Online Balanced Allocation of Dynamic Components

Rajmohan Rajaraman
Northeastern University, Boston, MA, USA

Omer Wasim
Northeastern University, Boston, MA, USA

Abstract

We introduce Online Balanced Allocation of Dynamic Components (OBADC), a problem motivated by the practical challenge of dynamic resource allocation for large-scale distributed applications. In OBADC, we need to allocate a dynamic set of at most $k\ell$ vertices (representing processes) in $\ell > 0$ clusters. We consider an over-provisioned setup in which each cluster can hold at most $k(1+\varepsilon)$ vertices, for an arbitrary constant $\varepsilon > 0$. The communication requirements among the vertices are modeled by the notion of a *dynamically changing component*, which is a subset of vertices that need to be co-located in the same cluster. At each time t, a request r_t of one of the following types arrives:

1. *insertion* of a vertex v forming a singleton component $\{v\}$ at unit cost.
2. *merge* of (u, v) requiring that the components containing u and v be merged and co-located thereafter.
3. *deletion* of an existing vertex v at zero cost.

Before serving any request, an algorithm can migrate vertices from one cluster to another, at a unit migration cost per vertex. We seek an online algorithm to minimize the total migration cost incurred for an arbitrary request sequence $\sigma = (r_t)_{t>0}$, while simultaneously minimizing the number of clusters utilized. We analyze competitiveness with respect to an optimal clairvoyant *offline* algorithm with identical (over-provisioned) capacity constraints.

We give an $O(\log k)$-competitive algorithm for OBADC, and a matching lower-bound. The number of clusters utilized by our algorithm is always within a $(2+\varepsilon)$ factor of the minimum. Furthermore, in a resource augmented setting where the optimal offline algorithm is constrained to capacity k per cluster, our algorithm obtains $O(\log k)$ competitiveness and utilizes a number of clusters within $(1+\varepsilon)$ factor of the minimum.

We also consider OBADC in the context of machine-learned predictions, where for each newly inserted vertex v at time t: i) with probability $\eta > 0$, the set of vertices (that exist at time t) in the component of v is revealed and, ii) with probability $1-\eta$, no information is revealed. For OBADC with predictions, we give a $O(1)$-*consistent* and $O(\min(\log \frac{1}{\eta}, \log k))$-*robust* algorithm.

2012 ACM Subject Classification Theory of computation → Online algorithms

Keywords and phrases online algorithms, competitive ratio, algorithms with predictions

Digital Object Identifier 10.4230/LIPIcs.ITCS.2025.81

Acknowledgements This work was partially supported by NSF award CCF-2335187.

1 Introduction

Modern large-scale applications are distributed over multiple physical machines or computing clusters in data centers, generating a considerable amount of network traffic. In a typical setting, a cloud service provider provisions computational resources (such as CPU, memory, storage) in the form of virtual machines (VMs) to users who pay a cost proportional to their resource requirements which varies over time. However, operating and allocating these resources in a "demand-aware" manner to minimize cost, and optimize performance and energy utilization while fulfilling service-level agreements (SLAs) for various users is technically challenging. This has garnered a surging interest in resource allocation strategies in cloud computing frameworks in recent years [7, 9, 1, 31].

Consider the setting where a massively distributed application attributed to a single cloud user may be spread across multiple VMs, necessitating inter-VM communication. If such VM's are allocated in the same cluster, inter-VM communication consumes little network bandwidth, leading to low *communication cost* and potential energy savings [31, 32]. However if inter-VM communication happens over physically separated clusters, this may compromise quality of service, responsiveness, and incur a large consumption of network resources. Almost all virtualization systems support "live-migration" of VMs between various clusters with the aim of reducing latency, improving resource-allocation and energy savings [12, 16, 17]. However, VM migration must be performed sparingly, as it creates network overhead and may compromise the responsiveness of the VM being migrated.

In addition to inter-process communication patterns that evolve over time, another challenge is to provision resources *dynamically* to users under varying resource requirements. This leads to the removal of existing virtual machines and the insertion of new virtual machines [6]. Thereafter, new communication patterns emerge between newly re-allocated VMs. Such re-allocations also incur a cost for the cloud service provider which is inevitable, in addition to future defragmentation of resources caused by multiple re-allocations over time [30].

Finally, *energy consumption* of data centers due to an ever-increasing need for large-scale cloud computing is well documented [19]. In practice, migration of VMs together with other load-balancing strategies have been exploited to reduce the number of "active" physical machines to save energy [16, 19, 17]. For example, VMs on under-loaded clusters are migrated so that such clusters can save energy by moving to an idle state. Thus, it is highly desirable to ensure that the number of "active" clusters *dynamically scales* with the workload.

In this paper, we address the challenges posed by unknown communication patterns, dynamic resource requirements, and energy consumption by introducing the problem of **Online Balanced Allocation of Dynamic Components** (OBADC). We model the scenario where clusters and VMs are homogeneous, and migration and inter-VM communication costs are uniform across different clusters, while the communication patterns and dynamic allocation requests are unknown.

1.1 Problem Formulation and Main Results

Our problem formulation is inspired by the online balanced graph partitioning (OBGR) problem, introduced by Avin, Bienkowski, Loukas, Pacut and Schmid [2], and considers additional technical aspects including the dynamics of VMs, and a secondary goal of minimizing the number of active clusters. In OBADC, we have a set V of at most $n \leq k\ell$ vertices (representing VMs) at any given point in time, and $\ell > 0$ clusters denoted by $S_1, S_2, ..., S_\ell$, each with capacity k. Initially $V = \emptyset$ and S_i for all $i \in [\ell]$ is empty. We model the communication requirements among the vertices using the notion of a dynamically changing *component*, which is a subset of vertices that need to be co-located in the same cluster. The request sequence $\sigma = \{r_t\}_{t \geq 1}$ is revealed online, such that each request r_t is of the following type:

1. *Insertion of a vertex v*: Insert v (and a corresponding singleton component $\{v\}$) at unit cost, to any cluster which can accommodate it. The cost of insertion models a one-time "startup" cost to initialize a VM in a cluster.
2. *Deletion of a vertex v*: Delete existing vertex v from its component C. The cost of deletion is zero; all vertices in $C \backslash \{v\}$ are still required to be co-located.
3. *Merge vertices u and v* : Merge the components containing u and v into a single component so that vertices in the resulting component are co-located thereafter.

Insertions and deletions of vertices model VM allocation and de-allocation, while merge requests model the scenario when frequently communicating VM's dedicated to a single application or a customer are consolidated for efficiency or energy savings [28].

The request sequence σ induces a unique set of components denoted by $\mathcal{C}$, over time. The migration cost of a single vertex is unit. All vertices of a single component are required to be assigned to the same cluster. Thus, we assume that the size of any component $C \in \mathcal{C}$ is most k at all times, and that there exists a component assignment to clusters such that each cluster is assigned a maximum of k vertices.

Competitive Analysis. We adopt the competitive analysis framework to analyze OBADC algorithms. A deterministic (resp. randomized) online algorithm $\mathcal{A}$ is said to be ρ-competitive (alternatively, ρ is the competitive ratio of $\mathcal{A}$) if for any input sequence σ, the cost (resp. expected cost) $c(\mathcal{A}, \sigma)$ incurred by $\mathcal{A}$ in σ satisfies $c(\mathcal{A}, \sigma) \leq \rho \mathrm{OPT}(\sigma) + \gamma$ where γ is a constant independent of the length of σ and $\mathrm{OPT}(\sigma)$ is the cost of an optimal offline algorithm denoted by OPT, which has complete information about the request sequence σ in advance. Abusing notation slightly, we refer to $\mathrm{OPT}(\sigma)$ as OPT when σ is clear from context.

Resource Competitiveness. To capture the practical objective of utilizing a number of clusters proportional to the total resource requirement (which varies over time), we seek algorithms which are also *competitive* with respect to the *number of clusters* utilized by OPT. Let n_t denote the number of vertices at any time t, and let α_t, β_t denote the number of clusters used to assign all n_t vertices at time t by OPT and an online algorithm $\mathcal{A}$. We say that $\mathcal{A}$ is μ-resource competitive if for all time t, $\beta_t \leq \mu \alpha_t$.

OBADC with Over-Provisioning. For the sake of obtaining polynomial-time online algorithms, we consider an overprovisioned setting, similar to the one considered in a recent work [23]. Here, each cluster C_i for $i \in [\ell]$ has capacity $(1+\varepsilon)k$ for some $\varepsilon \in (0, 1)$. While seemingly similar to a $(1+\varepsilon)$ resource-augmented model, we emphasize that in the over-provisioned setup, both the online algorithm and the optimal offline algorithm OPT are constrained to an *identical* cluster capacity of $(1+\varepsilon)k$ for any cluster C_i. As a result, competitiveness is analyzed on a stricter benchmark, in that the optimal offline and online algorithm have access to the same amount of resources, in strong contrast to resource augmentation where competitiveness is analyzed with respect to an unaugmented offline algorithm (see Chapter 4 of [25]).

Our main result is a $O(\log k)$-competitive algorithm for OBADC in the over-provisioned setting. We show that the competitive ratio of our algorithm is asymptotically optimal by establishing a lower bound. Furthermore, our algorithm is $(2+\varepsilon)$-resource competitive.

▶ **Theorem 1.** *Consider any instance of OBADC with ℓ clusters each of capacity $(1+\varepsilon)\ell$ and arbitrary sequence of insertion, deletion, and merge requests such that at any time the resulting components can be allocated to the ℓ clusters using capacity at most k. There exists an online algorithm that achieves cost at most $O(\log k)$ times the optimal, while utilizing at most $(2+\varepsilon)$ times more clusters than the minimum required at each time step.*

Furthermore, every online algorithm incurs cost $\Omega(\log k)$ times the optimal for the worst-case instance.

In a resource augmented setting where the optimal offline algorithm is constrained to capacity k per cluster, our algorithm obtains $O(\log k)$ competitiveness, while utilizing a number of clusters within $(1+\varepsilon)$ factor of the minimum. The proof of resource-competitiveness is deferred to the full version of the paper.

OBADC with Machine-Learned Predictions. We next investigate OBADC in the context of "algorithms with predictions" which is a growing research area [25, 15] with the over-arching goal of circumventing pessimistic limitations of lower and upper bounds for various problems in online, dynamic and approximation algorithms.

Prediction Model: At any time step t when a vertex v is inserted, with probability $\eta \geq 0$, we also receive a set P_v consisting of all vertices in V_t that lie in the same component as v, where V_t denotes the set of vertices at the beginning of time step t. Note that with probability $1-\eta$, we do not receive any information on the component of v. Furthermore, P_v only consists of vertices in v's component which exist at time t, and contains no information about future vertices in v's component which may arrive after time t.

Our "prediction error" is quantified by the confidence $\eta > 0$ of the predictions, similar to [11] and we analyze the performance of a learning-augmented online algorithm for OBADC in a standard *consistency-robustness* framework [11, 15, 24]. An algorithm $\mathcal{A}$ for OBADC with predictions is said to be α-consistent and β-robust if it is α-competitive when $\eta = 1$, and β-competitive where β is a non-increasing function of η. Roughly speaking if $\eta = 1$, and we have perfect predictions, then one would desire constant-factor competitiveness. On the other hand, as η approaches 0, the performance of the algorithm should gracefully degrade to the competitive ratio obtained by the best-known online algorithm for the problem. Our second result provides a near-tight consistency-robustness tradeoff for OBADC.

