)$. The lower bound provides a hard instance which essentially shows that an adversary can force any deterministic algorithm to make each of the ℓ servers extraordinary at some point in time. More generally, the adversary can cause such a high competitive ratio whenever it knows *which servers are extraordinary.* Hence, to obtain an algorithm with competitive ratio $O(\log \ell + \log k)$ we use randomization to keep the adversary from knowing the extraordinary servers.

Our strategy for picking the extraordinary servers randomly is as follows. First, we show that our algorithm experiences low cost (compared to OPT) when two pieces assigned to an ordinary server are merged, while the cost for merging pieces that are assigned to extraordinary servers is large (compared to OPT). Next, we reduce the problem of picking extraordinary servers to a paging problem, where the pages correspond to servers such that pages in the cache correspond to ordinary servers and pages outside the cache correspond to extraordinary servers. Now when two pieces are merged, we issue the corresponding page requests: A merge of pieces assigned to an ordinary server corresponds to a page request of a page which is inside the cache, while merging two pieces with at least one assigned to an extraordinary server corresponds to a page request of a page which is not stored in the cache. This leads the paging algorithm to insert and evict pages into and from the cache and our algorithm always changes the types of the servers accordingly. For example, when a page corresponding to an ordinary server is evicted from the cache, we make the corresponding server extraordinary. We conclude by showing that since the randomized paging algorithm of Blum et al. [10] allows for a polylogarithmic competitive ratio, we also obtain a polylogarithmic competitive ratio for our problem.

Future Directions. While the algorithms presented in this paper are asymptotically optimal (i.e., our upper and lower bounds on the competitive ratios match), our work opens interesting avenues for future research. A first intriguing question for future research regards the study of scenarios without the cc-condition. In this setting, Avin et al. [8, 5] provided a deterministic $O(k \log k)$-competitive algorithm, CREP, for servers with capacity $(2 + \epsilon)k$ and complemented this result with a $\Omega(k)$ lower bound for deterministic algorithms; while CREP had a super-polynomial runtime, Forner et al. [17] recently demonstrated a polynomial-time im-

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

plementation, pCrep, which monitors the connectivity of communication requests over time, rather than the density as in Crep, and this enables a faster runtime. However, nothing is known for randomized algorithms and it would be exciting to determine whether polylogarithmic competitive ratios are achievable. Second, the competitive ratios of our algorithms depend exponentially on $1/\epsilon$ (see the discussions after Theorems 5.1 and 6.1). It would be interesting to obtain tight competitive ratios with only a polynomial dependency on ϵ, as this might yield more practical algorithms. To realize this goal, it seems that one first has to come up with a PTAS for scheduling on parallel identical machines in the *offline* setting, where the dynamic program (DP) or the ILP formulation has only $\text{poly}(1/\epsilon)$ DP cells or variables, respectively. In particular, this rules out using the technique by Hochbaum and Shmoys [19], that our algorithm relies on.

1.2 Related Work

The online graph partitioning problem considered in this paper is generally related to classic online problems such as competitive paging and caching [15, 21, 25, 29, 2, 12], k-server [16], or metrical task systems [11]. However, unlike these existing problems where requests are typically related to specific items (e.g., in paging) or locations in a graph or metric space (e.g., the k-server problem and metrical task systems), in our model, requests are related to *pairs of vertices.* The problem can hence also be seen as a *symmetric* version of online paging, where each of the two items (i.e., vertices in our model) involved in a request can be moved to either of the servers currently hosting one of the items (or even to a third server). The offline problem variant is essentially a k-way partitioning or graph partitioning problem [27, 1]. The balanced graph partitioning problem is related to minimum bisection [13], and known to be hard to approximate [4, 20]. Balanced clustering problems have also been studied in streaming settings [26, 3].

Our model is also related to dynamic bin packing problems which allow for limited *repacking* [14]: this model can be seen as a variant of our problem where pieces (resp. items) can both be dynamically inserted and deleted, and it is also possible to open new servers (i.e., bins); the goal is to use only an (almost) minimal number of servers, and to minimize the number of piece (resp. item) moves. However, the techniques of [14] do not extend to our problem.

Another related problem arises in the context of generalized online scheduling, where the current server assignment can be changed whenever a new job arrives, subject to the constraint that the total size of moved jobs is bounded by some constant times the size of the arriving job. While the reconfiguration cost in this model is fairly different from ours, the sensitivity analysis of our ILP is inspired by the techniques used in Hochbaum and Shmoys [19] and Sanders et al. [24].

Our work is specifically motivated by the online balanced (re-)partitioning problem introduced by Avin et al. [8, 5]. In their model, the connected components of the graph G can contain more than k vertices and, hence, might have to be split across multiple servers. They presented a lower bound of $\Omega(k)$ for deterministic algorithms. They complemented this result by a deterministic algorithm with competitive ratio of $O(k \log k)$. This problem was also studied when the graph G follows certain random graphs models [6, 7]. For a scenario without resource augmentation, i.e., $\epsilon = 0$, there exists an $O(\ell^2 \cdot k^2)$-competitive algorithm [5] and a lower bound of $\Omega(\ell \cdot k)$ [23].

1.3 Organization

Section 2 introduces our notation and Section 3 gives an overview of the algorithmic framework. We explain the deterministic algorithm in detail, including the ILP, in Section 4 and analyze it in Section 5. Section 6 presents the randomized algorithm. Our lower bounds are presented in Section 7. Due to lack of space, omitted proofs can be found in the full verion of the paper on arxiv.

2 Preliminaries

Let us first re-introduce our model together with some definitions. We are given a graph $G = (V, E)$ with $|V| = \ell \cdot k$. In the beginning, $E = \emptyset$ and then edges are inserted in an online manner. Initially, every vertex v is assigned to one of ℓ servers such that each server is assigned exactly k vertices. We call this server the *source server* of v. For a server s we call the vertices which are initially assigned to s the *source vertices* of s. For normalization purposes, we consider each vertex to have a *volume* $\text{vol}(v)$ of $1/k$, so that the total volume of vertices initially assigned to a server is exactly 1.

After each edge insertion, the online algorithm must re-assign vertices to fulfill the *cc-condition*, i.e., so that all vertices of the same connected component of G are assigned to the same server. To this end, it can move vertices between servers at a cost of $1/k$ per vertex move. As described in the introduction, the optimum offline algorithm OPT is only allowed to place vertices with total volume up to 1 onto each server, while the online algorithm ONL is allowed to place total volume of total volume of $1 + \epsilon$ on each server, where $\epsilon > 0$ is a small constant. For notational convenience, we will place a total volume of $1+c\cdot\epsilon$ for some constant c, which does not affect our asymptotic results as the algorithm can be started with $\epsilon' = \epsilon/c$.

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

Formally, the objective is to devise an online algorithm ONL which minimizes the *(strict) competitive ratio* ρ defined as $\rho = \text{cost}(\text{ONL})/\text{cost}(\text{OPT})$, where $\text{cost}(\cdot)$ denotes the total volume of pieces moved by the corresponding algorithm. For deterministic online algorithms, the edge insertion order is adversarial; for randomized online algorithms, we assume an oblivious adversary that fixes an adversarial request sequence without knowing the random choices of the online algorithm.

The following definitions and concepts are used in the remainder of this paper.

Pieces. Our online algorithm proceeds by tracking the *pieces*, the connected components of G induced by the revealed edges. The *volume* of a piece p, denoted by $|p|$, is the total volume of all its vertices. For convenience, every server has a unique color from the set $\{1, \dots, \ell\}$ and every vertex has the *color* of the server it was *initially* assigned to. For a piece p, we define the *majority color* of p as the color that appears most frequently among vertices of p and, in case of ties, that is the smallest in the order of colors. We also refer to the corresponding server as the *majority server* of p. Similarly, we define the *majority color* for a vertex v to be the majority color of the piece of v. Note that the latter changes dynamically as the connected components of G change due to edge insertions.

Size Classes and Committed Volume. To minimize frequent and expensive moves, our approach groups the pieces into small and large pieces, and for the ILP also partitions them into a constant number of size classes. The basic idea is to "round down" the volume of a piece to a suitable multiple of $1/k$ and to call all pieces of zero rounded volume small. However, pieces can grow and, thus, change their size class, which in turn might create cost for the online algorithm. Thus, we need to use a more "refined" rounding, that gives us some control over when such a class change occurs.

More formally, let us assume that $1/4 > \epsilon \geq (10/k)^{1/4}$. We choose δ such that $\frac{1}{2}\epsilon^2 \leq \delta \leq \epsilon^2$ and $\delta = j\frac{1}{k}$ for some $j \in \mathbb{N}$. In addition, we assume $\lceil 1 \rceil_\delta - 1 \leq \delta/2$, where $\lceil \cdot \rceil_\delta$ is the operation of rounding up to the closest multiple of δ. In the full version of the paper, we show that we can always find such a δ provided that $k \geq 10/\epsilon^4$. We will also use a constant $\gamma = 2\delta < 1$.

We partition the volume of a piece into *committed* and *uncommitted* volume. The committed volume will always be a multiple of δ, while the uncommitted volume will be rather small (see below). We refer to the sum of committed and uncommitted volume as the *real volume* of the piece. We extend this definition to vertices: Each vertex is either committed or uncommitted. Now the committed volume of a piece is the volume of its committed vertices. For a piece p, we write $|p|_c$ to denote its committed volume and $|p|_u$ to denote its uncommitted volume. Hence, $|p| = |p|_c + |p|_u$.

We introduce size classes for the pieces. We say that a piece is in *class* $i \in \mathbb{N}$ if its *committed* volume is $i \cdot \delta$ (recall that committed volume is always a multiple of δ). Since the volume of a piece is never larger than 1, we have that $i \leq 1/\delta$ and thus there are only $O(1/\delta) = O(1/\epsilon^2) = O(1)$ size classes in total.

Large and Small Pieces. Intuitively, we want to refer to pieces with total volume at least ϵ as *large* and to the remaining pieces as *small*. For technical reasons, we change this as follows. A piece is *large* if its committed volume is non-zero and *small* otherwise. Thus, the small pieces are exactly the pieces in class 0. As the algorithm decides when to commit volume, it controls the transition from small to large. Note that committed volume never becomes uncommitted and, hence, a piece transitions only once from small to large.

Monochromatic Pieces. Pieces that overwhelmingly contain vertices of a single color have to be handled very carefully by an online algorithm because OPT may not have to move many vertices of such a piece and thus experience very little cost. Therefore we introduce the following notion, which needs to be different for small and large pieces since we use different scheduling techniques for them: A large piece is called *monochromatic* for its majority server s if the volume of its vertices that did not originate at s is at most δ. A small piece is called *monochromatic* if an ϵ-fraction of its volume did not originate at the majority server of the piece. We refer to pieces that are not monochromatic as *non-monochromatic.*

3 Algorithmic Framework

In this section we present our general algorithmic framework. Some further details follow in Section 4.

