Competitive Clustering of Stochastic Communication Patterns on a Ring[⋆]

Chen Avin[1], Louis Cohen[2], Mahmoud Parham[3], and Stefan Schmid[4]

[1] Ben-Gurion University of the Negev, Israel
[2] Ecole Normale Superieure Paris Saclay, France
[3] Aalborg University, Denmark
[4] University of Vienna, Austria

Abstract. This paper studies a fundamental dynamic clustering problem. The input is an online sequence of pairwise communication requests between n nodes (e.g., tasks or virtual machines). Our goal is to minimize the communication cost by partitioning the communicating nodes into ℓ clusters (e.g., physical servers) of size k (e.g., number of virtual machine slots). We assume that if the communicating nodes are located in the same cluster, the communication request costs 0; if the nodes are located in different clusters, the request is served remotely using inter-cluster communication, at cost 1. Additionally, we can migrate: a node from one cluster to another at cost $\alpha \geq 1$.
We initiate the study of a stochastic problem variant where the communication pattern follows a fixed distribution, set by an adversary. Thus, the online algorithm needs to find a good tradeoff between benefitting from quickly moving to a seemingly good configuration (of low inter-cluster communication costs), and the risk of prematurely ending up in a configuration which later turns out to be bad, entailing high migration costs.
Our main technical contribution is a deterministic online algorithm which is $O(\log n)$-competitive with high probability (w.h.p.), for a specific but fundamental class of problems: namely on ring graphs.
We also provide first insights in slightly more general models, where the adversary is not restricted to a fixed distribution or the ring.

1 Introduction

Modern distributed systems are often highly virtualized and feature unprecedented resource allocation flexibilities. For example, these flexibilities can be exploited to improve resource utilization, making it possible to multiplex more applications over the same shared physical infrastructure, reducing operational costs and increasing profits. However, exploiting these resource allocation flexibilities is non-trivial, especially since workloads and resource requirements are time-varying.

This paper studies a fundamental dynamic resource allocation problem underlying many network-intensive distributed applications, e.g., batch processing or streaming applications, or scale-out databases. To minimize the resource footprint (in terms of bandwidth) of such

[⋆] Research supported by the German-Israeli Foundation for Scientific Research and Development, G.I.F. No I-1245-407.6/2014). Part of the research was done while the second author was visiting Ben Gurion University and TU Berlin. A preliminary version of this paper was presented at the 5th International Conference on Networked Systems (NETYS), Marrakech, Morocco, May 2017 [3].

applications as well as latency, we want to collocate frequently communicating tasks or virtual machines on the same physical server, saving communication across the network. The underlying problem can be seen as a clustering problem [4]: nodes (the tasks or virtual machines) need to be partitioned into different clusters (the physical servers), minimizing inter-cluster communications.

The clustering problem is challenging as the detailed communication patterns are often stochastic and the specific distribution unknown ahead of time. In other words, a clustering algorithm must *deal with uncertainties*: although two nodes may have communicated frequently in the past, it can turn out later that it is better to collocate different node pairs. Accordingly, clustering decisions may have to be reconsidered, which entails migrations.

Our Contributions. This paper initiates the study of a natural dynamic clustering problem where communication patterns follow an unknown distribution, chosen by an adversary: the distribution represents the worst-case for the given online algorithm, and communication requests are drawn i.i.d. from this distribution. Our goal is to devise online algorithms which perform well against an optimal offline algorithm which has perfect knowledge of the distribution. Our main technical contribution is a deterministic online algorithm which, for a special but fundamental request pattern family, namely the ring, achieves a competitive ratio of $O(\log n)$, with high probability (w.h.p.), i.e., with probability at least $1-1/n^c$, where n is the total number of nodes and c is a constant.

We also initiate the discussion of slightly more general models, where the adversary is not restricted to a fixed distribution or a ring, but can pick arbitrary requests from a perfect partition. We present an $O(n)$-competitive algorithm for this more general model of learning a perfect partition.

Novelty and Challenges. Our work presents an interesting new perspective on several classic problems. For example, our problem is related to the fundamental statistical problem of guessing the most likely distribution (and its parameters) from which a small set of samples is drawn. Indeed, one natural strategy of the online algorithm could be to first simply sample requests, and once a good estimation of the actual distribution emerges, directly move to the optimal clustering configuration. However, as we will show in this paper, the competitive ratio of this strategy can be very bad: the communication cost paid by the online algorithm during sampling can be high. Accordingly, the online algorithm is forced to eliminate distributions early on, i.e., it needs to migrate to seemingly low-cost configurations. And here lies another difference to classic distribution learning problems: in our model, an online algorithm needs to pay for changing configurations, i.e., when revising the "guessed distribution". In other words, our problem features an interesting *combination of distribution learning and efficient searching*. It turns out that amortizing the migration costs with the expected benefits (i.e., the reduced communication costs) at the new configuration however is not easy. For example, if the request distribution is uniform, i.e., if all clustering configurations have the same probability, the best strategy is not to move: the migration costs cannot be amortized. However, if the distribution is "almost uniform", migrations are required and "pay off". Clearly, distinguishing between uniform and almost uniform distributions is difficult from an online perspective.

Organization. The remainder of this paper is organized as follows. In Section 2, we introduce our formal model. In Section 3, we provide intuition about our problem and highlight the challenges. In Section 4, we present our deterministic online algorithm, and we analyze it

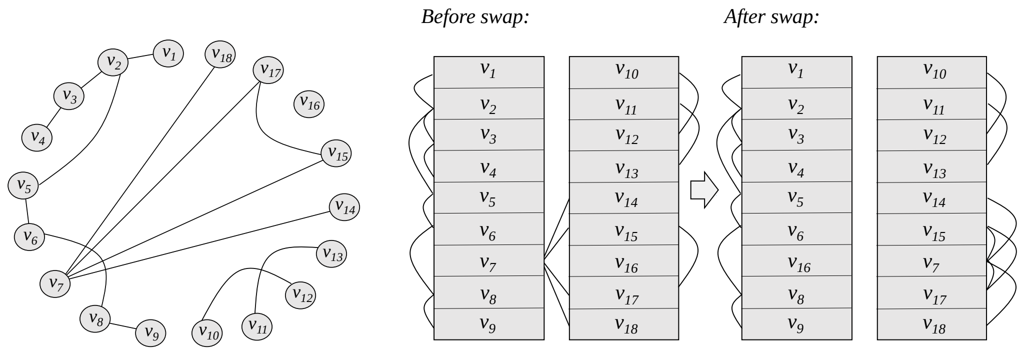


Fig. 1.1: Example: Communication patterns drawn from a certain distribution (*on the left*, represented as a communication graph) need to be learned and clustered. In this example, we have $\ell = 2$ clusters of size $k = 9$. In the middle, a bad clustering is shown: there are four inter-cluster edges ("before swap"). However, by swapping nodes v_7 and v_{16}, all inter-cluster edges can be removed (*on the right* in the figure). Note that different edges can have different frequencies, which however are not depicted in this example.

formally in Section 5. Next in section 6, we present additional online algorithms for more general models. After reviewing related work in Section 7, we conclude our contribution in Section 8.

2 Model

We consider the problem of partitioning n nodes $V = \{v_1, v_2, \ldots, v_n\}$ into ℓ clusters of capacity k each. We assume that $n = \ell \cdot k$, i.e., nodes perfectly fit into the available clusters, and there is no slack. We call a specific node-cluster assignment a *configuration* c. We assume that the communication request is generated from a fixed distribution $\mathscr{D}$, chosen in a worst-case manner by the adversary. The sequence of actual requests $\sigma(\mathscr{D}) = (\sigma_1, \sigma_2, ..., \sigma_T)$, is sampled i.i.d. from this distribution: the communication event at time t is a (directed) node pair $\sigma_t = (v_i, v_j)$. Alternatively, we represent the distribution $\mathscr{D}$ as a weighted graph $G = (V, E)$. For an edge $(v_i, v_j) \in E(G)$, let the weight of the edge $p(v_i, v_j)$ denote the probability of a communication request from between v_i and v_j: each edge $e \in E$ has a certain probability $p(e)$ and $\sum_{e \in E} p(e) = 1$. A request (i.e., edge in G) $\sigma_t = (v_i, v_j)$ is called *internal* if v_i and v_j belong to the same cluster at the current configuration (i.e., at the time of the request); otherwise, the request (edge) is called *external*. We will assume that the communication cost of an external request is 1 and the cost of an internal request is 0.