▶ **Theorem 2.** *There exists an $O(1)$-consistent and $O(\min(\log \frac{1}{\eta}, \log k))$-robust algorithm for OBADC with predictions in the over-provisioned setting.*

1.2 Technical Overview

In this section, we present a high level overview of our techniques towards obtaining our algorithms. We ignore several intricacies in the following discussion.

1.2.1 Overview of the Algorithms

Our algorithm for OBADC classifies components in terms of their volume (which refers to their size). A component is said to be *small* if it has volume at most Dk where $D = \Omega(\varepsilon)$, and large otherwise. Let $\mathcal{C}_S, \mathcal{C}_L$ denote the set of small and large components maintained at all times. Component volumes are rounded up to the nearest power of $(1 + \frac{\varepsilon}{4})$, such that components whose size is in the interval $[(1 + \frac{\varepsilon}{4})^{i-1}, (1 + \frac{\varepsilon}{4})^{i})$ belong to component class i. Thus, the total number of *small* component classes is $O(\log k)$ while the number of *large* component classes is $O(1)$. Our algorithm *assigns* volume to each component C on a cluster such that the assigned volume is upper bounded by $(1 + \frac{\varepsilon}{4})|C|$. The precise volume assigned by our algorithm is more fine-grained to further distinguish components within the same component class.

The assignment of large components is accomplished by solving an integer linear program (ILP) whose objective function seeks to minimize the number of clusters utilized. We remark that large components are assigned *independently* of small components. Since the number of large component classes is $O(1)$, the ILP can be solved in polynomial time. Invoking a result from [26] allows us to limit the total "change" in large component assignments after component merges (resp. vertex deletions), which lead to increase (resp. decrease) in a component's class. Small components are assigned in a *first-fit* fashion. Our algorithm maintains a "tight" assignment of components throughout time which guarantees $(2+\varepsilon)$-resource competitiveness.

The main challenge arises in maintaining these assignments *dynamically*, while simultaneously minimizing the total migration cost and clusters utilized. To accomplish this, our algorithm maintains several invariants. We give an overview of our algorithm as follows.

Insertion of singleton components are handled by provisioning volume on a cluster, with sufficient residual volume whenever possible. For a merge request (u, v), if u, v are in the same component C, the algorithm does nothing; else distinct components C_i, C_j containing u and v respectively, are merged into a new component C_m where w.l.o.g. $|C_i| \geq |C_j|$.

When C_m is *small* and the volume assigned to C_i is at least $|C_m|$, the smaller component C_j is migrated (unless it is assigned on the same cluster as C_i). Otherwise, if there is sufficient residual volume on the cluster to which C_i is assigned, C_m is assigned volume and vertices in C_j are migrated (if necessary). If the residual volume is insufficient, this implies that at least k vertices are assigned to this cluster. Thereafter, vertices in C_i and C_j are migrated to another cluster with sufficient residual volume. This may lead to a reduction in the assigned volume of C_j's cluster. Whenever the residual volume of a cluster decreases below a certain threshold, our algorithm migrates small components as necessary from other clusters to ensure resource-competitiveness.

On the other hand, if C_m is large such that C_m belongs to a higher component class than C_i, our algorithm solves an ILP; otherwise, vertices in C_j are migrated (if necessary). If the ILP is solved, this may lead to multiple components (both large and small) being reassigned and migrated. We give a greedy procedure to assign large components, which together with the sensitivity analysis result from [26], allows us to bound the total migration cost by $O(f(\varepsilon)k)$. If the request sequence consists of only vertex insertions and component merges (i.e. only monotonic increases in component sizes), we can establish $O(\log k)$-competitiveness by a relatively elegant charging argument (see Section 3).

However, for non-monotonic changes in component sizes a major challenge arises towards ensuring resource-competitiveness, since vertex deletions in one component may necessitate migration of other unrelated components to maintain a *tight* packing. To this end, our algorithm always maintains the property that the assigned volume of any component C is within a $(1 + \frac{\varepsilon}{4})$ factor of $|C|$ under both component merges and vertex deletions. For small components undergoing vertex deletions, volume reassignment can be done after a constant fraction of vertices in C are deleted. This may require migration of small components to C's cluster. On the other hand, if vertex deletions in C lead to a drop in C's component class we solve an ILP to reassign large components. This may be prohibitive in general because repeated vertex insertions and deletions can lead to a cost of $O(f(\varepsilon)k)$ every time an ILP is solved. To circumvent this, our algorithm works with a *relaxed* volume threshold (and component class categorization) when "deciding" whether to re-solve an ILP under vertex deletions. Thus, for example, certain large components may be treated as class i components by our algorithm even when their volume is less than $(1 + \frac{\varepsilon}{4})^{i-1}$. However, once at least $\Omega(\varepsilon^2 k)$ vertices are deleted from such components, our algorithm re-solves the ILP. We show that treating certain components in this manner does not violate resource competitiveness, since the total number of large components which can be assigned to any cluster is $O(\frac{1}{\varepsilon})$.

For the learning-augmented model, we enhance our algorithm of Theorem 1 by executing the merges indicated by the predictions at the time of insertion. That is, on the insertion of a vertex v whose predicted set $P(v)$ is non-empty, our algorithm issues merge requests of the form (v, u) for all $u \in P(v)$. This allows us to inherit several desirable properties of our algorithm of Theorem 1 while minimizing the total migration cost when augmented with predictions. The resulting algorithm also yields a smooth interpolation between our algorithm without predictions and one with full predictions.

We present an overview of our analyses in Section 3 and Section 4 for our $O(\log k)$-competitive algorithm and learning-augmented algorithm, respectively.

1.3 Related Work

Online graph partitioning problems. At a high-level, OBADC is related to previous work on online partitioning of dynamically changing graphs. Related problems include variants of online graph coloring and the OBGR problem. In recent work, Azar et al. [3] introduce an online graph recoloring problem in which edges of a graph arrive online and an algorithm needs to distribute the vertices among ℓ clusters such that for any edge, the two endpoints must be in different clusters. Subsequent work [23] considers a capacitated variant of online recoloring. A key distinction between such coloring problems and OBADC is that while an edge in the former requires the endpoints to be *separated*, an edge in the latter requires the endpoints to be *co-located*, leading to different technical considerations.

OBADC is perhaps most closely related to OBGR [2], in which there is a *static* set V of $n = kl$ vertices and a set of $l > 0$ clusters each holding k vertices initially. An online sequence $\sigma = \{(u_t, v_t)\}_{t>0}$ of communication requests arrives over time. The cost of serving request (u_t, v_t) is 1 if u_t and v_t are in distinct clusters and 0 otherwise. Prior to serving any request, a vertex can be migrated at a cost of $\alpha \in \mathbb{Z}^{>0}$. The cost incurred by an online algorithm is the total communication and migration cost to serve all requests in σ. OBGR has a $O(k^2l^2)$-competitive algorithm [2] in the non-augmented setting while a lower bound of $\Omega(kl)$ is known. In recent work, a $O(kl2^{O(k)})$-competitive algorithm was given in [4], which is optimal for constant k. In the resource augmented setting, with $(2+\varepsilon)$-augmented cluster capacity an $O(k \log k)$-competitive deterministic algorithm was given in [2]. For the same setting, a polynomial time algorithm was recently presented in [10]. A lower bound of $\Omega(l)$ holds even in the $(1+\varepsilon)$-augmented setting where $\varepsilon < 1/3$ [20].

In [14], Henzinger et al. introduced the learning model of the problem in which the vertex set of the graph G_σ induced by the request sequence σ can be partitioned into $V_1, ...V_l$ such that $|V_i| = k$ for all i and there are no inter-cluster requests (u, v) where $u \in V_i.v \in V_j$ for $j \neq i$. For the learning model, the lower bound of $\Omega(kl)$ holds in the non-augmented setting. Allowing $(1+\varepsilon)$-augmentation, the best deterministic and randomized algorithms are $\Theta(l \log k)$ and $\Theta(\log k + \log l)$-competitive respectively which are shown to be tight[13].

OBADC vs. OBGR. We describe the similarities and differences between OBADC and OBGR. OBADC resembles a "fully dynamic" version of OBGR in the learning model, but differs as follows. In OBADC, vertices arrive and depart over time, and hence the online algorithm has to adapt to communication patterns which evolve over time between both existing vertices and new vertices. The migration costs are identical between the two models; however in OBADC, an insertion of a new vertex incurs a cost of 1 and we aim to have the total number of clusters utilized by an online algorithm to adapt with the number of vertices. This may incur strategic migrations of vertices to satisfy this additional objective, in contrast to OBGR where migrations are performed to solely minimize communication overhead.

Resource-augmentation and over-provisioning. To circumvent pessimistic lower-bounds and obtain polyonomial time algorithms, many online algorithmic problems have been studied in the *resource-augmented* model (e.g., see [21, 33, 34]), going back as far as the earliest work on caching [29]. OBGR has also been studied in the resource-augmentation model [13, 22, 10, 2], where OPT is constrained to a capacity of k on each cluster, while an online algorithm can utilize capacity $(1+\varepsilon)k$ for some $\varepsilon > 0$. In contrast, we study OBADC in an *over-provisioned* setting, which yields stronger guarantees than resource-augmentation since the algorithm and the optimal have the same resources. Any competitive ratio in

the over-provisioned model immediately implies the same bound in the resource augmented model; for the measure of number of clusters utilized, we are able to get stronger bounds in the resource-augmentation model (see the remark after Theorem 1 in Section 1.1).

Learning-augmented algorithms. Another approach toward addressing pessimistic bounds in online and approximation algorithms is to incorporate the growing availability of machine-learned predictions in the design of algorithms [11, 18, 15, 5]. A central theme in this area is the following: given access to an imperfect machine-learned oracle which can "predict" characteristics of future request sequences or actions (in the context of online algorithms [24, 15]), edge updates (in the context of dynamic algorithms [5]) or optimal solutions (in the context of approximation algorithms [8]), can improved competitive ratios, running/update times, or approximation ratios be obtained? The imperfection of predictions is quantified by prediction error, which is problem and instance dependent, or a confidence parameter [11]. There is a great variation in terms of various prediction models and prediction errors which have been utilized and there is no singular notion of either, which can be useful or applicable to wide variety of problems.

Our study of online OBADC with predictions is also inspired from prior empirical work in VM allocation and migration, in which various predictions on resource utilization in clusters [17], future VM requests and migrations [19], and communication traffic from historically collected data have been utilized towards efficient provisioning of resources.

1.4 Outline and Preliminaries

Section 2 presents our main algorithm for OBADC. In Section 3, we give a proof sketch of the competitive ratio of our algorithm. The proofs of resource competitiveness and correctness are deferred to the full version of the paper. Finally, Section 4 presents a learning-augmented algorithm for OBADC with predictions, together with a proof sketch of Theorem 2.