(1) The algorithm always maintains the current set of pieces $\mathcal{P}$, where each piece is annotated by its size class. If an edge insertion merges two pieces p_1 and p_2, into a new merged piece p_m, it holds that $|p_m|_u = |p_1|_u + |p_2|_u$ and $|p_m|_c = |p_1|_c + |p_2|_c$. We say that a merge is *monochromatic* if all of p_1, p_2 and p_m are monochromatic for the same server s. Throughout the rest of the paper, we assume w.l.o.g. that $|p_1| \leq |p_2$ and we let p_m denote the piece that resulted from the *last merge-operation.*

Invariants for Piece Volumes. Whenever the algorithm has completed its vertex moves after an edge insertion, the following invariants for piece volumes are maintained.

1. A piece p is small (i.e. has $|p|_c = 0$) iff $|p| < \epsilon$.
2. A large piece has committed volume $i \cdot \delta$ for some

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

$i \in \mathbb{N}, i > 0$. If it is monochromatic, all committed volume must be from its majority color.

3. The uncommitted volume of a large piece is at most 2δ, while the uncommitted volume of a small piece is at most ϵ. Note that $2\delta = O(1/\epsilon^2) \ll \epsilon$.

Now suppose that before a merge-operation, all pieces fulfill the invariants. Then after the merge, the new piece p_m might fulfill only the relaxed constraint $|p_m| < 2\epsilon$ if p_m is small (no committed volume) and the relaxed constraint $|p_m|_u \le \epsilon + 2\delta$ if p_m is large (with committed volume).[1] Before the next merge-operation, we will perform *commit-operations* on the piece p_m until p_m fulfills the above invariants. More concretely, if $|p_m| \ge \epsilon$ then a commit-operation is executed as long as $|p_m|_u > 2\delta$. It selects uncommitted vertices inside p_m of volume δ and sets their state to committed (which makes p_m large). If p_m is monochromatic, the commit-operation only selects vertices of the majority color, of which there is a sufficient number since for large monochromatic pieces, the volume of vertices of non-majority color is at most δ.

(2) The algorithm further maintains a *schedule* S, which is an assignment of the pieces in $\mathcal{P}$ to servers. The algorithm guarantees that this schedule fulfills certain invariants—the most important being the fact that the total volume of pieces assigned to a server does not exceed the server's capacity by much.

Adjusting Schedules. To reestablish these invariants after a change to $\mathcal{P}$, we run the *adjust schedule subroutine*. We provide the details of this subroutine in Section 4 and now give a very short summary. When the set $\mathcal{P}$ changes, this is due to one of two reasons: a *merge operation* or a *commit-operation*. Both types of changes might force us to change the old schedule S to a new schedule S'. To do so, we first solve an ILP (to be defined in Section 4) that computes the rough structure of the new schedule S'. The ILP solution defines, among other things, the number of extraordinary servers in the new schedule S'. Then we determine a concrete schedule S' that conforms to the structure provided by the ILP solution. Crucially, we have to determine an S' that is not too different from S, in order to keep the cost for switching from S to S' small.

We note that the subroutine for adjusting the schedules only moves pieces between the servers and hence does not affect the invariants for piece volumes.

Handling an Edge Insertion. We now give a high-level overview of the algorithm when an edge (u, v) is inserted. If u and v are part of the same piece, we do nothing since $\mathcal{P}$ did not change. Otherwise, assume that u is in piece p_1 and v is in p_2 with $p_1 \ne p_2$ and $|p_1| \le |p_2|$. Then we proceed as follows.

Step I *Move small to large piece:* Move the smaller piece p_1 to the server of the larger piece p_2.

Step II *Merge pieces:* Merge p_1 and p_2 into p_m. Run the adjust schedule subroutine.

Step III *Commit volume:* If $|p_m| \ge \epsilon$, then

```
    while $|p_m|_u > 2\delta$ do
        Commit volume $\delta$ for $p_m$.
        Run the adjust schedule subroutine.
```

4 Adjusting Schedules

Now we describe the subroutine for adjusting schedules in full detail. In the following, we define an ILP that helps in finding a good assignment of the pieces to servers. We ensure that when the set of pieces only changes slightly, then also the ILP solution only changes slightly. We also show how the ILP solution can be mapped to concrete schedules.

Before we describe the ILP in detail, we introduce reservation and source vectors, as well as configurations. In a nutshell, a server's reservation vector encodes how many pieces of each size class can be assigned to that server at most. A server's source vector, on the other hand, describes the structure of the monochromatic pieces for that server. A configuration is a pair of a reservation and a source vector and solving the ILP will inform us which configurations should be used for the servers in our schedule.

A *reservation vector* r_s for a server s has the following properties. For a size class $i > 0$, the entry r_{si} describes the total volume reserved on s for the (committed) volume of pieces in class i (regardless of their majority color). The entry r_{s0} describes the total volume that is reserved for uncommitted vertices (again, regardless of color); note that these uncommitted vertices could belong to small or large pieces. An entry r_{si}, $i > 0$, must be a multiple of $i\delta$ while the entry r_{s0} is a multiple of δ. Note that r_s does *not* describe which concrete pieces are scheduled on s and not even the exact number of pieces of a certain class, as it only "reserves" space.

A *source vector* m_s for server s has the following properties. For a size class $i > 0$, the entry m_{si} specifies the total committed volume of pieces in class i *that are monochromatic for* s. Again recall that a monochromatic piece only has committed volume of its majority color. The entry m_{s0} describes the total uncommitted volume of color s rounded up to a multiple

[1] The first case occurs when p_1 and p_2 are small and have volume just below ϵ (since then $|p_m|_c = |p_1|_c + |p_2|_c = 0$ and $|p_m|_u = |p_1|_u + |p_2|_u < 2\epsilon$); in this case, p_m is initially small and will become large only later when commit-operations are performed. The second case occurs when p_2 is large with $|p_2|_u$ just below 2δ and p_1 is small with $|p_1|$ just below ϵ.

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

of δ. Observe that similarly to the reservation vectors, (a) the entries m_{si} in the source vector are multiples of $i\delta$ and (b) the entry m_{s0} is a multiple of δ. In addition, (c) the entries in m_s sum up to at most $\lceil 1 \rceil_\delta$ as only vertices of color s contribute. Observe that the source vector of a server s just depends on the sizes of the s-monochromatic pieces and on which of their vertices are committed; *it does not depend on how an algorithm assigns the pieces to servers.*

A vector m is *a potential source vector* if it fulfills properties (a)-(c) without necessarily being the source vector for a particular server. Similarly, a *potential reservation vector* r is a vector where the i-th entry is a multiple of $i\delta$, the 0-th entry a multiple of δ, and r is γ-valid. Here, we say that r is *γ-valid* if $\|r\|_1 \leq 1+\gamma$. Note that there are only $O(1)$ potential reservation or source vectors since they have only $O(1/\delta) = O(1)$ entries (one per size class) and for each entry there are only $O(1/\delta) = O(1)$ choices.

A *configuration* (r, m) is a pair consisting of a potential reservation vector r and a potential source vector m. We further call a configuration (r, m) *ordinary* if $r \geq m$ (i.e., $r_i \geq m_i$ for each i) and otherwise we call it *extraordinary*. The intuition is that servers with ordinary configurations have enough reserved space such that they can be assigned all of their monochromatic pieces. Next, note that as there are only $O(1)$ potential source and reservation vectors, there are only $O(1)$ configurations in total.

Claim 4.1. *There exist only $O(1)$ different configurations (r, m).*

In the following, we assign configurations to servers and we will call a server *ordinary* if its assigned configuration is ordinary and *extraordinary* if its assigned configuration is extraordinary.

We now define the ILP. Remember that the goal in this step is to obtain a set of configurations, which we will then assign to the servers and which will guide the assignment of the pieces to the servers. Thus, we introduce a variable $x_{(r,m)} \in \mathbb{N}_0$ for each (ordinary or extraordinary) configuration (r, m). After solving the ILP, our schedules will use exactly $x_{(r,m)}$ servers with configuration (r, m). Furthermore, the objective function of the ILP is set such that the number of extraordinary configurations is minimized.

The constraints of the ILP are picked as follows. First, let V_i, $i > 0$, denote the total committed volume of all pieces in class i and let V_0 denote the total uncommitted volume of all pieces. Note that the V_i do not depend on the schedule of the algorithm. Now we add a set of constraints, which ensures that the configurations picked by the ILP reserve enough space such that all pieces of class i can be assigned to one of the servers. Second, let Z_m denote the *number* of servers with the potential source vector m at this point in time. (Recall that the source vectors of the servers only depend on the current graph and the commitment decisions of the algorithm and *not* on the algorithm's schedule.) We add a second set of constraints which ensures that for each m, the ILP solution contains exactly Z_m configurations with source vector m. Now the ILP is as follows.

$$\begin{array}{lll} \min & \sum_{(r,m):\, r \ngeq m} x_{(r,m)} & \\ \text{s.t.} & \sum_{(r,m)} x_{(r,m)} r_i/\delta \quad \geq V_i/\delta & \text{for all } i \\ & \sum_r x_{(r,m)} \quad = Z_m & \text{for all } m \end{array}$$

In the ILP we wrote r_i/δ and V_i/δ to ensure that ILP only contains integral values. Further observe that the ILP has constant size and can, hence, be solved in constant time: As there are only $O(1)$ different configurations (Claim 4.1), the ILP only has $O(1)$ variables. Also, since there are only $O(1)$ size classes i and $O(1)$ source vectors m, there are only $O(1)$ constraints.

Next, we show that an optimal ILP solution serves as a lower bound on the cost paid by OPT.

Lemma 4.1. *Suppose the objective function value of the ILP is h, then* $\text{cost}(\text{OPT}) \geq (\gamma - \delta)h = \Omega(h)$.

4.1 Schedules That Respect an ILP Solution

Next, we describe the relationship of schedules and configurations. A *schedule S* is an assignment of pieces to servers. The set of pieces assigned to a particular server s is called the *schedule for s*. A schedule for a server s with source vector m_s *respects a reservation r* if the following holds:

1. The committed volume of class i pieces scheduled on s is at most r_i.
2. The total uncommitted volume scheduled on s is at most $r_0 + 14\epsilon$.
3. If $r \geq m_s$ then all pieces that are monochromatic for s are placed on s.

A schedule *respects an ILP solution x* if there exists an assignment of configurations to servers such that:

– A server s with source vector m_s is assigned a configuration (r, m) with $m = m_s$.
– A configuration (r, m) is used exactly $x_{(r,m)}$ times.
– The schedule of each server respects the reservation of its assigned configuration.

Note that if all source vectors are different, then assigning the configurations to the servers is clear (each server s is assigned the configuration (r, m) with $m = m_s$). In the case that some source vectors appear in multiple

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

configurations, we describe a procedure for assigning the configurations to the servers below.

The next lemma shows that servers respecting a reservation only slightly exceed their capacities.