Note that each configuration uniquely defines external edges that form a *"cut"*, interconnecting ℓ clusters in G. Therefore in the following, we will treat the terms "configuration" and "cut" as synonyms and use them interchangeably; we will refer to them by c. Moreover, we define the probability of a cut (or identically a configuration) c as the sum of the probabilities of its *external edges*: $p(c) = \sum_{e \in c} p(e)$. We also note that there are many configurations which are symmetric, i.e., they are equivalent up to cluster renaming. Accordingly, in the following, we will only focus on the actually different (i.e., non-isomorphic) configurations.

To reduce external communication costs, an algorithm can change the current configuration by using *node swaps*. Swapping a node pair costs 2α (two node migrations of cost α each). Since the request probability of different configurations/cuts differs, the goal of the algorithm will be to quickly guess and move toward a good cut, a configuration that reduces its future cost. Figure 1.1 shows an example.

In particular, we are interested in the *online problem variant*: we assume that the distribution $\mathscr{D}$ of the communication pattern (and hence the σ we observe is generated from) is initially *unknown* to the online algorithm. Nevertheless, we want the performance of an online clustering algorithm, *ON*, to be similar to the one of a hypothetical offline algorithm, *OFF*, which knows the request distribution as well as the number of requests in σ, henceforth denoted by $|\sigma|$, ahead of time. In particular, *OFF* can move before any request occurs or σ is generated.

We aim to minimize the competitive ratio, the worst ratio of the online algorithm cost divided by the offline algorithm cost (for a given distribution $\mathscr{D}$ and the same starting configuration c_o):

$$\rho = \max_{\sigma(\mathscr{D})} \frac{ON(\sigma(\mathscr{D}))}{OFF(\sigma(\mathscr{D}))}$$

Here, the cost $ON(\sigma(\mathscr{D}))$ of any algorithm *ON* for a sequence $\sigma(\mathscr{D})$ is the sum of the overall communication costs and the migration costs. Note that for a given $\mathscr{D}$, *ON*, *OFF* and ρ are probabilistic and hence we consider bounds on ρ with high probability.

As a first step, we focus on partitioning problems where $\ell = 2$. We consider a fundamental case, the ring communication pattern. That is, the communication graph G is the cycle graph (a ring) and the event space is defined over the edges $E = \{(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_k), (v_n, v_1)\}$. Moreover, we assume configurations that minimize the cut, that is nodes are partitioned according to contiguous subsequences of the identifier space. Each cluster is (up to modulo) of the form, $\{(v_i, v_{i+1}, \ldots, v_{i+k-1}\}$. This communication pattern is not only fundamental but also captures the aspects and inherent tradeoffs rendering the problem non-trivial. In this model, an algorithm changes configurations using *rotations* (either clockwise or counter-clockwise). A rotation swaps two nodes incident to opposite cut edges (hence incurring the cost 2α). See Figure 3.1.

3 The Challenge of Dynamic Clustering

In order to acquaint ourselves with the problem and understand the fundamental challenges involved in dynamic clustering, we first provide some examples and discuss naive strategies. Let us consider an example with $n = 2k$ nodes divided into $\ell = 2$ clusters of size k. There are k possible configurations/cuts: $\{c_0, c_1, \ldots, c_{k-1}\}$. At one end of the algorithmic spectrum lies a lazy algorithm which never moves, let's call it *LAZY*. At the other end of the spectrum lies a very proactive algorithm which greedily moves to the configuration which so far received the least external requests, let's call it *GREEDY*. Both *LAZY* and *GREEDY* are doomed to fail, i.e., they have a large competitive ratio: *LAZY* fails under a request distribution where the initial external cut has probability 1, i.e., $p(c_0) = 1$ and for any $i > 0$, $p(c_i) = 0$: *LAZY* pays for all

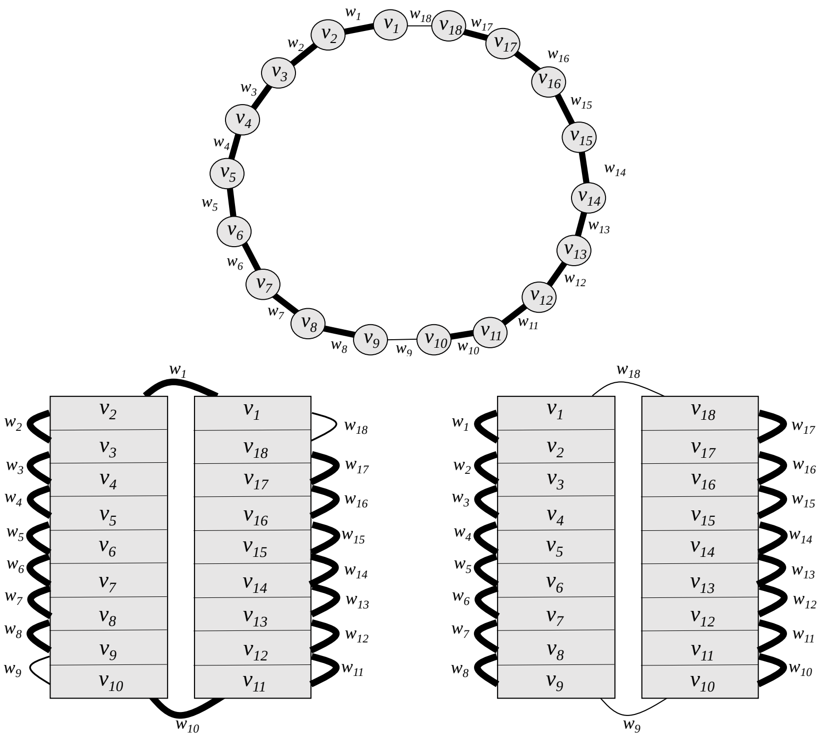

v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9 v_{10} v_{11} v_{12} v_{13} v_{14} v_{15} v_{16} v_{17} v_{18}
w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8 w_9 w_{10} w_{11} w_{12} w_{13} w_{14} w_{15} w_{16} w_{17} w_{18}

Fig. 3.1: Weighted ring communication pattern: frequently used edges (in *bold*) should not be part of the cut. The cut can be changed using rotations: in the figure, a counter-clockwise rotation leads from the middle to the right configuration.

requests, while after a simple node swap all communication costs would be 0. *GREEDY* fails in uniform distributions, i.e., if $p(c_i) = 1/k$ for all i: the best configuration is continuously changing, and in particular, the best cut is likely to be at distance $\Omega(k)$ from the initial configuration c_0: *GREEDY* quickly incurs migration costs in the order of $\Omega(\alpha \cdot k)$, while staying at the same location would cost $1/k$ per request. Thus, the competitive ratios grow quickly in the number of requests and in the number of nodes.

Another intuitive strategy could be to wait in the initial configuration c_0 for some time, simply observing and sampling the actual distribution, until a “sufficiently accurate” estimation of the distribution is obtained. Then, we move directly to the (hopefully) optimal configuration. Thus, the problem boils down to the classic statistical problem of estimating the distribution (and its parameters) from samples. However, it is easy to see that waiting for the optimal distribution to emerge is costly. Imagine for example a scenario where the initial configuration/cut c_0 has a high probability, and there are two additional cuts c_1 and c_2 which have almost the same low probability (for example polynomially low probability) . Clearly, waiting at c_0 to learn whether c_1 or c_2 is better is not only very costly, but it may also be pointless: even if the online algorithm ended up at c_1 although c_2 was a little bit better, the resulting competitive ratio could be still small.

Thus, the key challenge of our problem lies in its required joint optimization of *learning and searching*: while learning the distribution, an efficient search algorithm must be employed to minimize reconfiguration costs. In particular, the following criteria need to be met:

1. *Migrate early...:* An online algorithm should migrate away from a suboptimal configuration early, possibly long before the optimal configuration can be guessed.
2. *... but not too early...:* An online algorithm should avoid frequent migrations, e.g., due to a wrong or poor estimate of the actual request distribution.
3. *... and locally:* Especially if the length of σ is small (small number of requests), it may not make sense to migrate to an optimal but faraway location, even if the distribution is known: even *OFF* would not move there.