We present some useful notation. Let V_t denote the set of vertices at any given point in time t such that $n_t = |V_t| \leq kl$. We use the notation $[i]$ for any $i \in \mathbb{Z}^+$ to refer to the set $\{1, 2, \ldots, i\}$. The volume of a component C is simply its size, and denoted by $|C|$.

Our algorithms maintain a set of components $\mathcal{C} = \{C_1, C_2, ...\}$ induced on V_t for all time t. We use $\mathcal{S} = \{S_1, S_2, \ldots, S_\ell\}$ to denote the set of all clusters. When there is merge a request (u, v) between distinct components C_i and C_j, our algorithms merge C_i and C_j into C_m. A merge is viewed as deletion of components C_i and C_j from $\mathcal{C}$ and an insertion of C_m to $\mathcal{C}$.

2 An $O(\log k)$-competitive algorithm

In this section, we present an *asymptotically* optimal deterministic $O(\log k)$-competitive algorithm for OBADC. We begin by giving some definitions and invariants that our algorithm maintains.

2.1 Definitions and Invariants

Component Classes. Our algorithm classifies components according to their size. For a component $C \in \mathcal{C}$, we say that C is in class i if $|C| \in [(1 + \frac{\varepsilon}{4})^{i-1}, (1 + \frac{\varepsilon}{4})^{i})$, where $1 \leq i \leq \lfloor \ln_{(1+\frac{\varepsilon}{4})} k \rfloor$. We say that a component is small if it is in class i where $i \leq c_s$ where $c_s \leq \lfloor \ln_{(1+\frac{\varepsilon}{4})} \frac{\varepsilon k}{4} \rfloor$. Thus, any small component has size at most Dk where $\frac{\varepsilon}{5} \leq D \leq \frac{\varepsilon}{4}$. All the other components are said to be large.

A large component is understood to be in class i if it is in class $i + c_s$. By the above definitions, it follows that a large component of class i has size in $[Dk(1+\frac{\varepsilon}{4})^{i-1}, Dk(1+\frac{\varepsilon}{4})^{i})$. The following lemma bounds the number of large component classes.

▶ **Lemma 3.** *The number of large component classes is* $c_l \leq \frac{26}{\varepsilon^2} = O(1)$.

Proof. Note that a large component has volume at least $\frac{\varepsilon k}{5}$ and at most k. Thus, $c_l \leq \frac{\ln(\frac{5}{\varepsilon})}{\ln(1+\frac{\varepsilon}{4})} + 1$. Since $1 - \frac{1}{x} \leq \ln x \leq x - 1$ for any $x \in \mathbb{R}^{\geq 0}$, it follows that $c_l \leq \frac{(5/\varepsilon)}{1/(1+\varepsilon/4)} \leq \frac{4+\varepsilon}{\varepsilon} \cdot \frac{5}{\varepsilon} + 1 \leq \frac{26}{\varepsilon^2}$ since $\varepsilon < 1$. ◀

Assigned Volume. For every component $C \in \mathcal{C}$, our algorithm provisions volume on some cluster $S_i \in \mathcal{S}$ to which it is assigned, which we refer to as the assigned volume and denote it by $A(C)$.

For any cluster $S_i \in \mathcal{S}$ let $A(S_i)$ denote the total assigned volume for all components assigned to S_i. Let R_i denote the residual *unassigned* volume of S_i, where $R_i = (1+\varepsilon)k - A(S_i)$. We present a fine-grained scheme to assign volumes to components as follows.

Let $A_{ij} = (1 + j\frac{\varepsilon}{16})(1+\frac{\varepsilon}{4})^{i-1}$, for $j \in \{0, 1, 2, 3, 4\}$. Thus, $A_{i0} = (1+\frac{\varepsilon}{4})^{i-1}$ and $A_{i4} = (1+\frac{\varepsilon}{4})^{i}$. Our algorithm maintains the following invariant for assigned volumes at all times.

▶ **Invariant 1** (Assigned Volume). *For all classes i, and any class i component such that $|C| \in [(1+\frac{\varepsilon}{4})^{i-1}, (1+\frac{\varepsilon}{4})^{i})$, one of the following statements holds:*

1. *if $A_{i0} \leq |C| < A_{i1}$, then $|C| \leq A(C) \leq A_{i3}$.*
2. *if $A_{i1} \leq |C| < A_{i2}$, then $|C| \leq A(C) \leq A_{i4}$.*
3. *if $A_{i2} \leq |C| < A_{i3}$, then $|C| \leq A(C) \leq (1+\frac{\varepsilon}{16})(1+\frac{\varepsilon}{4})^{i} = A_{i+1,1}$.*
4. *if $A_{i3} \leq |C| < A_{i4}$, then $|C| \leq A(C) \leq (1+\frac{\varepsilon}{8})(1+\frac{\varepsilon}{4})^{i} = A_{i+1,2}$.*

The following lemma is immediate based on the invariant above.

▶ **Lemma 4.** *If Invariant 1 is maintained, then for any component C, $|C| \leq A(C) \leq (1+\frac{\varepsilon}{4})|C|$.*

Active Clusters. A cluster is said to be *active* if there is at least one component assigned to it. The set of all active clusters is denoted by $\mathcal{S}_A$.

Marked Clusters. Our algorithm marks and un-marks active clusters over time. If a cluster is marked, then its residual volume is small. We denote the set of marked clusters by $\mathcal{S}_M$, where $\mathcal{S}_M \subseteq \mathcal{S}_A$. Initially, all clusters are unmarked, i.e. $\mathcal{S}_M = \emptyset$. Our algorithm maintains the following invariant at all times.

▶ **Invariant 2.** *For a cluster $S_j \in \mathcal{S}_M$, $R_j \leq \frac{\varepsilon k}{3}$.*

Furthermore, let $\mathcal{S}_S$ denote the set of clusters on which only small components (and no large components) are assigned and let $\mathcal{S}_L$ denote the set of clusters to which at least one large component is assigned. To ensure that the number of active clusters is at most $(2+\varepsilon)$ times the minimum needed, our algorithm maintains the following invariant.

▶ **Invariant 3.** *At any time t, if $|\mathcal{S}_S| > 0$ then both of the following hold: i) either $\mathcal{S}_L = \emptyset$ or $\mathcal{S}_L \subseteq \mathcal{S}_M$ (i.e. all clusters in $\mathcal{S}_L$ are marked) and, ii) exactly one cluster in $\mathcal{S}_S$ is unmarked.*

Marked Components. Our algorithm marks and un-marks certain components when their component class decreases due to vertex deletions. If a large component of class $i \in [c_l - 1]$, (resp. a small component of class c_s) is marked, then it is treated as a large component of class $i+1$ (resp. a large component of class 1). Intuitively, we mark a component to ensure that the ILP is not solved too frequently if its volume shrinks only slightly due to deletions. The set of marked components at any time is denoted by $\mathcal{C}_M$. Note that all marked components are *treated* as large components by our algorithm, thus $\mathcal{C}_M \subseteq \mathcal{C}_L$. Our algorithm maintains the following invariant for marked components.

▶ **Invariant 4.** *At any point in time, a marked component $C \in \mathcal{C}_L$ satisfies one of the following conditions:*

1. *C is a small component of class c_s, such that $|C| \geq Dk - \frac{\varepsilon^2 k}{100}$.*
2. *C is a large component of class $i \in [c_l - 1]$, such that $|C| \geq Dk(1+\frac{\varepsilon}{4})^i - \frac{\varepsilon^2 k}{100}$.*

Data Structures. Our algorithms maintain the sets $\mathcal{C}_S$ and $\mathcal{C}_L$ of small and large components, respectively, together with the set $\mathcal{C}_M$ of marked components. For a component C, the assigned volume $A(C)$ is maintained, and for all $S_i \in \mathcal{S}$, $A(S_i)$ and R_i are maintained. Furthermore, the sets of clusters $\mathcal{S}_A, \mathcal{S}_M, \mathcal{S}_S$ and $\mathcal{S}_L$ are maintained.

In the following section, we outline our approach for assigning large components which is completely independent of the assignment of small components.

2.2 An Approach to Assign Large Components

We first define the notion of a signature, similar to [22, 13]. Intuitively, a signature encodes the number of large components of each component class that can be assigned to a single cluster while respecting cluster capacity.

▶ **Definition 5.** *A signature $\tau = (\tau_1, \tau_2, ..., \tau_{c_l})$ is a non-negative vector of dimension c_l where τ_i denotes the number of large components of class i such that $\sum_{i=1}^{c_l}(Dk(1+\frac{\varepsilon}{4})^{i-1} - \frac{\varepsilon^2 k}{100})\tau_i \leq k$.*

Let $\mathcal{T} = \{T_1, T_2, ...\}$ denote the set of all possible signatures. The following lemma bounds the size of $\mathcal{T}$.

▶ **Lemma 6.** *The total number of signatures, $|\mathcal{T}|$ is bounded by $(\frac{6}{\varepsilon})^{\frac{26}{\varepsilon^2}} = O(1)$.*

Proof. We first note that $Dk(1+\frac{\varepsilon}{4})^{i-1} - \frac{\varepsilon^2 k}{100} \geq Dk - \frac{\varepsilon^2 k}{100} \geq \frac{\varepsilon k}{5} - \frac{\varepsilon^2 k}{100} \geq \frac{\varepsilon k}{6}$ for any $i > 0$. Thus, any entry τ_i is upper bounded by $\frac{k}{\varepsilon k/6} = \frac{6}{\varepsilon}$. Since the number of large component classes is $\frac{26}{\varepsilon^2}$ by Lemma 3, the total number of signatures is at most $(\frac{6}{\varepsilon})^{\frac{26}{\varepsilon^2}} = O(1)$. ◀

We solve an integer linear program (ILP) in polynomial time, and obtain signatures which guide the placement of large components. Let T_{ij} denote the j^{th} entry of a signature $T_i \in \mathcal{T}$. For each signature T_i, we let $x_i \in [0, \ell]$ denote the number of clusters which are assigned signature T_i. Let σ_j denote the number of class j large components at any given point in time. Our ILP is as follows.

ILP.

$$\min \quad \sum_{i=1}^{|\mathcal{T}|} x_i \quad \text{s.t.} \quad \sum_{i=1}^{|\mathcal{T}|} T_{ij} x_i = \sigma_j \ \forall j \in [c_l], \quad x_i \in [0, \ell] \ \forall i \in [\mathcal{T}].$$

In matrix form, our ILP has $n_r = O(c_l) = O(1)$ rows and $n_c = O(|\mathcal{T}|) = O(1)$ columns. Thus, the ILP can be solved in polynomial time.

Our algorithm solves this ILP whenever σ_j changes for any $j \in [c_l]$. This could be after components are merged leading to a larger component with a higher component class or, vertex deletions from a component leading to a decrease in its component class. In both cases, at most three components in $\mathcal{C}$ change, and as a result, at most three component classes change. We invoke a well-known result in sensitivity analysis from [27], which quantifies the *change* in optimal solutions of the ILP.