Lemma 4.2. *If the schedule for a server s respects a γ-valid reservation r, then the total volume of all pieces assigned to s is at most $1 + \gamma + 14\epsilon = 1 + O(\epsilon)$.*

4.2 How to Find Schedules In this section, we describe how to resolve the ILP and adjust the existing schedule after a merge or commit-operation so that it respects the ILP solution, in particular, that the schedule of every server respects the reservation of its assigned configuration, i.e., Properties 1-3 above. It is crucial that this step can be performed at a small cost. We present different variants: In most situations, the algorithm uses a generic variant that is based on sensitivity analysis of ILPs. However, in some special cases (cases in which OPT might pay very little) using the generic variant might be too expensive. Therefore, we develop special variants for these cases that resolve the ILP and adjust the schedule at zero cost.

Before we describe our variants in detail, note that it is not clear how to assign small pieces to servers based on the ILP solution. Hence, we define our variants such that in the first phase they move some pieces around to construct a respecting schedule but they ignore Property 2 while doing so, i.e., they only guarantee Property 1 and Property 3. After this (preliminary) schedule has been constructed, we run a *balancing procedure* (described below), which ensures that Property 2 holds. The balancing procedure only moves small pieces and we show that its cost is at most the cost paid for the first phase. As it is relatively short, we describe it first.

Balancing Procedure for Small Pieces. We now describe our balancing procedure, which moves only small pieces and for which we show that Property 2 of respecting schedules is satisfied after it finished. The balancing procedure is run after one of the variants of the ILP solving is finished.

For a server s, let $v_u(s)$ denote the total uncommitted volume scheduled at s. We define the slack of a server s by $\text{slack}(s) := r_{s0} - v_u(s)$. Note that because of the first constraint in the ILP with $i = 0$, there is always a server with non-negative slack. Next, we equip every server s with an *eviction budget* $\textit{budget}(s)$ that is initially 0. Now, any operation outside of the balancing procedure that decreases the slack must increase the eviction budget by the same amount. Such an operation could, e.g., be a piece p that is moved to s (which decreases the slack by $|p|_u$) or a decrease in r_{s0} when a new configuration is assigned to s. (Note that increasing the eviction budget increases the cost for the operation performing the increase; we will describe how we charge this cost later.) Intuitively it should follow that $\textit{budget}(s)$ roughly equals $-\text{slack}(s)$ and indeed we can show through a careful case analysis that $\textit{budget}(s) \geq -\text{slack}(s) - 2\epsilon$ (see the full version of the paper).

This eviction budget is used to pay for the cost of moving small pieces away from s when the balancing procedure is called. We say a small piece p is *movable* if either its majority color has at most a $(1-2\epsilon)$-fraction of the volume of p (the piece is far from monochromatic), or its majority color corresponds to an extraordinary server. The balancing procedure does the following for each server s:

```
while there is a movable piece $p$ on $s$ with $|p| < budget(s)$ do
    Move $p$ to a server with currently non-negative slack.
    $budget(s) = budget(s) - |p|$
```

The following lemma shows that after balancing procedure finished, $\text{slack}(s) \geq -14\epsilon$. This implies that when the balancing procedure finished, Property 2 holds since $v_u(s) = r_{s0} - \text{slack}(s) \leq r_{s0} + 14\epsilon$.

Lemma 4.3. *After the balancing procedure for a server s finished, we have that* $\text{slack}(s) \geq -14\epsilon$.

4.2.1 Overview of the Variants Next, we give the full details of the main algorithm for adjusting the schedules in different cases.

Step I *Move small to large piece:* Move the smaller piece p_1 to the server of the larger piece p_2.

Step II *Merge pieces:* Merge p_1 and p_2 into p_m. Then adjust the schedule as follows:

- If p_1 is small, then the ILP does not change and no adjustment is necessary.
- Else: if the merge is non-monochromatic *or* s is extraordinary, use the Generic Variant, otherwise, use Special Variant A.
- Run the rebalancing procedure.

Step III *Commit volume:* If $|p_m| \geq \epsilon$, then

```
while $|p_m|_u > 2\delta$: do
    Commit volume $\delta$ for $p_m$.
    If $p_m$ is non-monochromatic or $s$ is extraordinary, use the Generic Variant, otherwise, use Special Variant B.
    Run the rebalancing procedure.
```

4.2.2 The Generic Variant We now describe the Generic Variant of the schedule adjustment. Suppose the set of pieces $\mathcal{P}$ changed into $\mathcal{P}'$ due to a merge or a commit-operation.

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

The algorithm always maintains for the current set $\mathcal{P}$ an optimum ILP solution. Let x be the ILP solution for $\mathcal{P}$. When $\mathcal{P}$ changes, the algorithm runs the ILP to obtain the optimum ILP solution x' for $\mathcal{P}'$.

In the following, we first argue how to assign the configurations from x' to the servers and then we argue how we can transform a schedule S (respecting x) into a schedule S' (respecting x') with little cost.

We first assign the configurations given by x' to servers by the following greedy process. A configuration (r, m) is *free* if it has not yet been assigned to $x'_{(r,m)}$ servers. As long as there is a free configuration (r, m) and a server s that had been assigned (r, m) in schedule S, we assign (r, m) to s. The remaining configurations are assigned arbitrarily subject to the constraint that a server s with source vector m_s obtains a configuration of the form (r, m_s) for some r.

Now that we have assigned the configurations to the servers, we still have to ensure that the new schedule respects these new server configurations. We start with some definitions.

First, let $\mathcal{A}$ be the set of servers for which the set of scheduled pieces changed due to the merge- or commit-operation. For a merge-operation, these are the servers that host one of the pieces p_1, p_2, or p_m, and for a commit-operation, this is the server that hosts the piece p_m that executes the commit. Note that $|\mathcal{A}| \leq 3$. Second, let $\mathcal{B}$ be the set of servers that changed their source vector due to the merge or commit-operation. Note that for a commit-operation $|\mathcal{B}|$ could be large, because the committed volume could contain many different colors and for each corresponding server, the source vector could change by a reduction of m_0. Third, we let $\mathcal{C}$ be the set of servers that changed their assigned configuration between S and the current schedule S'. Note that $|\mathcal{C}| \leq |\mathcal{B}| + \|x - x'\|_1$.

Observe that for servers $s \notin \mathcal{A} \cup \mathcal{C}$ neither their assigned configuration (since $s \notin \mathcal{C}$) nor their set of scheduled pieces (since $s \notin \mathcal{A}$) has changed. Thus, these servers already respect their configuration and, hence, we do not move any pieces for these servers now. For the servers in $\mathcal{A} \cup \mathcal{C}$ we do the following:

1. We mark all pieces currently scheduled on servers in $\mathcal{A} \cup \mathcal{C}$ as *unassigned*.
2. Every ordinary server in $\mathcal{A} \cup \mathcal{C}$ moves all of its monochromatic pieces to itself. This guarantees Property 3 of a respecting schedule. Note that this step may move pieces away from servers in $\overline{\mathcal{A} \cup \mathcal{C}}$.
3. The remaining pieces are assigned in a first fit fashion. We say a server is *free* for class $i > 0$ if the *committed volume* of class i pieces already scheduled on it is (strictly) less than r_i. It is *free* for class 0 if the uncommitted volume scheduled on it is less than r_0.

   To schedule an unassigned piece p of class i, we determine a free server for class i and schedule p there. The first set of constraints in the ILP guarantees that we always find a free server.

This scheduling will guarantee Property 1 of a respecting schedule, i.e., for all $i > 0$ the volume of class i pieces scheduled on a server s is at most r_{si}. This holds because r_{si} is a multiple of $i\delta$. If we decide to schedule a class i piece on s because a server is free for class i then it actually has space at least $i\delta$ remaining for this class. Hence, we never overload class i, $i > 0$.

In the following, we develop a bound on the cost of the above scheme. For the analysis of our overall algorithm we use an involved amortization scheme. Therefore, the cost that we analyze here is not the real cost that is incurred by just moving pieces around but it is inflated in two ways:

(A) If we move a piece p to a server s, we increase the eviction budget of s by $|p|_u$.

(B) Whenever we change the configuration of a server from a ordinary to extraordinary, we experience an *extra cost* of $4(1+\gamma)/\delta$. This will be required later in Case IIa of the analysis.

Observe that Cost Inflation (A) clearly only increases the cost by a constant factor. Cost Inflation (B) will also only increase the cost by a constant factor as the analysis below assumes constant cost for every server that changes its configuration. Note that the Generic Variant is the only variant for adjusting the schedule for which Inflation (B) has an affect; the other variants do not move pieces around and do not generate any new extraordinary configurations.

The following lemma provides the sensitivity analysis for the ILP. Its first point essentially states that for adjusting the schedules, we need to pay cost proportional to the number of servers that change their source configuration from $\mathcal{P}$ to $\mathcal{P}'$ plus the change in the ILP solutions. The second point then bounds the change in the ILP solutions by the number of servers that change their source vectors from $\mathcal{P}$ to $\mathcal{P}'$.

LEMMA 4.4. *Suppose we are given a schedule S that respects an ILP solution x for a set of pieces $\mathcal{P}$. Let $\mathcal{P}'$ denote a set of pieces obtained from $\mathcal{P}$ by either a merge or a commit-operation, and let D denote the number of servers that have a different source vector in $\mathcal{P}$ and $\mathcal{P}'$. Then:*

1. *If x' is an ILP solution for $\mathcal{P}'$, then we can transform S into S' with cost $O(1+D+\|x-x'\|_1)$.*
2. *Then we can find an ILP solution x' for $\mathcal{P}'$ with $\|x-x'\|_1 = O(1+D)$.*
3. *If the operation was a merge-operation, then $D \leq 3$.*

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

4.2.3 Special Variant A: Monochromatic Merge

Special Variant A is used if we performed a monochromatic merge-operation of two large pieces p_1, p_2 and if the server s that holds the piece p_2 is ordinary. Then OPT may not experience any cost. Therefore, we also want to resolve the ILP and adjust the schedule S with zero cost.

Since the merge is monochromatic, all of p_1, p_2 and p_m are monochromatic for s, and since s has an ordinary configuration, p_1 and p_2 are already scheduled at s. Hence, the new piece p_m (which is generated at p_2's server) is already located at the right server s.