4 Deterministic and Competitive Clustering

With these intuitions and challenges in mind, we present our solution. Let us first start with the offline algorithm. It is easy to see that *OFF*, knowing the distribution as well as the number of requests, only moves *once* in time (i.e., one move consisting of multiple migrations or node swaps): namely *in the beginning* and *to the configuration providing an optimal expected cost-benefit tradeoff.* Concretely, *OFF* computes for each configuration c_i, its expected cost-benefit tradeoff: the communication cost of configuration c_i is $|\sigma| \cdot p(c_i)$ and the cost of moving there is $2\alpha \cdot d(c_0, c_i)$, where $d(\cdot,\cdot)$ is the rotation distance between the two configurations (the smallest number of rotation moves to reach the other configuration). Thus, *OFF* will move to $c_{OFF} := \arg\min_{c_i} |\sigma|.p(c_i) + (2\alpha \cdot d(c_0, c_i))$ (note that this configuration is not necessarily unique). In the following, we will use the short form $d_i = d(c_0, c_i)$ to denote distances relative to c_0, the initial configuration.

The online algorithm is more interesting. The competitive and deterministic online algorithm presented in this paper relies on three key ideas:

– *Eliminating bad configurations:* We define conditions for configurations which, if met, allow us to eliminate the corresponding configurations once and for all. In particular, we will guarantee (w.h.p.) that an online algorithm be competitive (even) if it never moves back to such a configuration anymore in the future. In other words, our online algorithm will only move between configurations for which this condition is not true yet.
– *Local migrations and growing-radius search strategy:* In order to avoid high migration costs, our online algorithm is local in the sense that it only moves to nearby cuts/configurations once the condition of the current configuration is met and it needs to be eliminated. Concretely, our online algorithm is based on a growing-radius search strategy: we only migrate to valid configurations lying within the given radius. Only if no such configurations exist, the search radius is increased.
– *Amortization:* The radius growth strategy alone is not sufficient to provide the necessary amortization for being competitive. Two additions are required:
  1. *Directed search:* An online algorithm may still incur a high migration cost when frequently moving back-and-forth within a given radius, chasing the next best configuration. Therefore, our proposed online algorithm first moves in one direction only (clockwise), and then in the other direction, bounding the number of times the c_0 configuration is crossed.

2. *Lazy expansion:* Even once all configurations within this radius have been eliminated, the online algorithm should not immediately move to configurations in the next larger interval. Rather, the algorithm waits until a certain amount of requests have been accumulated, allowing to amortize the migrations (an "insurance").

With these high-level ideas in mind, we now describe the algorithm in detail (cf. Algorithm 1). We consider a time t, and assume that the online algorithm is at configuration c_t. The algorithm maintains an array $r[]$ where it counts, for each possible configuration $c_0, \dots c_{k-1}$, the number of samples that hit an external edge of the corresponding cut; in other words, $r[]$ is used to estimate the distribution of the communication pattern. Let $\mathscr{E}$ be the set of the eliminated configurations, and let $\overline{\mathscr{E}}$ be the complement of $\mathscr{E}$: the set of configurations not eliminated yet. R is the search radius, initially $R = 1$. Upon each request, σ_t, we first increment the value of the corresponding configuration in the sampling array $r[]$ (only one configuration is affected by a given external request). We then compare all configurations not eliminated yet to the "seemingly best configuration": the configuration which received the least (external) requests so far (i.e., $\arg\min_{c_i} r[c_i]$). Let $r_{\min} := \min_{c_i} r[c_i]$ be the minimum value. We now eliminate any configuration c_j for which the condition $\texttt{Cond}(r[c_j], r_{\min}, \epsilon)$ is fulfilled: c_j is too far from the optimum. Concretely, w.l.o.g. assume that $r[c_j] > r[c_i]$ and let $\gamma = r[c_i]/r[c_j] < 1$. Then for $\epsilon > 0$ (a parameter for the error probability), we use the following condition:

$$\texttt{Cond}(r[c_j], r[c_i], \epsilon) := \begin{cases} \texttt{True} & r[c_j] \geq \frac{2\ln(\frac{1}{\epsilon})}{1+\gamma\left(2\ln\left(\frac{2\gamma}{\gamma+1}\right)-1\right)} \\ \texttt{False} & \text{otherwise} \end{cases} \tag{1}$$

If on this occasion, we eliminated our own current configuration $c(t)$, i.e. $\texttt{Cond}(c(t), c_{min}, \epsilon)$ evaluates to True, then at line 11 we decide where to move next (unless all configurations have been eliminated). The distance from the suggested next configuration c_{next} to c_0 (the initial configuration) may be greater than the current radius R, in which case we double R until $R \geq d_{next}$. However, before moving, we also test whether $\min_{\{d_{next}<R\}}(r[c_{next}]) \geq \alpha \cdot R$. Only if this is fulfilled, we can move to the new configuration c_{next}; otherwise, we lazily stay on the current configuration.

Let us now elaborate more on the moving strategy. Before going into the details however, let us note that for ease of presentation, we will use two different but equivalent numbering schemes to refer to configurations: depending on what is more useful in the current context. In particular, while talking about the number of requests, $r[]$, we often enumerate configurations globally, $0, 1, 2, \dots, k$. When discussing moving strategies, we often enumerate configurations relative to c_0, i.e., $-1, 1, -2, 2, \dots, c_{k/2}$, depending on whether they are located clock- or counter-clock wise from c_0.

Given this remark, let us consider a simple migration strategy: we could always move to the closest not eliminated configuration next. However, we can show that this strategy is flawed. To see this, consider the following distribution:

$$\forall i \in [1, \frac{k}{2}] \ : \ p(c_i) = \frac{1}{k^i}, \quad p(c_0) = \left(1 - \sum_{i \in [1; \frac{k}{2}]} p(c_i)\right), \quad \forall i \in (-\frac{k}{2}, -1] \ : \ p(c_i) = 0$$

In such a situation, we have to move away from the configuration c_0 as soon as possible: we pay a cost close to 1 on this configuration, for each request. In particular, we cannot wait

Algorithm 1 Online Algorithm *ON* (upon receiving request $\sigma(t)$ and current configuration $c(t)$)

```
Initialize: $r := [0,..,0]$, $\mathscr{E} := \{\}$, $\overline{\mathscr{E}} := \{-\frac{k}{2}+1,\dots,\frac{k}{2}\}$, $R := 1$  $\epsilon := \frac{1}{n^2}$
 1: $c_j := c(\sigma(t))$      (* configuration to which $\sigma(t)$ is external *)
 2: $r[c_j]++$
 3: $r_{\min} := \min\{r[i] \mid i \in [1,k]\}$
 4: if $c_j \in \overline{\mathscr{E}}$ then
 5:     if Cond($r[c_j], r_{\min}, \epsilon$) then
 6:         remove $c_j$ from $\overline{\mathscr{E}}$
 7:         add $c_j$ to $\mathscr{E}$
 8:     end if
 9: end if
10: if $c(t) \in \mathscr{E}$ then
11:     $c_{next}$ := The next configuration $c_i \in \overline{\mathscr{E}}$ on the searching path
12:     while $d_{next} > R$ do
13:         $R = 2R$
14:     end while
15:     if $r[c(t)] \geq \alpha \cdot d_{next}$ then
16:         move from $c(t)$ to $c_{next}$
17:         $c(t) := c_{next}$
18:     end if
19: end if
```

until we even observe the first request on c_1: we would incur high communication costs. Now, however, the algorithm may move in the wrong direction: e.g., to c_1, and then to the closest configuration not eliminated, c_2. Thus, eventually all configurations in $[c_0, c_{k/2}[$ may be visited before reaching the minimal configurations.