▶ **Theorem 7** ([27]). *Let A be an integral $n_r \times n_c$ matrix, such that each subdeterminant of A is at most Δ in absolute value; let b' and b'' be column n_r-vectors, and let c be a row n_c-vector. Suppose $\max\{cx|Ax \leq b' : x \text{ integral}\}$ and $\max\{cx|Ax \leq b'' : x \text{ integral}\}$ are finite. Then for each optimum solution z' of the first maximum there exists an optimum solution z'' of the second maximum such that $\|z' - z''\|_\infty \leq n_c\Delta(\|b' - b''\|_\infty + 2)$.*

We prove the following lemma.

▶ **Lemma 8.** *After a merge request or deletion request, the number of signatures which change in the optimal solution to the ILP is $f(\varepsilon) = \frac{|\mathcal{T}|^{2+|\mathcal{T}|}}{\varepsilon^{|\mathcal{T}|}} = O(1)$ for constant $\varepsilon > 0$.*

Proof. Whenever two components are merged or a deletion causes a large component's size class to decrease, the RHS vector in our ILP changes by at most 1 in the infinity norm. To bound the sub-determinant, we use the Hadamard inequality to derive that $\Delta \leq (n_c A_{\max})^{n_c/2}$ where $A_{\max}$ denotes the maximum entry (in absolute value) of the constraint matrix A. Each entry in the constraint matrix of our ILP has value either 1 or T_{ij} so that $A_{\max} \leq \frac{k}{Dk} = O(\frac{1}{\varepsilon})$. As a result, $\Delta = O((\frac{|\mathcal{T}|}{\varepsilon})^{|\mathcal{T}|})$. Thus, the optimal solution to the ILP changes by $O(|\mathcal{T}|\Delta)$ in the infinity norm. Since x has dimension $|\mathcal{T}|$, the number of signatures which change between any two optimal solutions is $O(|\mathcal{T}|^2\Delta) = O(1)$. ◀

Greedy Assignment of Signatures to Clusters. Given the solution $x = (x_1, x_2, ..., x_{|\mathcal{T}|})$ from the ILP, we perform a *greedy* assignment of signatures to clusters which minimizes the number of clusters whose assigned signatures change. Let $x' = (x'_1, x'_2, ..., x'_{|\mathcal{T}|})$ denote the current signature, such that there are exactly x'_i clusters which are assigned signature T_i. Our algorithm for greedy assignment, `Assign-Signatures` works as follows.

Let $z_i = x_i$ for all $i \in [|\mathcal{T}|]$ and $\mathcal{S}' := \mathcal{S}_A$ denote the set of active clusters. We iterate over all $i \in [|\mathcal{T}|]$: while $z_i > 0$, if there exists an active cluster S_j in $\mathcal{S}'$ which is currently assigned signature T_i, z_i is decremented, S_j is removed from $\mathcal{S}'$ and we continue. Else if no cluster in $\mathcal{S}'$ is assigned signature T_i, we pick an arbitrary active cluster S_j in $\mathcal{S}_A$ and assigns T_i to S. If $\mathcal{S}' = \emptyset$, our algorithm picks an arbitrary inactive cluster $S_j \in \mathcal{S} \backslash \mathcal{S}_A$, adds S_j to $\mathcal{S}_A$ and assigns T_i to S.

Let $\mathcal{S}_N \subseteq \mathcal{S}_A$ denote the set of clusters which are assigned a new signature after the call to `Assign-Signatures`. Any cluster $S \in \mathcal{S}_N$ satisfies one of the following conditions prior to the call to `Assign-Signatures`: i) S was inactive, ii) S was active but its signature changed after the call to `Assign-Signatures` or iii) S was active and was not assigned any signature prior to the call to `Assign-Signatures`.

Let $\mathcal{S}_O \subseteq \mathcal{S}$ denote the set of clusters which were assigned a signature prior to solving the ILP, and are no longer assigned a signature. Finally, let $\mathcal{S}_I$ denote the set of clusters whose assigned signatures are identical before and after the call to `Assign-Signatures`. Our algorithm returns sets $\mathcal{S}_N, \mathcal{S}_O$ and $\mathcal{S}_I$. Clearly, the assigned signatures change only for clusters which are in $\mathcal{S}_N \cup \mathcal{S}_O$.

By virtue of subroutine `Assign-Signatures` and Lemma 8, the following lemma is immediate.

Algorithm 1 `Assign-Signatures`.

Let $x' = (x'_1, x'_2, ..., x'_{|\mathcal{T}|})$ denote the current signature.
$\mathcal{S}' \leftarrow \mathcal{S}_A$, $\mathcal{S}_N \leftarrow \emptyset$, $\mathcal{S}_I \leftarrow \emptyset$.
Solve the ILP and let $x = (x_1, x_2, ..., x_{|\mathcal{T}|})$ denote the obtained signature
for $i \in [|\mathcal{T}|]$ **do**
    $z_i = x_i$.
    while $z_i > 0$ **do**
        if there exists $S_j \in \mathcal{S}'$ assigned a signature T_i **then**
            $z_i \leftarrow z_i - 1$, $\mathcal{S}' \leftarrow \mathcal{S}' \backslash \{S_j\}$, $\mathcal{S}_I \leftarrow \mathcal{S}_I \cup \{S_j\}$.
        else
            if $\mathcal{S}' = \emptyset$ **then**
                Pick an arbitrary cluster $S_j \in \mathcal{S} \backslash \mathcal{S}_A$, and assign T_i to S_j.
                $\mathcal{S}_N \leftarrow \mathcal{S}_N \cup \{S_j\}$.
            else
                Pick an arbitrary active cluster S_j in $\mathcal{S}'$ and assign T_i to S_j.
                $\mathcal{S}_N \leftarrow \mathcal{S}_N \cup \{S_j\}$, $\mathcal{S}' \leftarrow \mathcal{S}' \backslash \{S_j\}$.
$\mathcal{S}_O \leftarrow \mathcal{S}_L \backslash (\mathcal{S}_I \cup \mathcal{S}_N))$.
$\mathcal{S}_L \leftarrow \mathcal{S}_N \cup \mathcal{S}_I$.
Return $(\mathcal{S}_N, \mathcal{S}_O, \mathcal{S}_I)$.

▶ **Lemma 9.** *After a call to* `Assign-Signatures`, $|\mathcal{S}_N \cup \mathcal{S}_O| = f(\varepsilon) = O(1)$.

For all clusters $S_i \in \mathcal{S}_I$ whose assigned signatures do not change, our algorithm does not modify the assignment of large components. Let $\mathcal{C}'_L$ denote the set of all large components in $\mathcal{C}_L$ which are not assigned to any cluster in $\mathcal{S}_I$. It follows from Lemma 9 that the total volume of all large components in $\mathcal{C}'_L$ is $O(kf(\varepsilon))$.

Components in $\mathcal{C}'_L$ are assigned to clusters in $\mathcal{S}_N$, corresponding to the newly assigned signatures. Let $\mathcal{C}'_S$ denote the set of all small components previously assigned to $\mathcal{S}_N \cup \mathcal{S}_O$. From Lemma 9, it follows that the total volume of all components in $\mathcal{C}'_S$ is $O(kf(\varepsilon))$. Our algorithm re-assigns small components in $\mathcal{C}'_S$ in a *first-fit* fashion, explained in the next section. The following Lemma is immediate from Lemma 8, Lemma 9 and our preceding discussion.

▶ **Lemma 10.** *The total volume of components in* $\mathcal{C}'_L \cup \mathcal{C}'_S$ *is bounded by* $O(kf(\varepsilon) = O(k)$.

2.3 The Full Algorithm

In this section, we present our algorithm which takes as input a request r_t at time t, and maintains an assignment of components to clusters while satisfying Invariants 1 through 4. Our algorithm relies on various subroutines, which we present in the following.

Subroutine `Assign-Volume`. This subroutine takes as input a component C, and a cluster $S_j \in \mathcal{S}_A$ and assigns volume for C on S_j. In the case when C is a class $i-1$ marked component, it is treated as a class i component and assigned volume A_{i2}.

Observe that Invariant 1 is satisfied for any component C after a call to `Assign-Volume`.

Subroutine `Assign-Small`. The subroutine `Assign-Small` is invoked whenever a small component C needs to be assigned volume on a cluster. We iterate over active unmarked clusters $S_j \in \mathcal{S}_A \backslash \mathcal{S}_M$, and if $R_j \geq (1 + \frac{\varepsilon}{4})|C|$, C is assigned to S_j. Else, S_j is marked and added to $\mathcal{S}_M$.

Algorithm 2 `Assign-Volume`(C, S_j).

if $C \notin \mathcal{C}_M$ **then**
  Let $i > 0$ s.t. $|C| \in [(1+\frac{\varepsilon}{4})^{i-1}, (1+\frac{\varepsilon}{4})^{i})$.
  $A(C) \leftarrow 0$.
  if $A_{i0} \leq |C| < A_{i1}$ **then**
    $A(C) \leftarrow A_{i2}$.
  else if $A_{i1} \leq |C| < A_{i2}$ **then**
    $A(C) \leftarrow A_{i3}$.
  else if $A_{i2} \leq |C| < A_{i3}$ **then**
    $A(C) \leftarrow A_{i,4}$.
  else
    $A(C) \leftarrow A_{i+1,1}$.
  $A(S_j) \leftarrow A(S_j) + A(C)$, $R_j \leftarrow R_j - A(C)$.
else ▷ If C is marked.
  Let $i \geq 0$ s.t. $|C| \in [Dk(1+\frac{\varepsilon}{4})^{i-1} - \frac{\varepsilon^2 k}{100}, Dk)$.
  $A(C) \leftarrow A_{i2}$.

If all clusters are marked, i.e. $\mathcal{S}_M = \mathcal{S}_A$ we add an arbitrary cluster in $\mathcal{S} \backslash \mathcal{S}_A$ to $\mathcal{S}_A$ and assign volume for C.

Algorithm 3 `Assign-Small`(C).

is-assigned=false.
for all $S_j \in \mathcal{S}_A \backslash \mathcal{S}_M$ **do**
  if $R_j \geq (1+\frac{\varepsilon}{4})|C|$ **then**
    `Assign-Volume`(C, S_j).
    is-assigned=true.
    break
  else
    $\mathcal{S}_M \leftarrow \mathcal{S}_M \cup \{S_j\}$.
if is-assigned=false **then**
  Pick an arbitrary cluster $S_j \in \mathcal{S} \backslash \mathcal{S}_A$.
  $\mathcal{S}_A \leftarrow \mathcal{S}_A \cup \{S_j\}$.
  `Assign-Volume`(C, S_j).
  $\mathcal{S}_S \leftarrow \mathcal{S}_S \cup \{S_j\}$.

Subroutine `Handle-Insertion`. The subroutine `Handle-Insertion` is called whenever a request r_t at time t corresponds to a vertex insertion v. Subroutine `Assign-Small` is invoked to assign the singleton component $\{v\}$.

Algorithm 4 `Handle-Insertion`(v).

$C \leftarrow \{v\}$.
$\mathcal{C}_S \leftarrow \mathcal{C}_S \cup \{C\}, \mathcal{C} \leftarrow \mathcal{C} \cup \{C\}$.
`Assign-Small`(C).