We obtain our schedule S' by deleting the assignments for p_1 and p_2 from S and adding the location s for the new piece p_m. Now let i_1, i_2, and i_m denote the classes of pieces p_1, p_2, and p_m, respectively (note that these classes are at least 1 as all pieces are large). Then the new ILP can be obtained by only changing the configuration vector m_s and setting

$$\begin{array}{lcccc} m'_{si_1} & := & m_{si_1} & - & |p_1|_c \\ m'_{si_2} & := & m_{si_2} & - & |p_2|_c \\ m'_{si_m} & := & m_{si_m} & + & |p_m|_c \end{array} ,$$

$$\begin{array}{lcccc} Z'_{m_s} & := & Z_{m_s} & - & 1 \\ Z'_{m'_s} & := & Z_{m'_s} & + & 1 \end{array}$$

and

$$\begin{array}{lcccc} V'_{i_1} & := & V_{i_1} & - & |p_1|_c \\ V'_{i_2} & := & V_{i_2} & - & |p_2|_c \\ V'_{i_m} & := & V_{i_m} & + & |p_m|_c \end{array} .$$

To obtain a solution x' to this new ILP, we change the reservation vector for the server s as follows.

$$\begin{array}{lcccc} r'_{si_1} & := & r_{si_1} & - & |p_1|_c \\ r'_{si_2} & := & r_{si_2} & - & |p_2|_c \\ r'_{si_m} & := & r_{si_m} & + & |p_m|_c \end{array} .$$

This does not change the $\|\cdot\|_1$-norm of the vector r because $r_{i_1} \geq m_{i_1} \geq |p_1|_c$ (this follows from the definition of m_{i_1} and the fact that $r_s \geq m_s$ holds) and because $|p_1|_c + |p_2|_c = |p_m|_c$. We obtain the ILP solution x' by setting

$$x'_{(r_s,m_s)} := x_{(r_s,m_s)} - 1 \quad \text{and} \quad x'_{(r'_s,m'_s)} := x_{(r'_s,m'_s)} + 1.$$

Note that $r_s \geq m_s$ implies $r'_s \geq m'_s$. Hence, our new ILP solution does not increase the objective function value of the ILP (i.e., the number of extraordinary configurations). In the full version of the paper we show that merging two large monochromatic pieces of a server cannot decrease the objective function value of the ILP. Therefore, the new ILP solution x', which has the same objective function value as x, is optimal.

Finally, observe that we only changed the configuration of server s and that we did not move any pieces. Hence, we can transform $\mathcal{P}$, x and S into $\mathcal{P}'$, x' and S' with zero cost.

4.2.4 Special Variant B: Monochromatic Commit

Suppose we perform a commit-operation for a monochromatic piece p_m that is located at an ordinary server s. Then OPT may not experience any cost. Therefore, we present a special variant for adjusting the schedule that also induces no cost. We perform a routine similar to Special Variant A and provide the details in the full version of the paper on arxiv.

5 Analysis

We first give a high level overview of the analysis. Let $\mathcal{P}^*$ denote the final set of pieces. A simple lower bound on the cost of OPT is as follows. Let NM denote the set of vertices that do not have the majority color within their piece in $\mathcal{P}^*$. Then cost(OPT) $\geq \frac{1}{k}|\text{NM}|$, because each vertex has volume $\frac{1}{k}$ and for each piece in $\mathcal{P}^*$, OPT has to move all vertices apart from vertices of a single color. Hence, the total volume of pieces moved by OPT is at least $\frac{1}{k}|\text{NM}|$.

We want to exploit this lower bound by a charging argument. The general idea is that whenever our online algorithm experiences some cost C, we *charge* this cost to vertices whose color does not match the majority color of their piece. If the total charge made to each such vertex v is at most $\alpha \cdot \text{vol}(v)$, then the cost of the online algorithm is at most $\alpha \cdot \text{cost(OPT)}$. When we charge cost to vertices, we will refer to this as *vertex charges.*

The difficulty with this approach is that at the time of the charge, we do not know whether a vertex will have the majority color of its piece in the end. Therefore, we proceed as follows. Suppose we have a subset S of vertices in a piece p and a subset $Q \subseteq S$ does not have the current majority color of S . Then *regardless of the final majority color of* p, a total volume of vol(Q) of vertices in S will not have this color in the end. Hence, when we distribute a charge of C evenly among the vertices of S, a charge of $\text{vol}(Q) \cdot C/\text{vol}(S)$ goes to vertices that do not have the final majority color. We call this portion of the charge *successful.*

The following lemma shows that to obtain algorithms competitive to OPT, it suffices if we bound the successful and the total vertex charges.

LEMMA 5.1. *Suppose the total successful charge is at least* $\textit{charge}_{\text{succ}}$ *while the maximum (successful and unsuccessful) charge to a vertex is at most* $\textit{charge}_{\max}$. *Then* cost(OPT) $\geq \frac{1}{k}\textit{charge}_{\text{succ}}/\textit{charge}_{\max}$.

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

Proof. Note that successful charge only goes to vertices in NM. Hence, $|\mathrm{NM}| \geq \mathit{charge}_{\mathrm{succ}}/\mathit{charge}_{\max}$, and, therefore, we obtain that $\mathrm{cost}(\mathrm{OPT}) \geq \frac{1}{k}|\mathrm{NM}| \geq \frac{1}{k}\mathit{charge}_{\mathrm{succ}}/\mathit{charge}_{\max}$. ∎

Another lower bound that we use is due to Lemma 4.1. Let $h_{\max}$ denote the maximum objective value obtained when solving different ILP instances during the algorithm. From time to time, when vertex charges are not appropriate, we perform *extraordinary charges* or just *extra charges*. In the end, we compare the total extra charge to $h_{\max}$. We stress that we only perform extra charges when extraordinary configurations are involved. This means if $h_{\max} = 0$ we never perform extra charges, as otherwise, it would be difficult to obtain a good competitive ratio.

In the following analysis, we go through the different steps of the algorithm. For every step, we charge the cost either by a vertex charge or by an extra charge. If we apply a vertex charge, we argue that (1) enough of the applied charge is successful and (2) the charge can accumulate to not too much at every vertex. For extra charges, we require a more global argument and we will derive a bound on the total extra charge in terms of $h_{\max}$ in Section 5.1.1.

5.1 Analysis Details

When merging a piece p_2 and p_1 with $|p_1| \leq |p_2|$ we proceed in several steps.

Step I: Small to Large. In this first step, we move the vertices of p_1 to the server of p_2. If p_1 and p_2 are on different servers we experience a cost of $|p_1|$. Also, we have to increase the eviction budget of the server that holds piece p_2 (if p_1 is a small piece). The cost for this step is 0 if p_1 and p_2 are on the same server and, otherwise, it is at most $2|p_1|$. We charge the cost as follows.

Case (Ia) Merge is monochromatic. If p_1, p_2, and p_m are monochromatic for the same server s we only experience cost if s is extraordinary because otherwise p_1 and p_2 are located at s. We make an extra charge for this cost.

Case (Ib) Merge is not monochromatic. We make the following vertex charges:

- Type I charge: We charge $\frac{2}{\delta} \cdot \frac{|p_1|}{|p_m|} \cdot \mathrm{vol}(v)$ to every vertex in p_m.
- Type II charge: We charge $\frac{2}{\delta} \cdot \mathrm{vol}(v)$ to every vertex in p_1.

Claim 5.1 below shows that the Type I and Type II charge at a vertex can accumulate to at most $O(\log k)$. In the following, we argue that at least a charge of $2|p_1|$ is successful. We distinguish several cases.

- If either p_2 or p_m is not monochromatic, we know that at least a volume of δ (if the non-monochromatic piece is large) or a volume of $\epsilon|p_2|$ of vertices does not have the majority color. Hence, we get that at least $\min\{\delta, \epsilon|p_2|\}\frac{2|p_1|}{\delta|p_m|} \geq 2|p_1|$ of the Type I charge is successful. The inequality uses $|p_m| \leq 1$, $|p_2| \geq \frac{1}{2}|p_m|$, and $\delta \leq \epsilon^2 \leq \epsilon/2$.
- If p_1 is not monochromatic then at least $\delta|p_1|$ volume in p_1 has not the majority color. This gives a successful charge of at least $\delta|p_1| \cdot \frac{2}{\delta} \geq 2|p_1|$.
- Finally suppose that p_1 and p_2 are monochromatic for different colors C_s and C_ℓ, respectively. If in the end C_s is not the majority color of the final piece then we have a successful charge of at least $(1-\epsilon)|p_1| \cdot 2/\delta \geq 2|p_1|$ from the Type II charge. Otherwise, C_ℓ is not the majority color and we obtain a successful charge of $(1-\epsilon)|p_2| \cdot \frac{2|p_1|}{\delta|p_m|} \geq 2|p_1|$.

Claim 5.1. *The combined Type I and Type II charge that can accumulate at a vertex v is at most $O(\log k \cdot \mathrm{vol}(v)/\delta)$.*

Step II: Resolve ILP and Adjust Schedule. In this step, we merge the pieces p_1 and p_2 into p_m and run the subprocedure for adjusting the schedule, which finds a new optimum solution to the ILP and finds a schedule respecting the ILP solution. Due to Lemma 4.4 this incurs at most constant cost. In the following, we distinguish several cases. For some cases, the bound of Lemma 4.4 is sufficient and we only have to show how to properly charge the cost. For other cases, we give a better bound than the general statement of Lemma 4.4. In the following, s denotes the server where the merged piece p_m is located now (and where p_2 was located before).

Case (IIa) p_1 small. In this case, the input to the ILP did not change. This holds because no volume was committed and no uncommitted volume changed between classes. Therefore we do not experience any cost for resolving the ILP.

However, it may happen that p_2 was not monochromatic but the merged piece p_m is. Note that this can only happen if p_2 is also small (since large pieces cannot transition from non-monochromatic to monochromatic). Suppose p_m is monochromatic for a server $s' \neq s$, and this server has an ordinary configuration. Then we have to move p_m to s' for the new schedule to respect the configuration of s'. We incur a cost of $|p_m| + |p_m|_u \leq 2|p_m|$, where $|p_m|_u$ is required to increase the eviction budget at s'. (Indeed, moving p_m to s' might cause the rebalancing procedure to move pieces away from s'.) We charge $4/\delta \cdot \mathrm{vol}(v)$ to every vertex in p_m. We call this charge a Type III charge.

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

How much of the charge is successful? Observe that p_2 was not monochromatic for s' before the merge as otherwise it would have been located at s'. This means vertices with volume at least $\delta|p_2| \geq \delta|p_m|/2$ in p_m have a color different from s' (the majority color in p_m). This means we get a successful charge of at least $\delta|p_m|/2 \cdot 4/\delta = 2|p_m|$, as desired.

To obtain a good bound on the total Type III charge accumulating at a vertex v we have to add a little tweak. Whenever a server s switches its configuration from ordinary to extraordinary, we cancel the most recent Type III charge operation for all vertices currently scheduled on s.

This negative charge is accounted for in the *extra cost* that we pay when switching the configuration of a server from ordinary to extraordinary. Recall that in Cost Inflation (B), we said that we experience an extra cost of $4(1+\gamma)/\delta$ whenever we switch the configuration of a server s from ordinary to extraordinary. This cost is used to cancel the most recent Type III charge for all pieces currently scheduled on s.