This is reminiscent of classic line searching [13] type problems like "the goat searches the hole in the fence"-escape problems: moving in one direction only, the goat may risk missing a nearby hole in the other direction. That is, moving greedily in one direction is $\Omega(F)$ competitive only, where F is the circumference of the fence, which in our case means that the competitive ratio is $\Omega(k)$. Accordingly, some combination of search-left and search-right is required. Our search radius R is centered around c_0 at any time during the execution of the algorithm, and we always first explore all remaining non-eliminated configurations in one direction, and then explore the remaining configurations in the other direction. In other words, starting from c_0, we alternate the search between the positive and negative configurations following the sequence: $(0,-1,-2,1,2,3,4,-3,-4,\dots,-8,5,\dots,16,\dots,-2^{2i-1}-1,\dots,-2^{2i+1},2^{2i}+1,\dots,2^{2i+2},\dots)$. Thus, configuration c_0 is crossed only a constant number of times per given radius R. We call this sequence the *searching path*.

Given a moving strategy, we next note that we should not move too fast: we introduce a second condition for when it is safe to move. When in a configuration 2^{2i} and before we want to explore configurations in $[-2^{2i+1},-2^{2i-1}]$, we wait in the configuration $c_{\min}$ between configurations -2^{2i-1} and 2^{2i}, until this configuration fulfills $r[c_{\min}] \geq \alpha \cdot 2^{2i+1}$. Similarly, when moving from the configuration -2^{2i+1} to explore the configurations in $[2^{2i}, 2^{2i+2}]$, we will wait at $c_{\min}$ between $[-2^{2i+1}, 2^{2i}]$, until $r[c_{\min}] \geq \alpha \cdot 2^{2i+2}$.

5 Analysis

We first make some general observations on our elimination condition. Subsequently, we will present a cost-breakdown which will be helpful to analyze the competitive ratio of *ON*: we will show that each cost component is competitive with respect to the optimal offline algorithm. We first prove the following helper claim.

Claim 1 *Assume c_i and c_j are hit with probabilities $p(c_j) \le p(c_i)$. Let random variables R_i and R_j represent the respective number of hits. After a total of $a+b$ hits on these two configurations for any $b > a$, it holds that*

$$\Pr\Big(R_i \le a \;\; and \;\; R_j \ge b\Big) \le \left(\frac{e^{-\delta}}{(1-\delta)^{1-\delta}}\right)^{X}$$

where $\delta = \frac{b-a}{b+a}$ and $X = \frac{a}{1-\delta} = \frac{b}{1+\delta}$.

Proof. The proof idea is to consider two probabilities using known Chernoff Bounds [17])(since R_i and R_j are sum of i.i.d's):

$$P_i := \Pr(R_i \le (1-\delta')E[R_i]) \le \left(\frac{e^{-\delta'}}{(1-\delta')^{1-\delta'}}\right)^{E[R_i]} \text{ and} \tag{2}$$

$$P_j := \Pr(R_j \ge (1+\delta')E[R_j]) \le \left(\frac{e^{\delta'}}{(1+\delta')^{1+\delta'}}\right)^{E[R_j]} ,0 \le \delta' < 1. \tag{3}$$

The two events ($R_i \le a$ and $R_j \ge b$) are not independent, but we can bound the probability for the intersection of the two events by the minimum of the two probabilities. Towards this objective, we find out which one of the bounds is smaller. Let

$$B_i := \frac{e^{-\delta'}}{(1-\delta)^{1-\delta'}} \text{ and } B_j := \frac{e^{\delta'}}{(1+\delta')^{1+\delta'}}.$$

In order to determine the smallest of the bounds in (2) and (3), we first study the function:

$$F(\delta') := \frac{B_i}{B_j} = e^{-2\delta'} \cdot \frac{(1+\delta')^{1+\delta'}}{(1-\delta')^{1-\delta'}}.$$

We obtain that for $0 \le \delta' < 1$, $F(\delta') \le 1$, therefore $B_i \le B_j$. Since $E[R_i] \ge E[R_j]$, we have $B_i^{E[R_i]} \le B_j^{E[R_j]}$:

$$\left(\frac{e^{-\delta'}}{(1-\delta')^{1-\delta'}}\right)^{E[R_i]} \le \left(\frac{e^{\delta'}}{(1+\delta')^{1+\delta'}}\right)^{E[R_j]}. \tag{4}$$

Next, we study $E[R_i]$. We know that in a random subsequence of length ℓ, $E[R_i] = \ell \cdot p(c_i)$. In order to prove the claim, we need to consider a random subsequence of length $a+b$ in which each request hits either c_i or c_j. Given this assumption, the probability of hitting either configurations scales up by $\frac{1}{p(c_i)+p(c_j)}$. Therefore,

$$E[R_i] = (a+b)\cdot\frac{p(c_i)}{p(c_i)+p(c_j)} \geq \frac{(a+b)}{2} = \frac{b}{1+\delta} = \frac{a}{1-\delta} = X, \tag{5}$$

where the inequality is due to the assumption $p(c_j) \leq p(c_i)$.

Using the smaller bound (left-hand side in (4)) as the upper bound, we conclude the claim:

$$\begin{aligned}
\Pr\Big(R_i \leq a \text{ and } R_j \geq b\Big) &\leq \Pr\Big(R_i \leq a\Big) \\
&= \Pr\Big(R_i \leq (1-\delta)X\Big) && (6)\\
&\leq \Pr\Big(R_i \leq (1-\delta)E[R_i]\Big) && (7)\\
&\leq \left(\frac{e^{-\delta}}{(1-\delta)^{1-\delta}}\right)^{E[R_i]} && (8)\\
&\leq \left(\frac{e^{-\delta}}{(1-\delta)^{1-\delta}}\right)^{X} := B\langle a,b\rangle, && (9)
\end{aligned}$$

where (6) holds by definition, (8) is the lower tail Chernoff bound in (2), and (7) and (9) hold due to the fact $X \leq E[R_i]$.

End of Proof of Claim 1

The next lemma provides an intuition of our algorithm and its condition.

Lemma 1. *Let $\epsilon > 0$, then if $\texttt{Cond}(r[c_j], r[c_i], \epsilon) = \mathit{True}$,*

$$\Pr\big(p(c_j) > p(c_i)\big) \geq 1-\epsilon.$$

Proof. Equivalently, we prove that whenever $p(c_j) \leq p(c_i)$, $\Pr\big(\texttt{Cond}(r[c_j], r[c_i], \epsilon) = True\big) \leq \epsilon$. Using Claim 1, if $p(c_j) \leq p(c_i)$ and $r[c_j] > r[c_i]$ then

$$\Pr\big(\texttt{Cond}(r[c_j], r[c_i], \epsilon) = True\big) \leq \Pr\big(R_i \leq r[c_i] \text{ and } R_j \geq r[c_j]\big) \leq B\langle r[c_i], r[c_j]\rangle.$$

We want that $B\langle r[c_i], r[c_j]\rangle \leq \epsilon$:

$$\left(\frac{e^{-\delta}}{(1-\delta)^{1-\delta}}\right)^{\frac{r[c_j]}{1+\delta}} \leq \epsilon \iff \frac{r[c_j]}{1+\delta}(-\delta-(1-\delta)\ln(1-\delta)) \leq \ln(\epsilon) \iff$$

$$r[c_j] \geq \frac{(1+\delta)\ln(\epsilon)}{(-\delta-(1-\delta)\ln(1-\delta))} = \frac{(1+\delta)\ln(\frac{1}{\epsilon})}{\delta+(1-\delta)\ln(1-\delta)}.$$

Now let $\gamma = \frac{r[c_i]}{r[c_j]} < 1$, so $\delta = \frac{1-\gamma}{1+\gamma}$, and we have:

$$\begin{aligned}
r[c_j] &\geq \frac{\left(1+\frac{1-\gamma}{1+\gamma}\right)\ln(\frac{1}{\epsilon})}{\left(\frac{1-\gamma}{1+\gamma}\right)+\left(1-\frac{1-\gamma}{1+\gamma}\right)\ln\left(1-\frac{1-\gamma}{1+\gamma}\right)} \\
&= \frac{\left(\frac{2}{1+\gamma}\right)\ln(\frac{1}{\epsilon})}{\left(\frac{1-\gamma}{1+\gamma}\right)+\left(\frac{2\gamma}{1+\gamma}\right)\ln\left(\frac{2\gamma}{1+\gamma}\right)} = \frac{2\ln(\frac{1}{\epsilon})}{(1-\gamma)+2\gamma\ln\left(\frac{2\gamma}{1+\gamma}\right)} = \frac{2\ln(\frac{1}{\epsilon})}{1+\gamma\left(2\ln\left(\frac{2\gamma}{\gamma+1}\right)-1\right)},
\end{aligned}$$

which concludes the proof of the lemma. □

5.1 A Cost Breakdown

It is convenient to break down the algorithm costs into different components. In case of *OFF*, the situation is fairly easy: *OFF* simply incurs a migration cost, hencefoth denoted by OFF_{mig}, of $\text{OFF}_{mig} = 2\alpha \cdot d_{OFF}$ to move to the optimal location c_{OFF}, where d_{OFF} is the rotation distance between c_0 and c_{OFF}, plus an expected communication cost OFF_{comm} of $|\sigma| \cdot p(c_{OFF})$.