Subroutine `Unassign-Volume`. This subroutine takes as input a component C which is currently assigned to a cluster S_i. The quantity $A(S_i)$ is decremented by $A(C)$ and $A(C)$ is set to 0. If $S_i \in \mathcal{S}_M$ and $R_i \geq \frac{\varepsilon k}{3}$, S_i is unmarked. The reason why S_i is unmarked is that if $R_i \geq \frac{\varepsilon k}{2}$, a small component can be assigned volume on S_i, since the assigned volume for a small component is bounded by $\frac{\varepsilon k}{4}(1+\frac{\varepsilon}{4}) < \frac{\varepsilon k}{3}$ by Invariant 1. The pseudo-code is given as follows.

Algorithm 5 `Unassign-Volume`(C, S_i).

1: $A(S_i) \leftarrow A(S_i) - A(C)$, $R_i \leftarrow R_i + A(C)$.
2: $A(C) \leftarrow 0$.
3: **if** $S_i \in \mathcal{S}_M$ and $R_i \geq \frac{\varepsilon k}{2}$ **then**
4: $\quad \mathcal{S}_M \leftarrow \mathcal{S}_M \backslash \{S_i\}$.

Subroutine `Handle-Unmarked`. This subroutine takes as input a newly unmarked cluster S_i. If the assigned volume $A(S_i) = 0$, then S_i is removed from the sets $\mathcal{S}_A$ and $\mathcal{S}_S$. If $|\mathcal{S}_S| > 0$, an arbitrary cluster $S \in \mathcal{S}_S$ is chosen and unmarked.

Else, if $A(S_i) > 0$, Invariant 3 is restored as follows. Note that if $S_i \in \mathcal{S}_L$ and $\mathcal{S}_S = \emptyset$ or $S_i \in \mathcal{S}_S$ and $|\mathcal{S}_S| = 1$, nothing needs to be done since Invariant 3 is already maintained. Else, let $S \in \mathcal{S}_S$ denote an unmarked cluster where $S \neq S_i$. While the residual volume of S_i is sufficient (and S_i is unmarked and $|\mathcal{S}_S| > 0$) small components from S are migrated to S_i. If the residual volume of S_i is insufficient, S_i is marked and added to $\mathcal{S}_M$. If at any point $A(S) = 0$, S is removed from $\mathcal{S}_A$ and $\mathcal{S}_S$. Thereafter, if $S_i \in \mathcal{S}_L$, and $|\mathcal{S}_S| > 0$, an arbitrary cluster S in $\mathcal{S}_S$ is unmarked. If $S_i \in \mathcal{S}_S$, and $|\mathcal{S}_S| > 1$, an arbitrary cluster in $\mathcal{S}_S$ is unmarked. The procedure continues while the residual volume of S_i is sufficient. On the other hand, if S_i is the only cluster in $\mathcal{S}_S$ it is unmarked and the process terminates.

Subroutine `Reassign-Large`. We give a subroutine `Reassign-Large` which is invoked whenever the number of large components σ_i of any class $i \in [c_l]$ change. In `Reassign-Large`, the subroutine `Assign-Signatures` is invoked which returns the sets $\mathcal{S}_N, \mathcal{S}_O$ and $\mathcal{S}_I$ of clusters (see Section 2.2).

Let $\mathcal{C}'_S$ denote the set of all small components assigned to clusters in $\mathcal{S}_O \cup \mathcal{S}_N$. Small components currently assigned to clusters in $\mathcal{S}_O \cup \mathcal{S}_N$ are unassigned. Clusters in $\mathcal{S}_O \cup \mathcal{S}_N$ are unmarked and the set of active clusters $\mathcal{S}_A$ is updated by removing clusters in $\mathcal{S}_O$ and adding clusters in $\mathcal{S}_N$.

Next, let $\mathcal{C}'_L$ denote the set of large components which are not assigned to any cluster in $\mathcal{C}_I$. For all clusters $S \in \mathcal{S}_N$, let T_s be the signature assigned to cluster S. For all non-zero T_{sa} where $a \in [c_l]$, a class a large component C is chosen from $\mathcal{C}'_L$, and `Assign-Volume`(S, C) is invoked, concluding the the assignment of large components.

For small components $C \in \mathcal{C}'_S$, `Assign-Small`(C) is invoked. Finally, for all unmarked clusters S in $\mathcal{S}_N \notin \mathcal{S}_M$, `Handle-Unmarked`$(S)$ is invoked. The pseudo code of `Reassign-Large` is deferred to the full version of the paper.

Subroutine `Handle-Deletion`. We present a subroutine `Handle-Deletion` which takes as input a vertex v to be deleted. We assume that Invariant 1 holds prior to the deletion request. Let C denote the component containing v prior to v's deletion s.t. $|C| \in [(1+\frac{\varepsilon}{4})^{i-1}, (1+\frac{\varepsilon}{4})^{i})$ and S_j denote the cluster to which C is assigned. We consider two cases.

Algorithm 6 `Handle-Unmarked`(S_i).

if $A(S_i) = 0$ **then**
  $\mathcal{S}_A \leftarrow \mathcal{S}_A \backslash \{S_i\}, \mathcal{S}_S \leftarrow \mathcal{S}_S \backslash \{S_i\}$.
  if $|\mathcal{S}_S| > 0$ **then** Pick an arbitrary cluster $S \in \mathcal{S}_S$, $\mathcal{S}_M \leftarrow \mathcal{S}_M \backslash \{S\}$.
else
  if ($S_i \in \mathcal{S}_L$ and $\mathcal{S}_S \neq \emptyset$) or ($S_i \in \mathcal{S}_S$ and $|\mathcal{S}_S| > 1$) **then**
    Let $S \neq S_i$ be an unmarked cluster in $\mathcal{S}_S$.
    while $S_i \notin \mathcal{S}_M$ and $\mathcal{S}_S \neq \emptyset$ **do**
      Let C be an arbitrary small component assigned to S.
      if $R_i \geq (1 + \frac{\varepsilon}{4})|C|$ **then**
        `Assign-Volume`(C, S_i).
        `Unassign-Volume`(C, S).
      else
        $\mathcal{S}_M \leftarrow \mathcal{S}_M \cup \{S_i\}$. ▷ S_i is marked
      if $A(S) = 0$ **then**
        $\mathcal{S}_A \leftarrow \mathcal{S}_A \backslash \{S\}$, $\mathcal{S}_S \leftarrow \mathcal{S}_S \backslash \{S\}$. ▷ S is made inactive.
        if $S_i \in \mathcal{S}_L$ and $|\mathcal{S}_S| > 0$ **then**
          Let S be an arbitrary cluster in $\mathcal{S}_S$.
          $\mathcal{S}_M \leftarrow \mathcal{S}_M \backslash \{\mathcal{S}\}$. ▷ S is unmarked.
        if $S_i \in \mathcal{S}_S$ **then**
          if $|\mathcal{S}_S| > 1$ **then**
            Let $S \neq S_i$ be an arbitrary cluster in $\mathcal{S}_S$.
            $\mathcal{S}_M \leftarrow \mathcal{S}_M \backslash \{S\}$. ▷ S is unmarked.
          else
            break. ▷ S_i is the only unmarked cluster in $\mathcal{S}_S$ in this case.

1. C is small and unmarked: Vertex v is removed from C. There are five sub-cases to consider.
   a. If $A(C) = A_{i+1,2}$ and $|C| < A_{i3}$, then $A(C)$ is set to $A_{i+1,1}$.
   b. If $A(C) = A_{i+1,1}$ and $|C| < A_{i2}$, then $A(C)$ is set to A_{i4}.
   c. If $A(C) = A_{i4}$ and $|C| < A_{i1}$, then $A(C)$ is set to A_{i3}.
   d. If $A(C) = A_{i3}$ and $|C| < A_{i0} = (1 + \frac{\varepsilon}{4})^{i-1}$, then C has become a class $i-1$ component. $A(C)$ is set to A_{i2}.
   e. If $C = \emptyset$, $A(C)$ is set to 0.

   In all of the above cases, $A(S_j)$ and R_j are updated. If $S_j \in \mathcal{S}_M$ and $R_j \geq \frac{\varepsilon k}{2}$, S_j is removed from $\mathcal{S}_M$ and `Handle-Unmarked`(S_j) is invoked.
2. C is large or marked: We first remove v from C. We consider the following cases.
   a. C is large and unmarked (i.e. $C \in \mathcal{C}_L \backslash \mathcal{C}_M$): Our algorithm considers the following cases.
      i. If $A(C) = A_{i+1,2}$ and $|C| < A_{i3}$, then $A(C)$ is set to $A_{i+1,1}$.
      ii. If $A(C) = A_{i+1,1}$ and $|C| < A_{i2}$, then $A(C)$ is set to A_{i4}.
      iii. If $A(C) = A_{i4}$ and $|C| < A_{i1}$, then $A(C)$ is set to A_{i3}.
      iv. If $|C| < A_{i0}$, C is marked. If $A(C) = A_{i3}$ then $A(C)$ is set to A_{i2}.

      In the above cases, if $S_j \in \mathcal{S}_M$ and $R_j \geq \frac{\varepsilon k}{2}$, S_j is removed from $\mathcal{S}_M$ and `Handle-Unmarked`(S_j) is invoked.

b. C is marked: There are two cases to consider depending on whether C is large or small (after the deletion of v).

i. Let C be a marked large component of class c where $c = i - c_s$, such that $c \in [c_l - 1]$. If $|C| \geq Dk(1+\frac{\varepsilon}{4})^c - \frac{\varepsilon^2 k}{100}$, there is nothing to be done. Else, if $|C| < Dk(1+\frac{\varepsilon}{4})^c - \frac{\varepsilon^2 k}{100}$, C is unmarked. The counter σ_{c+1} is decremented since C is now considered a class c component, and σ_c is incremented. Then, the subroutine `Reassign-Large` is invoked.

ii. Let C be a marked small component of class c_s. If $|C| \geq Dk - \frac{\varepsilon^2 k}{100}$, nothing is done. Else, if $|C| < Dk - \frac{\varepsilon^2 k}{100}$, C is first unmarked. Then, C is removed from $\mathcal{C}_L$ and added to $\mathcal{C}_S$. The counter σ_1 is decremented. Then, the subroutine `Reassign-Large` is invoked.

The pseudo-code of `Handle-Deletion` is deferred to the full version of the paper.

`Merge-Components`. The subroutine `Merge-Components` is invoked by our algorithm to handle a merge request (u, v). Let C_i, C_j denote the components of u and v respectively. If $C_i = C_j$, nothing needs to be done. Otherwise, w.l.o.g., let $|C_j| \leq |C_i|$ and S_i and S_j denote the clusters on which C_i and C_j are currently assigned. Let $C_m = C_i \cup C_j$.