LEMMA 5.2. *Suppose a vertex v experiences a positive Type III charge at time t that is not canceled. Let t' denote the time step of the next Type III charge for vertex v, and let p and p' denote the pieces that contain v at times t and t', respectively. Then $|p'| \geq (1+\epsilon)|p|$.*

COROLLARY 5.1. *The total Type III charge that can accumulate at a vertex is only $O(\log k \cdot \mathrm{vol}(v))$.*

Case (IIb) p_1 large, merge not monochromatic. We resolve the ILP and adjust the schedule S. According to Item 1 and Item 3 of Lemma 4.4 this incurs constant cost. Let C_{IV} denote the bound on this cost. We perform a vertex charge of $C_{\mathrm{IV}}/\delta \cdot \mathrm{vol}(v)$ for every vertex in p_m. We call this charge a Type IV charge. In the following we argue that at least a charge of C_{IV} is successful. We distinguish two cases.

If one of the pieces p_1, p_2, or p_m is not monochromatic we know that at least vertices of volume δ in the piece do not have the majority color. Hence, we get that at least $C_{\mathrm{IV}}/\delta \cdot \delta \geq C_{\mathrm{IV}}$ of the Type IV charge is successful.

Now, suppose that p_1 is monochromatic for server s and p_m is monochromatic for a different server s'. Regardless of which color is the majority color in the end, there will be vertices of volume at least $(1-\epsilon)|p_1|$ that will not have this majority color. Hence, we obtain a successful charge of at least $(1-\epsilon)|p_1| \cdot C_{\mathrm{IV}}/\delta \geq (1-\epsilon)\epsilon \cdot C_{\mathrm{IV}}/\delta \geq C_{\mathrm{IV}}$, where the first step uses that p_1 is large and the second that $\delta \leq \epsilon^2 \leq (1-\epsilon)\epsilon$, which holds because $\epsilon \leq 1/4$.

CLAIM 5.2. *A vertex v can accumulate a total Type IV charge of at most $C_{\mathrm{IV}}/\delta \cdot \mathrm{vol}(v)$.*

Case (IIc) p_1 large, merge monochromatic, s extraordinary. In this case, we also resolve the ILP and adjust the schedule, which according to Item 1 and Item 3 of Lemma 4.4 incurs constant cost. Let C denote this cost. We make an extra charge of C. Observe that $C = O(|p_1|)$ because p_1 is a large piece. This will be important when we derive a bound on the total extra charge.

Case (IId) p_1 large, merge monochromatic, s ordinary. Suppose that the server s has an ordinary configuration. In this case we do not want to have any cost, because we cannot perform an extra charge as no extraordinary configurations are involved and we cannot charge against the vertices of p_m as the piece is monochromatic. We use Special Variant A for adjusting the schedule. This induces zero cost.

Step III: Commit-operation. We analyze the commit-operation. We will call a commit-operation monochromatic if it is performed on a monochromatic piece and, otherwise, we call it non-monochromatic.

Case (IIIa) p_m not monochromatic, s ordinary. The commit-operation may change the source vector of several servers. Let D denote the number of servers that changed their source vector. The cost for handling the commit-operation is at most $O(1+D)$ according to Lemma 4.4. Let C_{V} denote the hidden constant, i.e., the cost is at most $C_{\mathrm{V}}(1+D)$. We split this cost into two parts: C_{V} is the *fixed cost* and $C_{\mathrm{V}}D$ is the *variable cost* of the commit.

We charge $3C_{\mathrm{V}}/\delta \cdot \mathrm{vol}(v)$ to every vertex v in p_m. We call this charge a Type V charge. In p_m at least vertices of volume δ have not the majority color because p_m is not monochromatic. Therefore we get a successful charge of $3C_{\mathrm{V}}/\delta \cdot \delta = 3C_{\mathrm{V}}$.

Clearly, the charge is sufficient for the fixed cost. However, the remaining successful charge of $2C_{\mathrm{V}}$ may not be sufficient for the variable cost. In the following, we argue that the total remaining successful charge that is performed *for all* non-monochromatic commits is enough to cover the variable cost for these commits.

LEMMA 5.3. *Let $X_{\mathrm{nm}}(s)$ denote the number of times that a non-monochromatic commit causes a change in the source vector of s. Then the variable cost for all non-monochromatic commits is at most $\sum_s C_{\mathrm{V}} X_{\mathrm{nm}}(s) \leq 2C_{\mathrm{V}}N$, where N denotes the total number of non-monochromatic commits.*

Observe that the total remaining charge for the non-monochromatic commits is $2C_{\mathrm{V}}N$ (a charge of $2C_{\mathrm{V}}$ for

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

every commit). Hence, the previous lemma implies that this remaining charge is sufficient for the variable cost of all non-monochromatic commits.

CLAIM 5.3. *The Type V charge at a vertex v can accumulate to at most $3C_V/\delta^2 \cdot \mathrm{vol}(v)$.*

Case (IIIb) p_m monochromatic, s ordinary. Suppose we perform a commit-operation for the piece p_m. Here we use Special Variant B for resolving the ILP and adjusting the schedule. This incurs zero cost.

Case (IIIc) p_m monochromatic, s extraordinary. We resolve the ILP and adjust the schedule. The cost for this is $O(1)$, since we can use Item 1 of Lemma 4.4 with $D = 1$, because p_m is monochromatic and thus we only commit volume of color s. Let C_1 denote the upper bound for this cost. We perform an extra charge of C_1. Since the committed volume has only color s, the total number of monochromatic commits for a specific server s is at most $1/\delta = O(1)$ because each commit increases the committed volume of color s by δ. Consequently, the total extra charge that we perform for monochromatic commits of a specific server s is at most C_1/δ. To simplify the analysis of the total extra charge in Section 5.1.1 we combine all these extra charges into one extra charge of C_1/δ that is performed whenever the server s switches its state from ordinary to extraordinary *for the first time*.

5.1.1 Analysis of Extra Charges

In this section we derive a bound on the total extra charge generated by our charging scheme. Let us first recap when we perform extra charges:

(I) During the merge-operation we perform an extra charge of $O(|p_1|)$ in Case Ia and Case IIc, when the merge-operation is monochromatic for server s and s has an extraordinary configuration.
We stress the fact that whether a merge is monochromatic only depends on the sequence of merges and not on the way that pieces are scheduled by our algorithm.

(II) Whenever a server changes its configuration from ordinary to extraordinary *for the first time*, we generate an extra charge of $C_1/\delta = O(1)$ to take care of the cost of monochromatic commits (Case IIIc).

Now let $h_{\max}$ denote the maximum number of extraordinary configurations that are used throughout the algorithm. Clearly, if $h_{\max} = 0$ there is never any extraordinary configuration and the extra charge will be zero. If $h_{\max} \geq 1$, we show that the previously described deterministic online algorithm guarantees an extra charge of at most $O(\ell \log k)$.

LEMMA 5.4. *If $h_{\max} = 0$, there is no extra charge. If $h_{\max} \geq 1$, the total extra charge is $O(\ell \log k)$.*

Next, we show that the maximum vertex charge (successful or unsuccessful) is $O(\log k \cdot \mathrm{vol}(v))$.

LEMMA 5.5. *The maximum vertex charge $charge_{\max}$ (successful or unsuccessful) that a vertex v can receive is at most $O(\log k \cdot \mathrm{vol}(v))$.*

Combining Lemma 5.4 and Lemma 4.1 for extra charges and our arguments about vertex charges with Lemma 5.1, we obtain the following theorem.

THEOREM 5.1. *There exists a deterministic online algorithm with competitive ratio $O(\ell \log k)$.*

Note that we obtain an even stronger result if $h_{\max} = 0$: the cost is at most $O(\log k) \cdot \mathrm{cost}(\mathrm{OPT})$ because of the bound on the total vertex charge (and the fact that we do not have extra charges). Otherwise ($h_{\max} > 0$), the total extra charge is at most $O(\ell \log k)$, which means that we are $O((\ell \log k)/h_{\max})$-competitive. So the worst-case competitive ratio occurs when $h_{\max} = 1$.

The constant hidden in the $O(\cdot)$-notation in the theorem is $(1/\epsilon)^{O(1/\epsilon^4)}$. The exponential dependency on $1/\epsilon$ is caused by the ILP sensitivity analysis in Lemma 4.4. In particular, the hidden constants in Items 1 and 2 of the lemma are $(1/\epsilon)^{O(1/\epsilon^4)}$, since the ILP has one variable for each potential configuration and the number of such configurations in Claim 4.1 is $(1/\epsilon)^{O(1/\epsilon^2)}$. All other steps of the analysis only add factors $\mathrm{poly}(1/\epsilon)$.

Next, consider the case $\epsilon > 1$. Then the servers can store vertices of volume $2 + \epsilon'$ and the above algorithm is $O(\log k)$-competitive: Indeed, in this case, servers can always store all of their monochromatic pieces (of total volume at most 1) and never become extraordinary; thus, we are in the setting with $h_{\max} = 0$ above. Furthermore, the algorithm above never assigns pieces of volume more than $1 + \epsilon'$ to each server. Together, this bounds the total load of each server to $2 + \epsilon'$ and we obtain the following theorem.

THEOREM 5.2. *If $\epsilon > 1$, there exists a deterministic online algorithm with competitive ratio $O(\log k)$.*

6 Randomized Algorithm

How can randomization help to improve on the competitive ratio? For this observe that the cost that we charge to vertices is at most $O(\log k \cdot \mathrm{cost}(\mathrm{OPT}))$. Hence, the critical part is the cost for which we perform extra charges, which can be as large as $\Omega(\ell \log k)$ according to Theorem 7.1. A rough sketch of a (simplified) lower bound is as follows. We generate a scenario

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

where initially all servers have the same source vector but some server needs to schedule its source-pieces on different servers (as, otherwise, we could not fulfill all constraints).

In this situation, an adversary can issue merge requests for all vertices that originated at the server s that currently has its source-pieces distributed among several servers. Then the online algorithm incurs constant cost to reassemble these pieces on one server, and, in addition, has to split the source-pieces of another server between at least two servers. Repeating this for $\ell - 1$ steps gives a cost of $\Omega(\ell)$ to the online algorithm while an optimum algorithm just incurs constant cost.

The key insight for randomized algorithms is that the above scenario cannot happen if we randomize the decision of which server distributes its source-pieces among several servers. The online problem then turns into a paging problem and we use results from online paging to derive our bounds.