In case of *ON*, the situation is more complicated. In particular, while we do not distinguish between different migration costs for *ON* either, we consider three types of communication costs for *ON*: ON_{elim} is the elimination cost, i.e., the total communication cost incurred while *ON* is waiting on every configuration that has not been eliminated yet, until the condition $\texttt{Cond}(j, i, \epsilon)$ is fulfilled for the current configuration. ON_{ins} is the "insurance" cost paid by *ON* when waiting in an already eliminated configuration, until being allowed to actually move beyond the current radius to a non-eliminated configuration. Finally, ON_{final} is the communication cost paid by *ON* once it reached its final configuration and all other configurations have been eliminated. (Note that the cost incurred at the final configuration while there are still other, non-eliminated configurations, is counted toward elimination costs.)

The total communication cost ON_{comm} is the sum of these three costs. In the following, we will prove that all these cost components are competitive compared to *OFF*'s overall costs, from which the bound on the competitive ratio is obtained.

5.2 Competitive Ratio

We now prove that our online algorithm *ON* performs well with high probability (w.h.p.). That is, we derive a competitive ratio of $O(\log k)$ which holds with probability at least $1 - 1/n^q$ for some constant q.

Theorem 1. *The competitive ratio achieved by* ON *is* $\rho \in O(\log n)$ *with high probability.*

The remainder of this section is devoted to the proof of this theorem. In particular, we will use our cost breakdown, and express the competitive ratio as (where $\sigma = \sigma(\mathcal{D})$):

$$\rho = \max_{\sigma}\left(\frac{\text{ON}(\sigma)}{\text{OFF}(\sigma)}\right) = \max_{\sigma}\left(\frac{\text{ON}_{mig}(\sigma) + \text{ON}_{elim}(\sigma) + \text{ON}_{ins}(\sigma) + \text{ON}_{final}(\sigma)}{\text{OFF}_{comm}(\sigma) + \text{OFF}_{mig}(\sigma)}\right)$$

We will prove that each cost component in ON is competitive to *OFF*'s overall cost, therefore resulting in an $O(\log n \cdot \text{OFF}(\sigma))$ bound.

Elimination Costs To calculate the elimination cost (the total cost resulting from waiting at different configurations until $\texttt{Cond}()$ holds for the current configuration), we divide all configurations into two sets: configurations c for which $p(c) \leq 20p_{\min}$ and configurations c' for which $p(c') > 20p_{\min}$. We consider the elimination cost for these two sets in turn.

- All configurations c for which $p(c) \leq 20p_{\min}$. We will consider again two cases. Let $e[c]$ the cost of elimination on a position c (number of requests served until the condition of elimination of c is fulfilled). Either $e[c] \leq 20\log n$ or $e[c] > 20\log n$. In the first case we can

just say that the number of configuration we have to eliminate is in $O(\text{ON}_{migr})$ and so $\sum_{e(c_i)\le 20\log n} e(c_i) \le O(\log n \cdot \text{ON}_{migr}) = O(\log n \cdot \text{OFF})$.
For the other case, where $e(c_i) > 20 \cdot \log n$, we use the following claim:

Claim 2 *Let* $\Delta = [t_1, t_2]$ *be a time interval. We note* $r[c](\Delta) = r[c](t_2) - r[c](t_1)$*, where* $r[c](t)$ *is the number of requests on the configuration c at the time t. Then:*
If $p(c_j) \le 20p(c_i)$ *and* $r[c_j](\Delta) \ge 20\log n$ *then w.h.p.* $r[c_j](\Delta) \le 40r[c_i](\Delta)$.

Proof. First note that from the bound of Eq. (3) w.h.p. $r[c_j](\Delta) \le 2E[r[c_j](\Delta)]$. Similarly since $E[r[c_i]] \ge \frac{1}{20}E[r[c_j]]$ we have that w.h.p. $r[c_i](\Delta) \ge \frac{1}{2}E[r[c_i](\Delta)] \ge \frac{1}{40}E[r[c_j](\Delta)]$. So w.h.p. $r[c_j](\Delta) \le 40r[c_i](\Delta)$. □

From Claim 2 and union bound over at most n states we get that w.h.p. $r[c_j](\Delta_j) \le 40r[c_{\min}](\Delta_j)$ for all such configurations, with Δ_j denoting the time interval where we stayed on the configuration c_j, and c_j was not eliminated (which means $\bigcup_{c_j} \Delta_j = [0, |\sigma|]$). So

$$\sum_{e(c_j)>20\log n} e(c_j) = \sum_{e(c_j)>20\log n} r[c_j](\Delta_j) \le \sum_{e(c_j)>20\log n} 40r[c_{min}](\Delta_j)$$
$$\le 40r[c_{min}]([0,|\sigma|]) = 40r[c_{min}] \le O(OFF_{comm})$$

In conclusion as $\text{ON}_{elim\le 20} = \sum_{e(c_i)\le 20\log n} e(c_i) + \sum_{e(c_i)>20\log n} e(c_i)$ we have w.h.p.:

$$\frac{\text{ON}_{elim\le 20}(\sigma)}{\text{OFF}(\sigma)} = O(1).$$

– All configurations c' for which $p(c') > 20p_{\min}$. For this we claim:

Claim 3 *If* $p(c_j) \ge 20p(c_i)$ *and* $r[c_j] \ge 20\log n$ *then w.h.p.* $r[c_j] > 5r[c_i]$ *and* $\texttt{Cond}(r[c_j], r[c_i], \epsilon)$ *is* $\texttt{True}$ *for* $\epsilon = \frac{1}{n^2}$.

Proof. Since $r[c_j] \ge 20\log n$ w.h.p. $E[r[c_j]] \le 2r[c_j]$. If $r[c_i] > \frac{1}{5}r[c_j]$ then w.h.p. $E[r[c_i]] > \frac{1}{10}r[c_j]$, but this contradicts the assumption that $E[r[c_i]] \le \frac{1}{20}E[r[c_j]]$. So we have $\frac{r[c_i]}{r[c_j]} \le \frac{1}{5}$ and $\texttt{Cond}(j, i, \epsilon)$ holds for $\epsilon = \frac{1}{n^2}$. □

Now since the number of configurations ON needs to eliminate is lower than $\text{ON}_{mig}/\alpha \le \text{ON}_{mig}$, the total cost ON paid is $O(\text{ON}_{mig} \cdot \log n)$. But since $\frac{\text{ON}_{mig}(\sigma)}{\text{OFF}(\sigma)} = O(1)$ (as we show next) we have :

$$\frac{\text{ON}_{elim>20}(\sigma)}{\text{OFF}(\sigma)} = O(\log n)$$

To conclude $\text{ON}_{elim} = \text{ON}_{elim\le 20} + \text{ON}_{elim>20}$, and: $\text{ON}_{elim}(\sigma)/\text{OFF}(\sigma) = O(\log n)$.

Migration Cost We distinguish two cases. Let c_{far} be the farthest configuration reached by our online algorithm. Either d_{far} (the distance between c_{far} and c_0) is lower than d_{OFF}, or it is greater than d_{OFF}.

– In the first case, $d_{OFF} \geq d_{far}$, we can prove

Lemma 2. *if* $d_{OFF} \geq d_{far}$ *then* $\text{ON}_{mig} \leq 6 \cdot \text{OFF}_{mig}(\sigma)$.