1. C_m is small: The subroutine `Handle-Small` is called and considers two cases.

a. C_i is marked: We note that C_j cannot be a marked component since this would contradict that C_m is small. This is because if C_j is marked, then $|C_m| \geq 2(Dk - \frac{\varepsilon^2 k}{100}) = Dk + Dk - \frac{\varepsilon^2 k}{50} > Dk$ since $Dk - \frac{\varepsilon^2 k}{50} > \frac{\varepsilon k}{5} - \frac{\varepsilon k}{50} > 0$. Thus, the only possibility is that C_i is marked. Since any marked small component must have volume at least $Dk - \frac{\varepsilon^2 k}{100}$, and C_m is small it follows that $|C_m| \leq Dk$. In this case we note that $A(C_i) = A_{i2} \geq Dk$ so that $A(C_m)$ can be set to $A(C_i)$. C_i is removed from $\mathcal{C}_M$ and C_m is added to $\mathcal{C}_M$. If $|C_m| < Dk$, C_m is added to $\mathcal{C}_M$. As such, C_m is still treated by our algorithm as a large marked component. If $S_i = S_j$ nothing needs to be done. Else, vertices in C_j are migrated to S_i, and `Unassign-Volume`(C_j, S_j) is invoked. Subroutine `Handle-Unmarked`(S_j) is invoked to handle the case when S_j becomes unmarked.

b. C_i and C_j are not marked: In this case, there are two cases to consider.

i. If $A(C_i) \geq |C_m|$, we set $A(C_m)$ to $A(C_i)$ and $A(C_i), A(C_j)$ to 0. If $S_j = S_i$, nothing needs to be done. If $S_j \neq S_i$, vertices in C_j are migrated to S_i, and `Unassign-Volume`(C_j) is invoked. Subroutine `Handle-Unmarked`(S_j) is invoked to handle the case if S_j becomes unmarked.

ii. If $A(C_i) < |C_m|$, we invoke `Unassign-Volume`(C_i, S_i) and `Unassign-Volume`(C_j, S_j). If $R_i \geq (1 + \frac{\varepsilon}{4})|C_m|$, we call `Assign-Volume`$(C_m, S_i)$. If $S_i \neq S_j$, we migrate vertices in C_j to S_i. We invoke `Handle-Unmarked`(S_j) and `Handle-Unmarked`(S_i).
On the other hand, if $R_i < (1 + \frac{\varepsilon}{4})|C|$, S_i is marked and `Assign-Small`(C_m) is invoked.

2. C_m is large: The subroutine `Handle-Large` is invoked which adds C_m to the set of large components.
Let $m' \in [c_l]$ be an integer such that $|C_m| \in [Dk(1 + \frac{\varepsilon}{4})^{m'-1}, Dk(1 + \frac{\varepsilon}{4})^{m'})$. If C_i is marked, i.e. $C_i \in \mathcal{C}_M$, let $i' \coloneqq a + 1$ be the integer where a is an integer such that $|C_i| \in [Dk(1 + \frac{\varepsilon}{4})^a - \frac{\varepsilon^2 k}{100}, Dk(1 + \frac{\varepsilon}{4})^a)$, else we let $i' \coloneqq a$ be the integer where a is an integer such that $|C_i| \in [Dk(1 + \frac{\varepsilon}{4})^{a-1}, Dk(1 + \frac{\varepsilon}{4})^a)$. We consider the following cases.

a. C_j is large or C_j is marked or $m' > i'$: If C_j is marked, we let $j' \coloneqq a + 1$ be the integer where a is an integer such that $|C_j| \in [Dk(1 + \frac{\varepsilon}{4})^a - \frac{\varepsilon^2 k}{100}, Dk(1 + \frac{\varepsilon}{4})^a)$. Else if C_j is unmarked, we let $j' \coloneqq a$ be the integer where a is an integer such that

$|C_j| \in [Dk(1+\frac{\varepsilon}{4})^{a-1}, Dk(1+\frac{\varepsilon}{4})^{a})$. We decrement $\sigma_{j'}$ and $\sigma_{i'}$ by 1, and increment $\sigma_{m'}$ by 1. We remove C_i, C_j from $\mathcal{C}_M$ to make sure that the set of marked components is updated. Thereafter, we update the component sets $\mathcal{C}_S, \mathcal{C}_L$ and call subroutine `Reassign-Large` to handle the assignment of large components.

b. C_j is small and C_j is unmarked and $m' \leq i'$: In this case, we note that $\sigma_{i'}, \sigma_{m'}$ do not change. Thus, the ILP is not solved. We assign volume for the merged component as follows.

In the case when $A(C_i) \geq A(C_m)$, vertices in C_j are migrated to S_i if $S_i \neq S_j$. We invoke `Unassign-Volume`(C_j, S_j), and `Handle-Unmarked`(S_j). Else, nothing needs to be done. The assigned volumes $A(C_i), A(C_j)$ are set to 0. If C_i is marked and $|C_m| < Dk(1+\frac{\varepsilon k}{4})^{i'-1}$, then C_i is removed from $\mathcal{C}_M$ and C_m is added to $\mathcal{C}_M$. We note that in this case, $A(C_i) = A_{i'2} \geq A(C_m)$.

If $A(C_i) < |C_m|$, then we invoke `Unassign-Volume`(C_i, S_i) and `Unassign-Volume`(C_j, S_j).

If $R_i \geq (1+\frac{\varepsilon}{4})|C_m|$, we assign volume for C_m on S_i by invoking `Assign-Volume`(C_m, S_i). If $A(S_j) = 0$, we call `Handle-Unmarked`(S_j). Next, we call `Handle-Unmarked`(S_i). In the case when $A(S_j) \neq 0$ (and $S_j \in \mathcal{S}_A$ as a result), we invoke `Handle-Unmarked`(S_j).

If $R_i < (1+\frac{\varepsilon}{4})|C_m|$, we do the following. While $R_i < (1+\frac{\varepsilon}{4})|C_m|$, we call `Unassign-Volume`$(C, S_i)$ where C is an arbitrary small component assigned on S_i, and add it to a set $\mathcal{C}'_S$. Once $R_i \geq (1+\frac{\varepsilon}{4})|C_m|$, we invoke `Assign-Volume`$(C_m, S_i)$ and migrate vertices in C_j to S_i if needed (in the case when $S_i \neq S_j$) together with a call to `Unassign-Volume`(C_j, S_j). Next, we assign components in $\mathcal{C}'_S$ to S_i as long as there is sufficient volume and S_i is not marked. If all components in $\mathcal{C}'_S$ have been assigned and S_i is unmarked, we call `Handle-Unmarked`(S_i).

If S_j is unmarked, we invoke `Handle-Unmarked`(S_j). All remaining unassigned components C in $\mathcal{C}'_S$ (if any) are assigned by invoking `Assign-Small`(C). Finally, the component sets $\mathcal{C}_S, \mathcal{C}_L$ are updated.

The pseudo-code of subroutines `Merge-Components`, `Handle-Small` and `Handle-Large` are deferred to the full version of the paper.

We present the pseudo-code of our final online algorithm `OBA` that utilizes the aforementioned subroutines as follows.

Algorithm 7 `OBA`.

$\mathcal{C}_S, \mathcal{C}_L, \mathcal{C}_M, \mathcal{C}_C \leftarrow \emptyset$.
for all $S_i \in \mathcal{S}$ **do** $A(S_i) \leftarrow 0, R_i = (1+\varepsilon)k$
$\mathcal{S}_S, \mathcal{S}_L, \mathcal{S}_M, \mathcal{S}_A \leftarrow \emptyset$.
for r_t at time t **do**
    if r_t is an insertion request for vertex v **then**
        `Handle-Insertion`(v).
    else if r_t is a deletion request for vertex v **then**
        `Handle-Deletion`(v).
    else
        Let r_t be a merge request between vertices (u, v).
        `Merge-Components`(u, v).

3 Analysis of OBA

In this section, we give a proof sketch towards $O(\log k)$-competitiveness of our algorithm. Technical details of our proof, together with proofs of correctness and resource competitiveness are deferred to the full version of the paper.

Our analysis is intricate and relies on various charging arguments. At a high level, we show that a vertex can be assigned sufficient credit on insertion such that the total amount of credit on it at any point in time is sufficient to pay for the cost charged against it. The total credit on a vertex is the sum of its *direct* and *indirect* credits. *Direct* credit is a one-time credit assigned to v on its insertion. *Indirect* credit on v is credit passed on to it by other vertices which get deleted from v's component.

Warm up: An insertions-only case. Let us first consider a (finite) request sequence σ which only consists of insertion and merge requests. For this sequence, the assigned volume of any component containing a vertex v monotonically increases over time, and no components are ever marked. We show that it suffices to charge each vertex an amount $O(\log k + \frac{f(\varepsilon)}{\varepsilon^3})$ on its insertion. On v's insertion, our algorithm assigns $\{v\}$ to an active cluster in $\mathcal{S}_A$, or makes a new cluster active and assigns $\{v\}$ to it. On a merge request between components C_i and C_j which are merged into a *small* component C_m, where w.l.o.g., $|C_i| \geq |C_j|$ our algorithm always migrates vertices in C_j to C_i (in the case when $S_i \neq S_j$) as long as the cluster S_i containing C_i has sufficient residual volume. The first key observation is that for such a merge, all vertices in C_j are now part of a component (i.e. C_m) with a larger component class. Thus, if $A(C_i) \geq |C_m|$ or $R_i \geq (1 + \frac{\varepsilon}{4})|C_m|$, only vertices in C_j are migrated. Potentially some other vertices may be migrated to S_j if S_j becomes unmarked after volume is un-assigned due to the migration of C_j (we ignore these costs for now). For any vertex v, and the component C containing v, note that C's component class increases monotonically over time due to merge requests. The number of component classes, which is $O(c_s + c_l) = O(\log k + \frac{1}{\varepsilon^2})$ bounds the number of times the component class can increase.

On the other hand, if R_i is insufficient to accommodate vertices in C_j (i.e. in the case when $A(C_i) < |C_m|$ and $R_i < (1 + \frac{\varepsilon}{4})k$), vertices in both C_i and C_j are migrated to another cluster. Note that in this case, $A(C_m) > A(C_i)$. Since there are at most $4(c_s + c_l) = O(\log k + \frac{1}{\varepsilon^2}) = O(\log k)$ possible values that assigned volumes can take, and assigned volume increases monotonically for any component, each vertex v can migrate only $O(\log k + \frac{1}{\varepsilon^2})$ times in this manner.

Crucially, we note that if the component C containing v becomes large (or C is large and its component class increases), a total of $O(f(\varepsilon)k)$ migration cost may be incurred, by Lemma 10. This migration cost can be incurred a total of $c_l = O(\frac{1}{\varepsilon^2})$ times. By definition, a C has size at least $Dk \geq \frac{\varepsilon k}{5}$; we charge the migration cost to all $\Omega(\varepsilon k)$ vertices in C. Thus, each vertex in C contributes at least $O(\frac{f(\varepsilon)}{\varepsilon})$ credit a total of $O(\frac{1}{\varepsilon^2})$ times. Therefore, a credit of $O(\frac{f(\varepsilon)}{\varepsilon^3})$ on a vertex is sufficient to pay for migrations of this type.

We now analyze the potential migration costs incurred when a component C migrates from one cluster (say S_i) to S_j, which may lead to a call to `Handle-Unmarked`(S_i) leading to components migrations to S_i. To account for these costs, a cluster credit $\Theta(|C|)$ is left on S_i when C migrates to S_j, and each vertex in C is charged an amount $O(1)$. As a result, each time the residual volume of a cluster increases, there is enough cluster credit to pay for successive migrations (as part of `Handle-Unmarked`). Note that in subroutine `Handle-Unmarked`, only small components are migrated from an unmarked cluster, and the total amount of migrated components is proportional to the increase in residual volume.