6.1 Augmented ILP Let M denote the set of all potential source vectors. We introduce a partial ordering on M as follows. We say $m \geq_p m'$ if any prefix-sum of m is at least as large as the corresponding prefix-sum for m'. Formally,

$$m \geq_p m' \iff \forall i\colon \sum_{j=0}^{i} m_j \geq \sum_{j=0}^{i} m'_j \ .$$

Observe that $m \geq m'$ implies $m \geq_p m'$. We adapt the ILP by adding a cost-vector c that favors large source vectors w.r.t. $\geq_p$. This means as a first objective the ILP tries to minimize the number of extraordinary configurations as before but as a tie-breaker it favors extraordinary configurations with large source vectors. For this we assign unique ids from $1, \dots, |M|$ to the source vectors s.t. $m_1 \geq_p m_2 \implies \mathrm{id}(m_1) \leq \mathrm{id}(m_2)$. Then we define the cost-vector c by setting

$$(6.1) \qquad c_{(r,m)} := \begin{cases} 0 & r \geq m \\ 1 + \lambda \,\mathrm{id}(m) & \text{otherwise} \end{cases} ,$$

for $\lambda = 1/(|M|^2 \cdot \ell)$. Given the cost-vector c, we set the objective function of our new ILP to $\sum_{(r,m)} c_{(r,m)} x_{(r,m)}$. The choice of λ together with $\|x\|_1 = \ell$ imply that $\sum_{(r,m):r \not\geq m} \lambda \,\mathrm{id}(m) x_{(r,m)} \leq \lambda \cdot |M|\ell = 1/|M| < 1$. Thus, the ILP still minimizes the number of extraordinary servers.

In the full version of the paper, we show that the sensitivity analysis for the ILP still holds. This means if we have a constant change in the RHS vector of the ILP, we can adjust the ILP solution and the schedule at the cost stated in Lemma 4.4. Similarly, when we manually adjust the ILP solution (Case IId and Case IIIb), we do not increase the cost because only the configuration of a single server s changes and this server keeps its ordinary configuration, i.e., it does not contribute to the objective function of the ILP.

A crucial property of the partial order $\geq_p$ is that source vectors of servers are monotonically decreasing w.r.t. $\geq_p$ as time progresses and as more merge-operations are processed.

Observation 6.1. *Let $m_s(t)$ denote the source vector of some server s after some timestep t of the algorithm. Then $t_1 \leq t_2$ implies $m_s(t_1) \geq_p m_s(t_2)$, i.e., the source vector of a particular server is monotonically decreasing w.r.t. $\geq_p$.*

6.2 Marking Scheme The total extra charge that is generated by our algorithm is determined by how we assign extraordinary configurations to servers. We use a marking scheme to decide which servers *may* receive an extraordinary configuration. Formally, a (randomized) *marking scheme* dynamically partitions the servers into *marked* and *unmarked* servers and satisfies the following properties:

- Initially, i.e., before the start of the algorithm, all servers are unmarked.
- Let h_m denote the number of servers with source vector m that are assigned an extraordinary configuration by the ILP, i.e., $h_m = \sum_{(r,m):r \not\geq m} x_{(r,m)}$. The marking scheme has to mark at least h_m servers with source vector m.

The cost $\mathrm{cost}(\mathcal{M})$ of a marking scheme $\mathcal{M}$ is defined as follows:

- Switching the state of a server from marked to unmarked or vice versa induces a cost of 1.
- If a marked server experiences a monochromatic merge, the cost increases by $|p_1|$, where p_1 is the smaller piece involved in the merge-operation.

Suppose for a moment that the marked servers always are exactly the servers that are assigned an extraordinary configuration. Then the above cost is clearly an upper bound on the total extra charge as define in Section 5.1.1 (up to constant factors). This is because the marking scheme pays whenever switching between marked and unmarked, while in our analysis we only make one extra charge of constant cost when a server switches to an extraordinary configuration for the first time.

In the following, we enforce the condition that a server only has an extraordinary configuration if it is marked by the marking scheme. However, the marking scheme could mark additional servers that are not extraordinary. Thus, by enforcing this condition our algorithm incurs additional cost. Suppose, e.g., that the marking scheme decides to unmark a server s that is currently marked and has been assigned an

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

extraordinary configuration. Then we have to switch the (extraordinary) configuration (r, m_s) assigned to s with an ordinary configuration (r', m_s) that currently is assigned to a different marked server s'. Note that we always find such a server because there exist at least h_{m_s} marked servers with source vector m_s. The switch can then be performed at constant cost. We make an additional extra charge for this increased cost of our algorithm. Note that the marking scheme accounts for this additional cost as it incurs cost whenever the state of a server changes. Therefore, the cost of the marking scheme can indeed serve as an upper bound on the total extra charge (including the additional extra charge). This gives the following observation.

Observation 6.2. *Let $\mathcal{M}$ be a marking scheme. The total extra charge is at most $O(\mathrm{cost}(\mathcal{M}))$.*

Next, we construct a marking scheme with small cost. For simplicity of exposition we assume that we know $h_{\max}$, the maximum number of extraordinary configurations that will be used throughout the algorithm, in advance. We describe in the arxiv version of the paper how to adjust the scheme to work without this assumption by using a simple doubling trick (i.e., make a guess for $h_{\max}$ and increase the guess by a factor of 2 if it turns out to be wrong).

We will use results from a slight variant of online paging [10]. In this problem, a sequence of page requests has to be served with a cache of size z. A request (p, w) consists of a page p from a set of $\ell \geq z$ pages together with a weight $w \leq 1$.[2] If the requested page is in the cache, the cost for an algorithm serving the request sequence is 0. Otherwise, an online algorithm experiences a cost of w. It can then decide to put the page into the cache (usually triggering the eviction of another page) at an additional cost of 1.

The cost metric for the optimal offline algorithm is different and provides an advantage to the offline algorithm. If the offline algorithm does not have p in its cache, it pays a cost of w/r, where $r \geq 1$ being a parameter of the model, and then it can decide to put p into its cache at an additional cost of 1. In [10], the authors show how to obtain a competitive ratio of $O(r + \log z)$ in this model.

The Paging Problems. Let M denote the set of potential source vectors and recall that $|M| = O(1)$. We introduce $|M|$ different paging problems, one for every potential source vector $m \in M$.

[2]Note that our problem definition slightly differs from the model analyzed by Blum et al. [10], which has $w = 1$ for every request. However, it is straightforward to show that the results of [10] carry over to our model.

Fix a potential source vector m. Let S_m denote the set of servers that have a source vector $m' \geq_p m$. Essentially, we simulate a paging algorithm on the set S_m (i.e., servers correspond to pages) with a cache of size $|S_m| - h_{\max}$ and parameter $r = \log k$.

Note that a server may leave the set S_m, but it is not possible for a server to enter this set because the source vector m_s of a server is non-increasing w.r.t. $\geq_p$ (Observation 6.1). The fact that servers may leave S_m is problematic for setting up our paging problem because this would correspond to decreasing the cache size, which is usually not possible. Therefore, we define the paging problem on the set of *all servers* and we set the cache size to $\ell - h_{\max}$, but we make sure that servers/pages not in S_m are always in the cache. This effectively reduces the set of pages to S_m and the cache size to $|S_m| - h_{\max}$.

We construct the request sequence of the paging problem for S_m as follows. A monochromatic merge for a server $s \in S_m$ is translated into a page request for page s with weight $|p_1|$, where p_1 is the smaller piece that participates in the merge-operation. Following such a merge request, we issue a page request (with weight 1) for every page/server not in S_m. This makes sure that an optimum solution keeps all these pages in the cache at all times, thus reducing the effective cache-size to $|S_m| - h_{\max}$. The request sequence stops when $|S_m| = h_{\max}$.

The Marking Scheme. We obtain a marking scheme from all the different paging algorithms as follows. A server with source vector m is marked if it is *not* in the cache for the paging problem on set S_m, or if $|S_m| \leq h_{\max}$. The following lemma shows that this gives a valid marking scheme.

Lemma 6.1. *The marking scheme marks at least h_m servers with source vector m.*

Let $\mathrm{cost}(S_m)$ denote the cost of the solution to the paging problem for S_m. The following two claims give an upper bound on the cost of the marking scheme.

Claim 6.1. *We have that*

$$\mathrm{cost}(\mathcal{M}) \leq \sum_m \left(\mathrm{cost}(S_m) + h_{\max} + O(\log k) \cdot h_{\max}\right)$$
$$= O\left(\sum_m \mathrm{cost}(S_m) + \log k \cdot h_{\max}\right).$$

Claim 6.2. *There is a randomized online algorithm for the paging problem on S_m with (expected) cost $\mathrm{cost}(S_m) \leq O((\log k + \log \ell) \cdot h_{\max})$.*

Now combining the two claims above with Lemma 4.1 and the analysis of vertex charges from Section 5, we obtain our main theorem.

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

THEOREM 6.1. *There is a randomized algorithm with competitive ratio $O(\log \ell + \log k)$.*

The constant hidden in the $O(\cdot)$-notation in the theorem is $(1/\epsilon)^{O(1/\epsilon^4)}$. This follows from the same arguments mentioned after the statement of Theorem 5.1 and the fact that Claim 6.1 only adds another $(1/\epsilon)^{O(1/\epsilon^2)}$-factor since the number of monochromatic configurations is $(1/\epsilon)^{O(1/\epsilon^2)}$.

7 Lower Bounds

In this section, we derive lower bounds on the competitive ratios for deterministic and randomized algorithms. In particular, we show that any deterministic algorithm must have a competitive ratio of $\Omega(\ell \log k)$ and any randomized algorithm must have a competitive ratio of $\Omega(\log \ell + \log k)$.

We note that the lower bounds derived in this section also apply to the model studied by Henzinger et al. [18]. Their model is slightly more restrictive than ours in that eventually, every server must have exactly one piece of volume 1 (resp. k in their terminology); in contrast, in our model, servers may eventually host multiple pieces smaller than 1. However, our lower bounds are designed such that they also fulfill the definition of the model by Henzinger et al.

7.1 Lower Bounds for Deterministic Algorithms

THEOREM 7.1. *For any $k \geq 32$ and any constant $1/k \leq \epsilon \leq 1/32$ such that ϵk is a power of 2, any deterministic algorithm must have a competitive ratio of $\Omega(\ell \log k)$.*

We devote the rest of this subsection to prove the theorem.

Set m be a positive integer such that $\epsilon k = 2^m$. As $\epsilon \leq 1/32$ it follows that $k \geq 2^{m+5}$. Fix any deterministic algorithm ONL. We will show that there exists a sequence σ_{ONL} of edge insertions such that the cost of the optimum offline algorithm is $O(\epsilon)$, while the cost of ONL is $\Omega(\epsilon \ell \log(\epsilon k))$. The sequence σ_{ONL} depends on ONL, i.e., edge insertions will depend on which servers ONL decides to place the pieces.

Definitions. We assume that the servers are numbered sequentially. As before, each server has a color and every vertex is colored with the color of its initial server. For simplicity, we assume server i has color i. The *main server* of a color c is the server that, out of all servers, currently contains the largest volume of color-c vertices *and* whose index number of all such servers is the smallest[3].

A piece is called *single-colored* if all vertices of the piece have the same color. If a single-colored piece with color c is not assigned to the main server for c, it is called *c-away* or simply *away*. Any piece of volume at least 2ϵ is called a *large* piece, all other pieces are called *small*. We say *two pieces are merged* if there is an edge insertion connecting the two pieces.