Proof. $\exists x \in \mathbb{N}\ \ 2^{2x} \leq d_{far} < 2^{2x+2}$. Then, in the worst case, we have to go to 2^{2x+2}. Moreover, after completing the search within radius 2^i we double the radius and continue without changing direction. Therefore, the rotation distance 2^i is charged at most 3 times.

$$\text{ON}_{mig}(\sigma) \leq \sum_{i=0}^{2x+1} 3 \cdot 2^i \cdot \alpha \leq 6 \cdot 2^{2x+1} \cdot \alpha \leq 6 d_{far} \alpha \leq 6 \cdot d_{OFF} \cdot \alpha \leq 6 \cdot \text{OFF}_{mig}(\sigma)$$

□

– If $d_{OFF} < d_{far}$, then from Claim 3 and Claim 2 with $\Delta = [0, |\sigma|]$ it follows that w.h.p. $r[c_{OFF}] \in \Omega(\alpha \cdot d_{far})$: Recall that in our algorithm (line 15) we only move beyond the current radius if the corresponding costs have been amortized. Hence $\text{ON}_{mig} \leq \text{OFF}_{comm}$.

In conclusion, in both cases: $\text{ON}_{mig}(\sigma)/\text{OFF}(\sigma) = O(1)$.

Insurance Costs For the insurance cost we also consider several cases. Let c_{far} be the farthest configuration reached by our online algorithm. Let c_{OFF} denote the location of the offline algorithm. We split ON_{ins} into two parts: $\text{ON}_{ins<far}$ and $\text{ON}_{ins=far}$. $\text{ON}_{ins<far}$ is the insurance cost up to (not including) c_{far} while $\text{ON}_{ins=far}$ is the insurance cost paid on c_{far}. The last insurance cost, paid before the last migration to c_{far}, is αd_{far}, so we have $\text{ON}_{ins<far} \leq O(\text{ON}_{mig}) = O(\text{OFF})$ (see the migration cost analysis).

The only possible problem is therefore $\text{ON}_{ins=far}$. Now we consider two cases:

– c_{OFF} is in $\mathcal{E}$ (eliminated configuration). Since c_{OFF} was eliminated before c_{far} if follows from Claims 3 and 2 that w.h.p. $r[c_{OFF}] > \Omega(r[c_{far}])$ so $\text{ON}_{ins=far} < O(\text{OFF}_{comm})$.
– c_{OFF} is in $\overline{\mathcal{E}}$. In this case because of our *searching path* and the selection of c_{next}, we have $d_{OFF} \geq d_{next}/2$. Therefore $\text{ON}_{ins=far} \leq O(\text{OFF}_{mig})$.

Overall we have: $\text{ON}_{ins}(\sigma)/\text{OFF}(\sigma) = O(1)$.

Final Costs By definition, in the final configuration, all other configurations have been eliminated. Thus, our condition, $\texttt{Cond}(r[c_j], r[c_i], \epsilon)$, has been fulfilled at some point for any c_j, with respect to some c_i. The probability that we eliminate a minimum configuration and end up at a suboptimal configuration is small. This follows from Lemma 1, when setting $\epsilon := \frac{1}{n^2}$: once we stopped in a configuration, it is, with high probability, a (not necessarily unique) minimal configuration. Since *OFF* directly moves to a minimum configuration (which may not be unique), *ON* cannot incur a higher cost than *OFF* on a specific minimum configuration, i.e., not more than $r[c_{\min}]$. As the offline algorithm moved from the start to a configuration c_{OFF} and $r[c_{\min}]$ is the configuration with the lowest number of requests, $r[c_{OFF}] \geq r[c_{\min}]$. Thus, $\text{ON}_{final}(\sigma) \leq \text{OFF}(\sigma)$, and also $\text{ON}_{final}(\sigma)/\text{OFF}(\sigma) = O(1)$.

Overall Costs In conclusion, with high probability:

$$\rho \leq \max_{\sigma}\left(\frac{\mathrm{ON}_{mig}(\sigma)+\mathrm{ON}_{elim}(\sigma)+\mathrm{ON}_{ins}(\sigma)+\mathrm{ON}_{final}(\sigma)}{\mathrm{OFF}_{comm}(\sigma)+\mathrm{OFF}_{mig}(\sigma)}\right)=O(\log n)$$

6 Beyond Stochastic Adversary

So far we assumed that the adversary is restricted to sample requests i.i.d. from a distribution of its choice. In this section we make a first attempt to relax this assumption and consider an adversary who can adapt the communication frequencies depending ON's deterministic choices. However, we require that the requests come from a perfect partition in the following sense: there exists a configuration without any inter-cluster communications. An optimal offline algorithm may hence simply move to such a perfect partition (the closest one from the initial configuration), and the goal of the online algorithm is *to learn a perfect partition.*

6.1 Ring Communication Pattern

We start by assuming that the perfect partition is a subset of a ring communication pattern (but frequencies can change arbitrarily over time). Observe that in this model, ON must move as soon as a remote request hits the current configuration: otherwise the adversary will simply repeat this request arbitrarily. A naive strategy would be to move to the next configuration, e.g., in clockwise direction. This algorithm is $O(k)$-competitive: If $d_F = d(c_0, c_F) \geq 1$ (where c_F is the perfect partition closest to the initial configuration), OFF pays at least $\Omega(\alpha)$ to reach c_F, whereas ON may pay $O(\alpha k)$ if the optimal configuration is in counter-clockwise direction.

We can improve this algorithm by replacing the search strategy, in the spirit of our previous algorithms. Starting from c_0, ON visits all configurations within the current radius R before moving to a configuration c_i s.t. $R < d(c_0, c_i) \leq 2R$. Concretely, ON moves according to the sequence $(0,1,-1,-2,2,3,4,-3,-4,\ldots,-8,5,\ldots,8,\ldots,2^{i-1}+1,\ldots,2^i,-2^{i-1}-1,\ldots,-2^i,\ldots)$ when the current radius is $R = 2^i$ and $i > 0$ is even. Similarly, for $j > 0$ being odd, the search sequence within the current radius $R = 2^j$ is $-2^{j-1}-1,\ldots,-2^j,2^{j-1}+1,\ldots,2^j$. Notice that whenever the search within the current radius $R = 2^h$ is complete, we extend the search to the next radius 2^{h+1} without changing direction.

Note that ON crosses c_0 exactly once for each radius and this follows an extension from the previous radius. Thus, the cost of ON consists of the cost of extension and the cost incurred while crossing c_0. More precisely, ON pays for at most

$$\sum_{i=0}^{\lceil \log d_F \rceil} (2^i - \lfloor 2^{i-1} \rfloor) + 2^{i+1} \in \Theta(d_F)$$

rotations. Note that ON rotates away from a configuration on the first hit and it does not wait on any configuration that is already examined. Thus, the communication cost is also in $\Theta(d_F)$. Finally, since OFF rotates d_F times, the competitive ratio is in $\Theta(1)$.

6.2 More General Communication Pattern

Let us now remove the constraint that requests need to come from a ring, but allow the adversary to choose request sequences from an arbitrary perfect partition. Again, the goal of the online algorithm is to learn this perfect partition, at low cost.

As the adversary reveals communication edges one by one, we must take the best decision based on the current partially revealed graph. Towards this end, we propose the online algorithm PPL which mimics balanced offline partitioning algorithms and keeps track of (connected) components, based on the communication history. Our online algorithm is detailed in Algorithm 2. It initiates at line 1 by creating components, each containing a single node. The first time two nodes communicate, we merge the corresponding components into a single component (line 5). If the merging components are located on different clusters, then we “rebalance", in order to fit everything perfectly once again into clusters of equal size.

Algorithm 2 PPL: Perfect Partition Learner

```
Input: clusters $\mathscr{A}$ and $\mathscr{B}$ and initial partitions $A_I$ and $B_I$
1: for each node $v$ create a singleton component $C_v$ and add it to $\mathscr{C}$
2: on each communication $\sigma_t = \{u, v\}$:
3: Let $C_1 \ni u$ and $C_2 \ni v$ be the container components
4: if $C_1 \neq C_2$ then
5:     unite the two components into a single component $C'$ and $\mathscr{C} = (\mathscr{C} \setminus \{C_1, C_2\}) \cup \{C'\}$
6:     if cluster($C_1$) $\neq$ cluster($C_2$) then
7:         Rebalance()
8:     end if
9: end if
```

We specify the re-balancing separately as sub-routines in Algorithm 3.