Thus, a vertex is charged for cluster credit each time its component is promoted to a higher class or its assigned volume increases. Since the number of choices for assigned volumes (and component classes) is $O(\log k + \frac{1}{\varepsilon^2})$, any vertex is charged $O(\log k + \frac{1}{\varepsilon^2})$ in this manner.

By the above, it follows that assigning a one-time direct credit of $O(\frac{1}{\varepsilon^3} f(\varepsilon) + \log k)$ on each vertex v on its insertion is sufficient. Since OPT is lower bounded by the number of vertex insertions, this yields $O(\log k)$ competitiveness.

The General Case. Let us consider the case when the request sequence consists of insertions, merges and deletions. In this case, the component class of any component C and its assigned volume $A(C)$ do not necessarily increase over time, in general. Thus, direct credit on v is insufficient, and our analysis crucially relies on *indirect* credit. Indirect credit is assigned to any vertex in the following manner: whenever a vertex u is deleted from v's component C, an amount of indirect credit is distributed to all remaining vertices in C. Some credit is also left as cluster credit on the cluster containing C on u's deletion. Cluster credit is used to pay for re-balancing and migrating components as in the previous case. The total indirect credit assigned to vertices in C, together with cluster credit that is left on u's deletion is charged against vertex u. On the other hand, the total credit assigned to a vertex is the sum of the direct credit and indirect credit distributed to it by other vertices. Let C denote the component containing v. We show that even in the case of deletions, it suffices to assign a direct credit of $O(\frac{1}{\varepsilon^3} f(\varepsilon) + \log k)$ to vertex v upon insertion, which is used to pay for:

1. the unit migration cost incurred each time C is promoted to a component class for the *first* time, or the assigned volume of C increases for the first time as in the previous case.
2. indirect credit assigned to vertices in C upon the deletion of v.
3. cluster credit assigned to the cluster containing C, upon v's deletion.
4. cluster credit assigned to the cluster containing C, before C is migrated to another cluster as a result of C being promoted to either a component class for the first time, or an increase in its assigned volume for the first time.

Consider a vertex v and let C denote an (unmarked) component containing v with assigned volume $A(C)$. Upon vertex deletions from C, $A(C)$ may decrease. Suppose $A(C) = A_{ij}$ for some $j \in \{1, 2, 3, 4\}$. Consider a set of deleted vertices C_D which lead to a reduction in C's assigned volume to $A_{i,j-1}$. For this to happen, $|C_D| \geq \frac{\varepsilon}{16}(1 + \frac{\varepsilon}{4})^{i-1}$. On the other hand, $|C| \leq (1 + \frac{j-1}{16}\varepsilon)(1 + \frac{\varepsilon}{4})^{i-1} \leq (1 + \frac{3\varepsilon}{16})(1 + \frac{\varepsilon}{4})^{i-1}$, after vertices in C_D are deleted. For each vertex u in C_D, we distribute a total indirect credit of $O(\frac{1}{\varepsilon^2} f(\varepsilon))$ uniformly to all remaining vertices in C. As a result, the total indirect credit gained by each remaining vertex in $C \backslash C_D$ is at least $O(|C_D| \cdot \frac{O(\frac{f(\varepsilon)}{\varepsilon^2})}{(1+\frac{3\varepsilon}{16})(1+\frac{\varepsilon}{4})^{i-1}}) = O(\frac{f(\varepsilon)}{\varepsilon})$ after all vertices in C_D are deleted. The *indirect* credit gained by any vertex v in $C \backslash C_D$ is used to pay for: i) a single *future* migration of a vertex $v \in C \backslash C_D$ which leads to an increase in either the assigned volume of v's component or the component class of v's component and ii) leave $O(1)$ cluster credit when the aforementioned migration happens.

Next, consider the time when a component C is marked, such that $|C| < Dk(1 + \frac{\varepsilon}{4})^{i-1}$ where $i \geq 1$. Once $|C| < Dk(1 + \frac{\varepsilon}{4})^{i-1} - \frac{\varepsilon^2 k}{100}$, this leads to either a call to `Reassign-Large` (if C is large) or C is treated as a small component thereafter (see `Handle-Deletion`). Since C has size at most k, this implies that at least $\frac{\varepsilon^2}{100} = \Omega(\varepsilon^2)$ fraction of vertices in C are deleted before either of these events happen. The total migration and reassignment cost incurred due to `Reassign-Large` is $O(f(\varepsilon)k)$ by Lemma 10. To pay for this cost, leave an indirect credit of $O(\frac{f(\varepsilon)}{\varepsilon})$ on each remaining vertex in C, and assign a cluster credit of $O(1)$, it suffices to charge each deleted vertex from C an amount $O(\frac{1}{\varepsilon^3} f(\varepsilon))$. The cluster credit is used to pay for the migration cost of vertices to S_i if S_i becomes unmarked. The amount charged against a deleted vertex u is paid by the direct credit assigned to u upon its insertion.

By the above, it follows that assigning a direct credit of $O(\log k + \frac{f(\varepsilon)}{\varepsilon^3})$ to a vertex upon its insertion is sufficient. Since OPT is lower bounded by the number of vertices inserted over all time, this yields $O(\log k)$ competitiveness for constant $\varepsilon > 0$.

4 OBADC with Predictions

In this section, we present our algorithm for OBADC which is augmented with machine learned predictions. The problem formulation and objectives are identical as in the standard un-augmented setup. At each time step t, a request r_t which corresponds to either a vertex insertion, deletion or a merge between two components is issued. At any given point in time, an online algorithm is required to maintain an assignment of components to clusters, while respecting the over-provisioned cluster capacity of $(1+\varepsilon)k$, such that all vertices in any component are assigned to the same cluster. In light of the $\Omega(\log k)$ lower bound on the competitiveness of any randomized algorithm, it is natural to ask whether this barrier can be circumvented given oracle access to machine-learned predictions to guide the assignment of components. To this end, we recap our prediction model.

Prediction Model. Let V_{t-1} denote the set of vertices at the beginning of time $t-1$. For a request r_t at time t which corresponds to an insertion of vertex v, we obtain:

1. with probability $\eta > 0$, a predicted set $P(v)$ of all vertices in V_{t-1} which will be in the same component as v.
2. with probability $1-\eta$, a *null* prediction, i.e. $P(v) = \emptyset$. Thus no information is revealed on the insertion of v.

We present an algorithm `Predicted-OBA` which is $O(1)$-consistent and $\min\{\log \frac{1}{\eta}, \log k\}$-robust. This algorithm is similar in many ways to our $O(\log k)$ competitive algorithm `OBA`. We utilize identical data structures, definitions and subroutines as detailed in Section 2. We also maintain Invariants 1 through 4 as in Section 2.

Our algorithm `Predicted-OBA` is as follows.

Algorithm 8 `Predicted-OBA`.

$\mathcal{C}_S, \mathcal{C}_L, \mathcal{C}_M, \mathcal{C}_C \leftarrow \emptyset$.
for all $S_i \in \mathcal{S}$ **do** $A(S_i) \leftarrow 0, R_i = (1+\varepsilon)k$
$\mathcal{S}_S, \mathcal{S}_L, \mathcal{S}_M, \mathcal{S}_A \leftarrow \emptyset$.
for r_t at time t **do**
    if r_t is an insertion request for vertex v **then**
        `Predicted-Insertion`$(v, P(v))$.
    else if r_t is a deletion request for vertex v **then**
        `Handle-Deletion`(v).
    else
        Let r_t be a merge request between components C_i and C_j.
        `Merge-Components`(C_i, C_j).

We give an additional subroutine, `Predicted-Insertion` as follows.

Subroutine `Predicted-Insertion`. This subroutine takes as input a vertex v inserted at time t and a predicted set $P(v)$ consisting of vertices at time t that will be in the same component as v. If $P(v) = \emptyset$, the subroutine `Handle-Insertion`(v) is invoked. Else, a set σ_I of merge requests is created where $\sigma_I = \{(u, v) |\ u \in P(v)\}$. For each merge request in σ_I, the subroutine `Merge-Components`(u, v) is invoked.

The following lemma bounds the competitive ratio of Algorithm `Predicted-OBA`.

▶ **Lemma 11.** *Algorithm `Predicted-OBA` is $O(\min\{\log(1/\eta), \log k\})$-competitive.*

We build towards a proof of Lemma 11 in the following. We first analyze the probability of a merge request being incident to vertices in distinct components C_i, C_j as a function of component sizes.

▶ **Lemma 12.** *Let r_t be a merge request between components C and C' at time t, such that $s = |C| \leq |C'|$ without loss of generality. Then, the probability that $C' \neq C$ is bounded by $e^{-\eta s}$.*

Proof. We note that if $C \neq C'$, then for at least $s = |C|$ insertions of vertices w, $P(w) = \emptyset$. Since the probability that $P(w) = \emptyset$ for any w is $(1-\eta)$, it follows that for s vertex insertions, the probability that all predicted sets are empty is bounded by $(1-\eta)^s \leq e^{-\eta s}$. ◀

We define monotonic merges as follows.

▶ **Definition 13** (Monotonic Merge)**.** *Let C be the component containing vertex v at any point in time throughout v's lifetime. Let t_1 and t_2 be time-steps s.t. v is inserted at time t_1 and deleted at time t_2. A merge request involving components C and C' at time $t \in [t_1, t_2)$ is said to be a monotonic merge for v if $|C| \leq |C'|$ and one of the following holds:*

1. *The assigned volume of C is set to A_{ij} by our algorithm where $i \in [c_s + c_l], j \in \{1, 2, 3, 4\}$ for the first time during time interval $[t_1, t_2)$.*
2. *The component class of C is set to $i \in [c_s + c_l]$ by our algorithm for the first time during time interval $[t_1, t_2)$.*

A merge request for v is non-monotonic if it is not monotonic.

▶ **Lemma 14.** *The expected number of monotonic merge requests for any vertex v including merge requests created by Subroutine `Predicted-Insertion` is bounded by $O(\log(\frac{1}{\eta}))$.*

Proof. Let C denote the component of v, such that v is inserted at time t_1 and deleted at time t_2. Let $d = (1 + \frac{\varepsilon}{4})$, so that a class i component has volume in $[d^{i-1}, d^i)$. By Lemma 12, the probability that a merge request involving C where C is the smaller of the two components and belongs to class i, is bounded by $e^{-\eta \cdot d^{i-1}}$. Thus, the expected number of monotonic merge requests for v is bounded by

$$\sum_{i=0}^{\log k} e^{-d^i \eta} = \sum_{i=0}^{\log(1/\eta)} e^{-d^i \eta} + \sum_{i=\log(1/\eta)+1}^{\log k} e^{-d^i \eta} \leq \log(1/\eta) + O(1) = O(\log(1/\eta)).$$ ◀

Proof Sketch of Lemma 11. Let us recall the analysis of $O(\log k)$-competitiveness of our algorithm `OBA`. We argued that it suffices to leave a direct credit of $O(\log k + \frac{f(\varepsilon)}{\varepsilon^3})$ on any vertex v upon its insertion, which is sufficient to pay for: i) migration costs and cluster credits charged against v for both monotonic and non-monotonic merges and, ii) cluster and indirect credits charged against v upon its deletion. The $O(\log k)$ term comes from the fact a vertex v can be part of monotonic merges for a total of $c_s + c_l = O(\log k)$ times, and is charged $O(\log k)$ times for merges while C is small. On the other hand, for a non-monotonic merge, v is charged an amount $O(1)$ to pay for its migration cost and leave cluster credit if the residual volume of the cluster C is assigned to, is no longer sufficient to accommodate an increase in $A(C)$. Since the number of choices for assigned volumes is $O(c_s + c_l) = O(\log k + \frac{1}{\varepsilon^2})$, the total cost charged against v for non-monotonic merges is $O(\log k + \frac{1}{\varepsilon^2})$.