Initial Configuration. Initially each of the ℓ servers contains one large single-colored piece of volume 2ϵ and $(1-2\epsilon)k$ isolated vertices, each of volume $1/k$. The large pieces of on servers 1, 2, and 3 are called *special*. A color c is *deficient* if the total volume of all *small* c-away pieces is at least ϵ.

Sequence σ_{ONL}. The first two edge insertions merge the three special pieces into one (multi-colored) special piece of volume 6ϵ. As we will show *any* algorithm now has at least one deficient color. Note that all small pieces are single-colored and have volume $2^0/k = 1/k$.

Now σ_{ONL} proceeds in *rounds*. We will show that there is a deficient color at the end and, thus, also at the beginning of every round. In each round only small pieces of the same (deficient) color are merged such that their volume doubles. As a result, all small pieces continue to be single-colored and, at the end of each round, all small pieces of the same color have the same volume, namely $2^i/k$ for some integer i, *except* for potentially one piece of smaller volume, which we call the *leftover piece*. A leftover piece is created if the number of small items of color c and volume $2^i/k$ at the beginning of a round is an odd number. If this happens, it is merged with the leftover piece of color c of the previous rounds (if it exists) to guarantee that there is always just one leftover piece of color c. To simplify the notation we will use the term *almost all small pieces of color c* to denote all pieces of color c except the leftover piece of color c.

A round of σ_{ONL} consists of the following sequence of requests among the small pieces: If there exists a deficient color c such that the volume of almost all small pieces of color c is $2^i/k$ for some integer i and $2^i/k < \epsilon$, then σ_{ONL} contains the following steps.
If there are small pieces of color c and volume $2^i/k$ that are currently on different servers, they are connected by an edge, otherwise two such pieces on the same server are connected by an edge. Repeat this until there is at most one small piece of color c of volume $2^i/k$ left. Once this happens and if such a piece exists, it becomes a *leftover* piece of color c and if another leftover piece of color c exists from earlier rounds, the two are merged.

[3]The difference between *majority* and *main* color is that we added the second condition to guarantee that the main server of a color is unique.)

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

Note that almost all pieces of color c have now volume $2^{i+1}/k$ and the leftover piece has smaller volume. If $2^{i+1}/k \geq \epsilon$, merge all *non-special* (i.e. the small and the non-special large) pieces of color c and call color c *finished*. As long as there are at least two unfinished deficient colors, start a new round.

Once there are no more rounds we will show that there is exactly one unfinished deficient color c^* left and there are at least 2^{j+3} small pieces of color c^* and volume $\epsilon/2^j$ for some integer $j \geq 1$. Furthermore there exists the special piece of volume 6ϵ (which is not single-colored) and for every other color there exists one piece of volume 1 (if it does not belong to $\{1,2,3\}$) or of volume $1-2\epsilon$ (if it belongs to $\{1,2,3\}$).

Final Merging Steps. To guarantee that each piece has volume exactly 1 at the end, the remaining pieces of volume less than 1 are now suitably merged. First 2^{j+3} of the pieces of color c^* and volume $\epsilon/2^j$ are merged into 3 pieces of volume 2ϵ each, the rest is merged into one piece. Then consider two cases: If $c^* \in \{1,2,3\}$, let c' and c'' be the other two colors of $\{1,2,3\}$. In this case σ_{ONL} merges the first small piece of volume 2ϵ of c^* with the non-special piece of c' and then merges the second small piece of volume 2ϵ of c^* with the non-special piece of c''. Then all the remaining pieces of color c^* are merged with each other and with the special piece.

If $c^* \notin \{1,2,3\}$, then σ_{ONL} merges the small pieces of volume $\epsilon/2$ of color c^* with the non-special piece of color 1 and then does the same with color 2 and 3. Then all the remaining (small and large) pieces of color c^* are merged with each other and with the special piece.

Note that as a consequence all piece now have volume 1.

We show first that all the assumptions made in the description of σ_{ONL} hold. Specifically the next three lemmata will show that (1) after initialization and after each round there exists a deficient color for any algorithm, that (2) for each color c almost all small pieces of color c have volume $2^i/k$ and the leftover piece of color c has volume less than $2^i/k$, and that (3) at the beginning of the final merging steps there is exactly one unfinished deficient color left and there are at least 2^{j+3} small pieces of this color that have volume $\epsilon/2^j$ for some integer $j \geq 1$. Then we will show that algorithm ONL has cost at least $\Omega(\epsilon\ell\log(\epsilon k))$ to process the sequence.

LEMMA 7.1. *At the beginning of each round there exists an unfinished deficient color for algorithm* ONL.

Proof. After initialization and after each round there exists (1) the special piece of volume 6ϵ that is not single-colored and (2) for each color there exist small single-colored pieces of total volume at least $1-2\epsilon$. *Now suppose by contradiction that no color is deficient.* Then for each color c the total volume of small c-away pieces is less than ϵ, i.e. the volume of the small pieces on the main server for c is at least $1-3\epsilon$. As no server can have pieces of total volume more than $1+\epsilon$ assigned to it and $\epsilon \leq 1/8$, it follows that the non-special pieces on the main server of c require volume more than $(1+\epsilon)/2$, and, thus, each server can be the main server for at most one color. As there are as many colors as there are server, each server is the main server for exactly one color and each color has exactly one main server.

Now consider the server s^* on which the special piece is placed and let it be the main server for some color c. Then the total volume of the pieces on s^* is 6ϵ for the special piece. If c is not deficient, s^* has load at least $1-3\epsilon$ for the non-special pieces of color c. Thus, the server's load is at least $1+3\epsilon$ which is not possible. Hence, there must exist a deficient color.

Next we show that there is always an unfinished deficient color. This is trivially true after initialization as all colors are unfinished. Let us now consider the end of a round. Note that every color c that is finished has a non-special piece of volume at least $1-2\epsilon$ and, thus, *the special piece cannot be placed on the main server of a finished color c.* Recall that every non-deficient color has pieces of total volume at least $1-\epsilon$ on its main server. Thus, *the special piece cannot be placed the main server of any non-deficient color.* Thus, the special piece can only be placed on a server that is not the main server of a finished deficient or a non-finished color. If every deficient color is finished, every color has a main server and the special piece cannot be placed on any of them. As, however, there are as many servers as there are colors, it would follow that the special piece is not placed on any server, which is not possible. Thus, there must exist a deficient unfinished color. ∎

LEMMA 7.2. *For each color c it holds at the beginning and end of each round that almost all small pieces of color c have volume $2^i/k$ for some integer i and the other small piece has even smaller volume.*

Proof. By induction on the number of rounds. The claim holds after initialization for $i=1$ for every color. During each round for some color c the pieces of color c and volume $2^i/k$ are merged pairwise, and the possible left-over piece of volume $2^i/k$ is merged with the leftover piece of earlier rounds, if it exists. From the induction claim it follows that the leftover piece of earlier rounds has volume less than $2^i/k$. Thus, the resulting leftover piece has volume less than $2^{i+1}/k$. Furthermore, the pieces of the other colors remain unchanged. Thus, the claim follows. ∎

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

Lemma 7.3. *At the beginning of the final merging steps there is exactly one unfinished deficient color left and there are at least 2^{j+3} small pieces have volume $\epsilon/2^j$ for some integer $j \geq 1$.*

Proof. Lemma 7.1 holds after each round, thus, also after the final round. It shows that there is still at least one deficient unfinished color. As there are no more rounds, there at most one deficient unfinished color, which implies that there is exactly one deficient unfinished color. As it is unfinished, all its small pieces have volume less than ϵ. For the rest of the proof we only consider small pieces of this color.

Recall that $\epsilon k = 2^m$ and $k \geq 2^{m+5}$. Initially there are $k - 2\epsilon k \geq 2^{m+5} - 2^{m+1}$ small pieces of volume $1/k$ each. Let k' be the largest power of 2 that is at most $k - 2\epsilon k$. It follows that $k - 2\epsilon k \geq k' > k/2 - \epsilon k \geq 2^{m+4} - 2^m$. Thus, initially there are at least k' small pieces of volume $1/k = \epsilon/2^m$ each. Let j be any integer with $0 \leq j \leq m$ such that exactly $m - j$ rounds were executed for this color. Thus, there are at least $k'/2^{m-j}$ pieces of volume $2^{m-j}\epsilon/2^m = \epsilon/2^j$ at the beginning of the final merging steps. Note that $k'/2^{m-j} \geq (2^{m+4} - 2^m)/2^{m-j} = 2^{j+4} - 2^j \geq 2^{j+3}$. Thus, there are at least 2^{j+3} pieces of volume $\epsilon/2^j$. As the color is unfinished, each small piece has volume less than ϵ, i.e. $j \geq 1$. Thus the lemma holds. □

Next we analyze how many rounds are performed for a given color until it is finished. Consider any color c. The number of rounds necessary to increase the volume of almost all small pieces of color c from $1/k$ to ϵ is $\log(\epsilon k)$ as ϵk is a power of 2. Each round roughly halves the number of small pieces. Thus, we only have to show that there are enough small pieces available initially so that $\log(\epsilon k)$ many rounds are possible for color c.

Lemma 7.4. *For each finished color $\log(\epsilon k)$ rounds are executed.*

Proof. Fix a color c and consider in this proof only pieces of color c. As $\epsilon \leq 1/8$ and each initial small piece is a single vertex, there are $k - 2\epsilon k \geq 3k/4$ many such small pieces initially. Let k' be the largest power of 2 that is at most $3k/4$. Note that $k' > 3k/8$. Thus the number of small pieces of volume at least ϵ is at least $k'/2^{\log(\epsilon k)} > 3/(8\epsilon) \geq 3$. Hence for each finished color $\log(\epsilon k)$ rounds will be executed. □

As there are ℓ different colors, it suffices to show that in almost every round algorithm ONL moves pieces with total volume $\Omega(\epsilon)$ to achieve the desired lower bound of $\Omega(\epsilon\ell\log(\epsilon k))$ for the cost of ONL.

Lemma 7.5. *In one round of the above process, except in the last round for each color,* ONL *moves vertices with volume $\Omega(\epsilon)$. In total, the algorithm moves vertices with volume $\Omega(\epsilon\ell\log(\epsilon k))$*

Proof. Fix a color c and only consider pieces of color c in this proof. Note that when two pieces of different servers, of volume $2^i/k$ each, are merged, at least one of them has to change its server, resulting in a cost of $2^i/k$ for the algorithm. We proved in Lemma 7.1 that at the beginning of each round a deficient color exists. A deficient color has away pieces of total volume at least ϵ, i.e., there are small pieces of total volume at least ϵ not on the main server. During a round, as shown by Lemma 7.2, almost all of these pieces have volume $2^i/k$ for some integer i and their total contribution to the total volume of all away pieces of color c is larger than $\epsilon - 2^i/k$ (subtracting out the volume of the potentially existing leftover piece of even smaller volume). Thus, as long as $\epsilon - 2^i/k \geq \epsilon/2$, i.e., in all but the last round, the total volume of all the away pieces excluding the leftover piece is larger then $\epsilon/2$. In the following when we talk about a small piece, we mean a small piece that is *not* the leftover piece and we fix a round that is not the last round. We will show that at least $\epsilon/2$ volume is merged by pieces on different servers in this round, resulting in at least $\epsilon/4$ cost for the algorithm.