The re-balancing implements a standard dynamic program known from Partition or Subset Sum problems. In addition, "rebalance()" embeds a heuristic necessary to achieve a good competitive ratio. Specifically, we compute the specific partitions that are *closest* to the initial configuration, denoted by A_I. Let (A', B') be the current configuration (i.e. the content of the clusters $\mathscr{A}$ and $\mathscr{B}$). We define the current distance as $d(A', A_I) = |A' \triangle A_I|$.

The lines 13 and 15 reflect this choice of partitioning. More technically, the dynamic program in Algorithm 3 computes the minimum distance partition for all possible cluster sizes (up to k), which is stored as sub-solutions $P(i, j)$. Each sub-problem identified by the pair (i, j) corresponds to a min-distance partition of the first j components into two clusters of size i. Each $P(i, j)$ is computed by considering whether to take the last component $\mathscr{C}_j$ for the cluster $\mathscr{A}$ or not. If the component originally was located on $\mathscr{A}$ (i.e. $j \in A_I$) then not putting it back there increases the distance by $|\mathscr{C}_j|$. Conversely, relocating a component to $\mathscr{A}$ knowing that it was not initially there also increases the distance in a similar way.

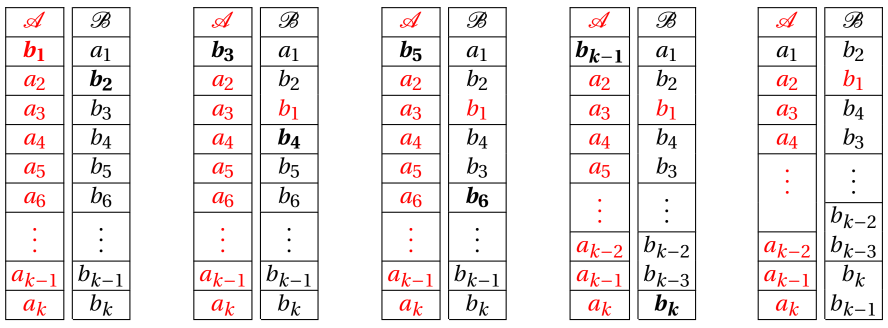
Next, at line 24, the algorithm traces the dynamic program choices in reverse direction beginning with the topmost solution $P(k, m)$ and constructs the new partition. Eventually at line 27, the actual re-balancing takes place by swapping nodes between the clusters until nodes that belong to the same component are collocated on the same cluster.

We have the following result.

Algorithm 3 Sub-algorithm for partitioning nodes in $\mathscr{A} \cup \mathscr{B}$

```
*Rebalance ()* {
$m = |\mathscr{C}|$                                          ▷ number of current components
$n = 2k$                                                      ▷ number of nodes
$P$ = empty table of $(k+1) \times (n+1)$ integers set to $\infty$   ▷ $P(i,j) \hat{=}$ partition of size $i$ on the first $j$ components
initialize the top row of $P$ to zero, i.e. $P(0,*) = 0$
for $i = 1$ to $k$ do
    for $j = 1$ to $m$ do
        if $|\mathscr{C}_j| > i$ then                         ▷ if the $j$th component does not fit in the given size
            continue;                                         ▷ skip the $j$th component
        end if
        $s = |\mathscr{C}_j|$
        if $j \in A_I$ then                                   ▷ then *not* taking the $j$th component for $A'$ must increase $d(A', A_I)$
            $(P(i,j), x) = \min\big(P(i,j-1) + s, P(i-s, j-1)\big)$      ▷ the minimum and its index (0 or 1)
        else                                                  ▷ else, taking the $j$th component for $A'$ must increase $d(A', A_I)$
            $(P(i,j), x) = \min\big(P(i,j-1), P(i-s, j-1) + s\big)$      ▷ the minimum and its index (0 or 1)
        end if
        if $x == 0$ then                                      ▷ the sub-solution does not take $j$th component (for $A'$)
            *pred*$(i,j) = (i, j-1)$
        else                                                  ▷ the sub-solution takes $j$th component (for $A'$)
            *pred*$(i,j) = (i-s, j-1)$
        end if
    end for
end for
$(A', B')$ = *ConstructPartition (pred)*                      ▷ compute the new partition
$A'' = \bigcup_{j \in A'} \mathscr{C}_j$
$B'' = \bigcup_{j \in B'} \mathscr{C}_j$
swap the nodes in $\mathscr{A} \cap B''$ with $\mathscr{B} \cap A''$     ▷ update clusters
}                                                             ▷ End of Rebalance()

*ConstructPartition (pred)* {
$A' = \emptyset; B' = \emptyset$                              ▷ sets for storing partitioned component IDs
$(i,j) = (k,m)$
while $(i,j) \neq (0,0)$ do
    $(i', j') = $ *pred*$(i,j)$
    if $i \neq i'$ then                                       ▷ then the $j$th component is assigned to $A'$
        $A' = A' \cup \{j\}$
    else                                                      ▷ else, the $j$th component is *not* assigned to $A'$ (i.e., assigned to $B'$)
        $B' = B' \cup \{j\}$
    end if
end while
Return $(A', B')$
}                                                             ▷ End of ConstructPartition()
```


$\mathscr{A}$ $\mathscr{B}$
$\mathbf{b_1}$, a_2, a_3, a_4, a_5, a_6, ⋮, a_{k-1}, a_k
a_1, $\mathbf{b_2}$, b_3, b_4, b_5, b_6, ⋮, b_{k-1}, b_k
$\mathbf{b_3}$, a_2, a_3, a_4, a_5, a_6, ⋮, a_{k-1}, a_k
a_1, b_2, b_1, $\mathbf{b_4}$, b_5, b_6, ⋮, b_{k-1}, b_k
$\mathbf{b_5}$, a_2, a_3, a_4, a_5, a_6, ⋮, a_{k-1}, a_k
a_1, b_2, b_1, b_4, b_3, $\mathbf{b_6}$, ⋮, b_{k-1}, b_k
$\mathbf{b_{k-1}}$, a_2, a_3, a_4, a_5, ⋮, a_{k-2}, a_{k-1}, a_k
a_1, b_2, b_1, b_4, b_3, ⋮, b_{k-2}, b_{k-3}, $\mathbf{b_k}$
a_1, a_2, a_3, a_4, ⋮, a_{k-2}, a_{k-1}, a_k
b_2, b_1, b_4, b_3, ⋮, b_{k-2}, b_{k-3}, b_k, b_{k-1}

(a) (A_I, B_I) (b) (A_1, B_1) (c) $(A_2, B_2)\ldots$ (d) $(A_{\frac{k}{2}-1}, B_{\frac{k}{2}-1})$ (e) (A_F, B_F)

Fig. 6.1 This example illustrates how PPL decides to collocate nodes on a simplistic request sequence that targets b-nodes first. It begins with the initial partition of k *red* and k *black* nodes (i.e. singleton components) into (A_I, B_I), collocates the communicating nodes (shown in bold), and converges to the final partition (A_F, B_F). Observe that in 6.1d, PPL decides to swap the components b_{k-1} and a_1 because with any other swap the new distance would be higher: $\forall A_{\frac{k}{2}} \neq A_F : d(A_{\frac{k}{2}}, A_I) > d(A_F, A_I)$. Also, OFF moves to the final partition in one swap since $d(A_F, A_I) = 2$.