We modify our charging argument for non-monotonic merges as follows. Ignoring the reassignment of components that takes place as a result of subroutine `Handle-Unmarked` for the sake of clarity, we crucially observe that whenever the residual volume of C's cluster (denoted by S_i) decreases, it is due to either: i) vertices in a smaller component involved in a merge request migrate to S_i, or ii) new vertices are inserted to S_i's cluster. We modify our charging argument such that the migrated or inserted vertex in the above cases leaves $O(1)$ extra cluster credit on S_i. Thus, whenever the residual volume of S_i decreases by $\Omega(k)$, these extra cluster credits can be used to pay for non-monotonic merges involving vertices already assigned on S_i. As a result, we focus on the cost charged for monotonic merges in the remainder of the analysis.

By Lemma 14, it follows that the expected number of monotonic merges for a vertex is bounded by $O(\log(\frac{1}{\eta}))$. By our analysis of `OBA`, it follows that the expected cost charged to a vertex for monotonic merges is $O(\frac{f(\varepsilon)}{\varepsilon})$. Thus, an expected credit of $O(\log(1/\eta)\frac{f(\varepsilon)}{\varepsilon})$ is sufficient for monotonic merges. On the other hand, when a vertex v is deleted, we charge an amount $O(\frac{f(\varepsilon)}{\varepsilon^3})$ to it which is assigned as indirect credit and cluster credit. As a result, it follows that a total direct credit of $O(\log(1/\eta)\frac{f(\varepsilon)}{\varepsilon} + \frac{f(\varepsilon)}{\varepsilon^3})$ in expectation, is sufficient to pay for all costs charged against v throughout its lifetime.

This yields $O(\log(1/\eta))$-competitiveness. On the other hand, as $\eta \to 0$, the behavior of algorithm `Predicted-OBA` resembles the behavior of algorithm `OBA`, and thus, the competitiveness of `Predicted-OBA` is bounded by $O(\log k)$ in the worst-case. This yields $O(\min(\log(1/\eta), \log k))$-competitiveness. The following lemma establishes $O(1)$ competitiveness when $\eta = 1$. The proof is deferred to the full version of the paper.

▶ **Lemma 15.** *Algorithm `Predicted-OBA` is $O(1)$ consistent.*

Theorem 2 follows from Lemmas 11 and 15.

5 Future Work

Our model for OBADC is restrictive in the sense that component sizes decrease only as a result of vertex deletions; in particular, our model does not allow for splitting of existing components. We believe that such a model will be much more challenging to handle; indeed, it could be considered as a generalization of the general model of OBGR, for which there is a significant gap between the upper and lower bounds for the best competitive ratio achievable.

In this work, we introduced the OBADC and studied it in the context of machine-learned predictions. It would be interesting to extend our prediction model to the case when predictions are always available but may be erroneous. Other prediction models for online balanced allocation problems would be worth exploring.

References

1 Khaled Almi'ani, Young Choon Lee, and Bernard Mans. Resource demand aware scheduling for workflows in clouds. In *2017 IEEE 16th International Symposium on Network Computing and Applications (NCA)*, pages 1–5. IEEE, 2017.

2 Chen Avin, Marcin Bienkowski, Andreas Loukas, Maciej Pacut, and Stefan Schmid. Dynamic balanced graph partitioning. *SIAM Journal on Discrete Mathematics*, 34(3):1791–1812, 2020. doi:10.1137/17M1158513.

3 Yossi Azar, Chay Machluf, Boaz Patt-Shamir, and Noam Touitou. Competitive Vertex Recoloring. In Mikołaj Bojańczyk, Emanuela Merelli, and David P. Woodruff, editors, *49th International Colloquium on Automata, Languages, and Programming (ICALP 2022)*, volume 229 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:20, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ICALP.2022.13.

4 Marcin Bienkowski, Martin Böhm, Martin Koutecký, Thomas Rothvoß, Jiří Sgall, and Pavel Veselý. Improved analysis of online balanced clustering. In *Approximation and Online Algorithms: 19th International Workshop, WAOA 2021, Lisbon, Portugal, September 6–10, 2021, Revised Selected Papers*, pages 224–233, Berlin, Heidelberg, 2021. Springer-Verlag. doi:10.1007/978-3-030-92702-8_14.

5 Jan van den Brand, Sebastian Forster, Yasamin Nazari, and Adam Polak. On dynamic graph algorithms with predictions. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3534–3557. SIAM, 2024. doi:10.1137/1.9781611977912.126.

6 Prasad Calyam, Sripriya Seetharam, Baisravan Homchaudhuri, and Manish Kumar. Resource defragmentation using market-driven allocation in virtual desktop clouds. In *2015 IEEE International Conference on Cloud Engineering*, pages 246–255. IEEE, 2015. doi:10.1109/IC2E.2015.37.

7 Jiuxin Cao, Zhuo Ma, Jue Xie, Xiangying Zhu, Fang Dong, and Bo Liu. Towards tenant demand-aware bandwidth allocation strategy in cloud datacenter. *Future Generation Computer Systems*, 105:904–915, 2020. doi:10.1016/J.FUTURE.2017.06.005.

8 Vincent Cohen-Addad, Tommaso d'Orsi, Anupam Gupta, Euiwoong Lee, and Debmalya Panigrahi. Max-cut with *epsilon*-accurate predictions. *arXiv preprint arXiv:2402.18263*, 2024.

9 Hancong Duan, Chao Chen, Geyong Min, and Yu Wu. Energy-aware scheduling of virtual machines in heterogeneous cloud computing systems. *Future Generation Computer Systems*, 74:142–150, 2017. doi:10.1016/J.FUTURE.2016.02.016.

10 Tobias Forner, Harald Räcke, and Stefan Schmid. Online balanced repartitioning of dynamic communication patterns in polynomial time. In *Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pages 40–54, 2021. doi:10.1137/1.9781611976489.4.

11 Anupam Gupta, Debmalya Panigrahi, Bernardo Subercaseaux, and Kevin Sun. Augmenting online algorithms with ε-accurate predictions. *Advances in neural information processing systems*, 35:2115–2127, 2022.

12 Sijin He, Li Guo, and Yike Guo. Real time elastic cloud management for limited resources. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 622–629. IEEE, 2011. doi:10.1109/CLOUD.2011.47.

13 Monika Henzinger, Stefan Neumann, Harald Räcke, and Stefan Schmid. *Tight Bounds for Online Graph Partitioning*, pages 2799–2818. Society for Industrial and Applied Mathematics, USA, 2021.

14 Monika Henzinger, Stefan Neumann, and Stefan Schmid. Efficient distributed workload (re-)embedding. In *Abstracts of the 2019 SIGMETRICS/Performance Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '19, pages 43–44, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3309697.3331503.

15 Thodoris Lykouris and Sergei Vassilvitskii. Competitive caching with machine learned advice. *Journal of the ACM (JACM)*, 68(4):1–25, 2021. doi:10.1145/3447579.

16 Zoltán Ádám Mann. Allocation of virtual machines in cloud data centers—a survey of problem models and optimization algorithms. *Acm Computing Surveys (CSUR)*, 48(1):1–34, 2015. doi:10.1145/2797211.

17 Suhib Bani Melhem, Anjali Agarwal, Nishith Goel, and Marzia Zaman. Markov prediction model for host load detection and vm placement in live migration. *IEEE Access*, 6:7190–7205, 2017. doi:10.1109/ACCESS.2017.2785280.

18 Michael Mitzenmacher and Sergei Vassilvitskii. Algorithms with predictions. *Communications of the ACM*, 65(7):33–35, 2022. doi:10.1145/3528087.

19 Saloua El Motaki, Ali Yahyaouy, and Hamid Gualous. A prediction-based model for virtual machine live migration monitoring in a cloud datacenter. *Computing*, 103(11):2711–2735, 2021. doi:10.1007/S00607-021-00981-3.

20 Maciej Pacut, Mahmoud Parham, and Stefan Schmid. Optimal online balanced partitioning. In *INFOCOM 2021*, 2021.

21 Cynthia A. Phillips, Cliff Stein, Eric Torng, and Joel Wein. Optimal time-critical scheduling via resource augmentation (extended abstract). In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 140–149, New York, NY, USA, 1997. Association for Computing Machinery. doi:10.1145/258533.258570.

22 Rajmohan Rajaraman and Omer Wasim. Improved bounds for online balanced graph repartitioning. In *30th Annual European Symposium on Algorithms (ESA 2022)*, 2022. doi:10.4230/LIPIcs.ESA.2022.83.

23 Rajmohan Rajaraman and Omer Wasim. Competitive capacitated online recoloring. In *32nd Annual European Symposium on Algorithms (ESA 2024)*, 2024. doi:10.4230/LIPIcs.ESA.2024.95.

24 Dhruv Rohatgi. Near-optimal bounds for online caching with machine learned advice. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1834–1845. SIAM, 2020. doi:10.1137/1.9781611975994.112.

25 Tim Roughgarden. *Beyond the worst-case analysis of algorithms*. Cambridge University Press, 2021.

26 A. Schrijver. Theory of linear and integer programming. In *Wiley-Interscience series in discrete mathematics and optimization*, 1999.

27 Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998.

28 Jaspreet Singh and Navpreet Kaur Walia. A comprehensive review of cloud computing virtual machine consolidation. *IEEE Access*, 11:106190–106209, 2023. doi:10.1109/ACCESS.2023.3314613.

29 Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, February 1985. doi:10.1145/2786.2793.

30 Mukundan Sridharan, Prasad Calyam, Aishwarya Venkataraman, and Alex Berryman. Defragmentation of resources in virtual desktop clouds for cost-aware utility-optimal allocation. In *2011 Fourth IEEE International Conference on Utility and Cloud Computing*, pages 253–260. IEEE, 2011. doi:10.1109/UCC.2011.41.

31 Ruitao Xie, Xiaohua Jia, Kan Yang, and Bo Zhang. Energy saving virtual machine allocation in cloud computing. In *2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops*, pages 132–137. IEEE, 2013. doi:10.1109/ICDCSW.2013.37.

32 Hiroki Yanagisawa, Takayuki Osogami, and Rudy Raymond. Dependable virtual machine allocation. In *2013 Proceedings IEEE INFOCOM*, pages 629–637. IEEE, 2013. doi:10.1109/INFCOM.2013.6566848.

33 N. Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 11(6):525–541, June 1994. doi:10.1007/BF01189992.

34 Neal E. Young. On-line file caching. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '98, pages 82–86, USA, 1998. Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=314613.314658.