Now consider two cases: (1) If the main server s^* contains small pieces of total volume at least $\epsilon/2$, then every away piece can be merged with a small piece either on s^* or on a different server. Thus at least $\epsilon/2$ volume is merged by pieces on different servers. (2) If, however, the main server contains small pieces of total volume less than $\epsilon/2$, then *every* server contains small pieces of total volume less than $\epsilon/2$. Small pieces of different servers are merged until all remaining small pieces are on the same server. However, this server has less than $\epsilon/2$ volume of small pieces, i.e., more than $\epsilon/2$ volume must have been merged between different servers. Thus in both cases the algorithm has cost at least $\epsilon/4$. The second claim follows immediately from the discussion preceding the lemma. □

Lemma 7.6. *In total,* OPT *moves vertices with volume $O(\epsilon)$.*

Proof. Right at the beginning OPT places the special piece of volume 6ϵ on server s^* and moves the small pieces of color c^* that are merged in the final merging step with a different color to the main server for the corresponding color. Thus, none of the other steps cause any cost for OPT. Thus, OPT only has cost $O(\epsilon)$. □

The previous two lemmas imply a lower bound on the competitive ratio of $\Omega(\ell\log(\epsilon k))$ for deterministic algorithms. This finishes the proof of the theorem.

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

7.2 Lower Bounds for Randomized Algorithms

THEOREM 7.2. *Any randomized online algorithm must have a competitive ratio of* $\Omega(\log \ell + \log k)$.

PROPOSITION 7.1. *If* $\epsilon < 1/6$*, then any randomized online algorithm must have a competitive ratio of* $\Omega(\log \ell)$.

Proof. We use Yao's principle [28] to derive our lower bound and provide a randomized hard instance against a deterministic algorithm. The hard instance starts by merging the vertices of each server into monochromatic pieces of volume 2ϵ each. Now the hard instance arbitrarily picks three pieces with different majority colors and merges them into a piece of volume 6ϵ and we call this piece *special*. Next, the hard instance proceeds in $\ell - 1$ rounds. Before the first round all servers are *unfinished*. In round i, the hard instance picks an unfinished server s uniformly at random. Now the hard instance uniformly at random picks monochromatic pieces with color s of total volume $1 - 2\epsilon$ and merges them in arbitrary order; after that we call s *finished*. When all $\ell - 1$ rounds are over, a final configuration in which all pieces have volume 1 is obtained as follows. First, observe that there is a unique unfinished server s^*. Now the hard instance merges the special piece and monochromatic pieces of color s^* of total volume $1 - 6\epsilon$. The remaining monochromatic pieces of color s^* are merged with the components of the finished servers from which the vertices of the special piece originated. All other monochromatic components of volume 2ϵ are merged with the large monochromatic components with the same color as the piece itself.

For a given schedule, we say that an unfinished server s is *split* if monochromatic pieces with color s and of volume at least ϵ are not scheduled on s. Now observe that after each round there exists a server which is split: First, observe that none of the finished servers can store its monochromatic piece of volume $1 - 2\epsilon$ together with the special piece of volume 6ϵ. Now if none of the (unfinished) servers was split, one of them would contain all of its monochromatic pieces of total volume at least $1 - 2\epsilon$ together with the special piece of volume 6ϵ. Thus, the total load of the server is $1 + 4\epsilon$ which is not a valid schedule.

Next, we show that if a server s is split, then the algorithm has moved monochromatic pieces with color s of volume at least ϵ: First, suppose the algorithm has scheduled all monochromatic pieces of color s on some server $s' \neq s$. Then the algorithm has paid at least $1 - 2\epsilon \geq \epsilon$ to move the monochromatic pieces of color s to s'. Second, suppose the monochromatic pieces of color s are scheduled on at least two different servers. Then the algorithm must have moved at least one monochromatic piece of color s away from s. Since s is unfinished and all monochromatic pieces of s have volume 2ϵ, the algorithm has paid at least ϵ for moving monochromatic pieces of color s.

Now we analyze the cost paid by the algorithm. Observe that before round i there are $\ell - i + 1$ unfinished servers and at least one of them is split. Let s be a split server. Thus with probability $1/(\ell - i + 1)$ the hard instance picks the split server s. It follows from the previous claims that the algorithm paid at least ϵ to move pieces of color s. Since the above arguments hold for each round, the total expected cost of the algorithm is

$$\sum_{i=1}^{\ell-1} \epsilon \frac{1}{\ell - i + 1} = \sum_{i=2}^{\ell} \epsilon \frac{1}{i} = \Omega(\epsilon \log \ell).$$

Next, observe that OPT never moves more than $O(\epsilon)$ volume: Indeed, the hard instance only merges pieces in which all vertices have the same color except when (1) creating the special piece of volume $O(\epsilon)$, (2) merging the special piece with the vertices from s^* and (3) merging the small pieces from s^* with the large pieces of the servers from which the special piece originated. All of these steps can be performed by only moving volume $O(\epsilon)$.

Thus, the competitive ratio is $\Omega(\log \ell)$. $\square$

PROPOSITION 7.2. *Any randomized algorithm must have a competitive ratio of at least* $\Omega(\log k)$.

Proof. We use Yao's principle [28] to derive our lower bound and provide a random instance against a deterministic algorithm. In the instance all pieces initially have volume $1/k$, i.e., the pieces consist of single vertices. The lower bounds proceeds in $\log k$ rounds. In each round, we pick a perfect matching between all pieces uniformly at random. Thus, after i rounds, all pieces have volume $2^i/k$. Note that after $\log k$ rounds all pieces have volume 1 and we have obtained a valid final configuration.

We claim that in each round the algorithm has to move volume $\Omega(\ell)$. Suppose we are currently in round i. Now consider two pieces p_1 and p_2 which are merged during a single round. Then the probability that p_1 and p_2 are assigned to different servers is $\Omega((\ell - 1)/\ell) = \Omega(1)$. Furthermore, observe that each piece has volume $2^i/k$ and in total there are $n/2^i$ pieces. Now by linearity of expectation we obtain that the expected volume moved by the algorithm in round i is $\Omega(2^i/k \cdot n/2^i) = \Omega(\ell)$.

Next, observe that the total cost paid by the algorithm is $\Omega(\ell \cdot \log k)$ since there are $\log k$ rounds. Furthermore, OPT never moves volume more than $O(\ell)$

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited

because it moves each vertex at most once. Thus, the competitive ratio is $\Omega(\log k)$. □

References

[1] Emmanuel Abbe. Community detection and stochastic block models: Recent developments. *JMLR*, 18(177):1–86, 2018.

[2] Anna Adamaszek, Artur Czumaj, Matthias Englert, and Harald Räcke. An $O(\log k)$-competitive algorithm for generalized caching. In *SODA*, pages 1681–1689, 2012.

[3] Dan Alistarh, Jennifer Iglesias, and Milan Vojnovic. Streaming min-max hypergraph partitioning. In *NeurIPS*, pages 1900–1908, 2015.

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

[5] Chen Avin, Marcin Bienkowski, Andreas Loukas, Maciej Pacut, and Stefan Schmid. Dynamic balanced graph partitioning. In *SIAM J. Discrete Math (SIDMA)*, 2019.

[6] Chen Avin, Louis Cohen, Mahmoud Parham, and Stefan Schmid. Competitive clustering of stochastic communication patterns on a ring. In *Journal of Computing*, 2018.

[7] Chen Avin, Louis Cohen, and Stefan Schmid. Competitive clustering of stochastic communication patterns on the ring. In *NETYS*, 2017.

[8] Chen Avin, Andreas Loukas, Maciej Pacut, and Stefan Schmid. Online balanced repartitioning. In *DISC*, 2016.

[9] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC*, pages 267–280, 2010.

[10] Avrim Blum, Carl Burch, and Adam Kalai. Finely-competitive paging. In *FOCS*, pages 450–458, 1999.

[11] Alan Borodin, Nati Linial, and Michael E. Saks. An optimal on-line algorithm for metrical task system. *Journal of the ACM*, 39(4):745–763, 1992. Also appeared in *STOC*, pages 373–382, 1987.

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

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

[14] Björn Feldkord, Matthias Feldotto, Anupam Gupta, Guru Guruganesh, Amit Kumar, Sören Riechers, and David Wajc. Fully-dynamic bin packing with little repacking. In *ICALP*, pages 51:1–51:24, 2018.

[15] 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.

[16] Amos Fiat, Yuval Rabani, and Yiftach Ravid. Competitive k-server algorithms. *J. Comput. Syst. Sci.*, 48(3):410–428, 1994.

[17] Tobias Forner, Harald Räcke, and Stefan Schmid. Online balanced repartitioning of dynamic communication patterns in polynomial time. In *APoCS*, 2021.

[18] Monika Henzinger, Stefan Neumann, and Stefan Schmid. Efficient Distributed Workload (Re-)Embedding. *SIGMETRICS/POMACS*, 3(1):13:1–13:38, 2019.

[19] Dorit S Hochbaum and David B Shmoys. Using dual approximation algorithms for scheduling problems theoretical and practical results. *Journal of the ACM (JACM)*, 34(1):144–162, 1987.

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

[21] Manor Mendel and Steven S. Seiden. Online companion caching. *Theoretical Computer Science*, 324(2–3):183–200, 2004.

[22] Jeffrey C. Mogul and Lucian Popa. What we talk about when we talk about cloud network performance. *SIGCOMM Comput. Commun. Rev. (CCR)*, 2012.

[23] Maciej Pacut, Mahmoud Parham, and Stefan Schmid. Brief announcement: Deterministic lower bound for dynamic balanced graph partitioning. In *PODC*, 2020.

[24] Peter Sanders, Naveen Sivadasan, and Martin Skutella. Online scheduling with bounded migration. *Math. Oper. Res.*, 34(2):481–498, 2009.

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

[26] Isabelle Stanton. Streaming balanced graph partitioning algorithms for random graphs. In *SODA*, pages 1287–1301. SIAM, 2014.

[27] Luis Vaquero, Félix Cuadrado, Dionysios Logothetis, and Claudio Martella. Adaptive partitioning for large-scale dynamic graphs. In *SOCC*, pages 35:1–35:2, 2013.

[28] Andrew Chi-Chih Yao. Probabilistic Computations: Toward a Unified Measure of Complexity. In *FOCS*, pages 222–227, 1977.

[29] Neal E. Young. On-line caching as cache size varies. In *SODA*, pages 241–250, 1991.

Copyright © 2021 by SIAM
Unauthorized reproduction of this article is prohibited