Theorem 2. *The online Algorithm 2 has a competitive ratio in $O(k)$ and runs in polynomial time (per request).*

Proof. For the sake of the proof, we assume the final configuration, $\mathscr{K}_F = (A_F, B_F)$, partitions all the nodes into two types: *a-nodes* $\in A_F$ *b-nodes* $\in B_F$. Additionally, we color the nodes in *red* or *black* according to their initial location, that is, either cluster $\mathscr{A}$ or cluster $\mathscr{B}$ (Fig. 6.1). Formally,

$$a\text{-}nodes = \{a_i^R \cup a_i^B \mid a_i^{R \text{ or } B} \in A_F, a_i^R \in A_I, a_i^B \in B_I, i \in [k]\}$$

$$b\text{-}nodes = \{b_i^R \cup b_i^B \mid b_i^{R \text{ or } B} \in B_F, b_i^R \in A_I, b_i^B \in B_I, i \in [k]\}$$

Moreover, assume both PPL and OFF start with the same configuration $\mathscr{K}_I = (A_I, B_I)$:

$$A_I = \{b_1^R, \ldots, b_x^R, a_{x+1}^R, \ldots, a_k^R\}, B_I = \{a_1^B, \ldots, a_x^B, b_{x+1}^B, \ldots, b_k^B\},$$

and they aim to reach $\mathscr{K}_F = (A_F, B_F)$:

$$A_F = \{a_1^B, \ldots, a_x^B, a_{x+1}^R, \ldots, a_k^R\}, B_F = \{b_1^R, \ldots, b_x^R, b_{x+1}^B, \ldots, b_k^B\}.$$

First, observe that the final configuration $\mathscr{K}_F$ is only x swaps away from $\mathscr{K}_I$. Furthermore, PPL knows only the colors but not the node types, whereas OFF knows also the types. Therefore, OFF moves to $\mathscr{K}_F$ paying for only $2x$ migrations.

For any intermediate configuration $\mathscr{K}'$, we define the distance measure as $dist(\mathscr{K}', \mathscr{K}_I) = d(A', A_I) = |A' \triangle A_I|$. Obviously, $dist(\mathscr{K}_F, \mathscr{K}_I) = d(A_F, A_I) = 2x$.

Since any inter-cluster communication is followed by a re-balancing, we only need to bound the migration cost PPL pays over the course of all requests until it reaches $\mathscr{K}_F$. Let $PPL_{mig}(\sigma_t)$ denote the number of nodes that migrate during the re-balancing triggered by σ_t.

First, note that there are at most $2(k-1)$ calls to *rebalance()*. Because, after each occurrence, the number of components decreases by at least 1 and there are initially k components of each type which eventually collocate in two large components (of size k). Now we analyze the worst-case cost of the re-balancing upon a request σ_t, i.e. $PPL_{mig}(\sigma_t)$.

Recall that due to the lines 12-16, PPL chooses a partition (A_t, B_t) that minimizes the distance $d(A_t, A_I)$. Now we argue that $PPL_{mig}(\sigma_t) \leq 4x$. For the sake of contradiction, assume $PPL_{mig}(\sigma_t) > 4x$. This implies that during the re-balancing more than $2x$ nodes migrate to $\mathscr{A}$. Consider the color of majority among these $2x$ nodes. Clearly, the number of nodes with that color is more than x. There are two cases:

1. the majority are *black*. In this case, after the re-balancing, there are more than x *black* nodes on cluster $\mathscr{A}$ which means the same number of *red* nodes exist on cluster $\mathscr{B}$. This in turn implies that the distance is more than $2x$ after the re-balancing, i.e., $d(A_t, A_I) > 2x$. On the other hand, we know by assumption that moving to the configuration $\mathscr{K}_F$ would yield the distance exactly $2x$. Thus, the partition $(A', B') = (A_t, B_t)$ computed by Algorithm 3 is not optimal.
2. the majority are *red*. Since these *red* nodes are on $\mathscr{B}$ before re-balancing (i.e. in B_{t-1}), the same number of *black* nodes exist in A_{t-1}. Therefore, the distance before re-balancing is $d(A_{t-1}, A_I) > 2x$. Similar to the first case (but for time $t-1$), this contradicts the optimality of Algorithm 3.

Thus, we conclude $PPL_{mig}(\sigma_t) \leq 4x$ and $\sum_t PPL_{mig}(\sigma_t) \leq 4x \cdot 2(k-1)$. The competitive ratio is therefore at most $\frac{4x \cdot 2(k-1)}{2x} = 4(k-1)$.

It remains to show the polynomial runtime. It is easy to see that the running time is dominated by line (7) when PPL computes a new partition. The dynamic program computes a table of $(k+1) \cdot (2k+1) \in \Theta(k^2)$ integers. Then we trace the optimal path in the table in time $\Theta(k)$. Thus, the total computation for each request takes time in $\Theta(k^2)$. □

7 Related Work

Our paper takes a novel perspective on a range of classic problems. First, clustering and graph partitioning problems as well as repartitioning problems [22] have been studied for many years and in many contexts. These problems are usually NP-complete and even hard to approximate [2]. Especially partitioning problems for two clusters ($\ell = 2$ in our case), known as minimum bisection problems [10], have been studied intensively. Minimum bisection problems are known to allow for good, $O(\log^{1.5} n)$-factor approximations [14]. Problem variants with $k = 2$ correspond to maximum matching problems, which are polynomial-time solvable. In contrast to our work however, these models assume an offline perspective where the problem input is given ahead of time. In the online world, our problem is related to page (resp. file) migration [5,7] and server migration [6] problems: in these problems, a server needs to be migrated close to requests occurring on a graph, trading off access and migration costs. In the former problem variant, migration costs relate to distance; in the latter, migration costs relate to the available bandwidth along migration paths. Moreover, in our problem, a ski-rental resp. rent-or-buy like tradeoff between migration and communication costs needs to be found. However, migrations do not occur along a graph but between clusters, and multiple nodes can be migrated simultaneously. The large configuration space also renders solutions based on metrical task system approaches [8] inefficient. Another interesting connection exists to k-server problems [12], where multiple servers can "collaboratively" serve requests. In some sense, our problem can be seen as the opposite problem, where rather than aiming to move servers to the locations where the requests occur, we aim to move away and *avoid* configurations (i.e., cuts) where requests occur. More importantly, compared to classic online migration problems where requests define a unique optimal location from which they can be served at minimal cost (namely at the corresponding graph vertex), in our case, a request only reveals very limited information about the optimal (minimal cost) configuration. In other words, a single request only contains very limited information about how good a current clustering is, and how far (in terms of migrations) we are from an optimal offline location.

Our model can be seen as a generalization of online paging [11,15,16,21,23], and especially its variants *with bypassing* [1,9]. However, in general, in our model, the "cache" is *distributed*: requests occur *between* nodes and *not to* nodes, and costs can be saved by collocation.

Our problem also has connections to online packing problems, where items of different sizes arriving over time need to be packed into a minimal number of bins [19,20]. In contrast to these problems, however, in our case the objective is not to minimize the number of bins but rather the number of "links" between bins, given a fixed number of bins.

The paper closest to ours is [4] which studies online partitioning problems from a deterministic perspective, i.e., σ is generated in a deterministic manner. In this setting, it has been shown that the competitive ratio is inherently high, at least linear in k, and even if the online algorithm is allowed to user larger clusters than the offline algorithm (scenario with augmentation). We in this paper initiate the study of stochastic models where request patterns are drawn from an unknown but fixed distribution, and show that polylogarithmic bounds can be achieved under ring patterns, even without augmentation.

In general, we believe that a key conceptual contribution of our model itself regards the underlying combination of learning and searching. Indeed, while the fundamental problem of how to efficiently learn a distribution has been explored for many decades [18], our perspective comes with an additional locality requirement, namely that searching induces costs (i.e., migrations).

8 Conclusion

This paper initiated the study of a natural cluster learning problem where the search procedure entails costs: communication costs occur in "suboptimal" clustering configurations and migration costs occur when switching between configurations. In particular, we presented an efficient online clustering algorithm which performs well even if compared to an offline algorithm which knows the distribution of the communication pattern ahead of time. Indeed, the $O(\log k)$ competitive ratio is interesting as k is likely to be small in the applications considered in this paper: k corresponds to the number of virtual machines that can be hosted on the same server, e.g., the number of cores. Moreover, we believe that our online approach is interesting in practice as it does not rely on any assumptions on the communication distribution, which may turn out to be wrong.

We believe that our work sheds an interesting new light on multiple classic problems, and opens an interesting field for future research. In particular, it would be interesting to know whether similar competitive ratios can be achieved even for more general communication patterns. Moreover, so far we have only focused on deterministic algorithms, and the exploration of randomized algorithms constitutes another interesting avenue for future research.

Acknowledgements. Research supported by the German-Israeli Foundation for Scientific Research and Development (GIF), grant number I-1245-407.6/2014.

References

1. A. Adamaszek, A. Czumaj, M. Englert, and H. Räcke. An $O(\log k)$-competitive algorithm for generalized caching. In *Proc. 23rd SODA*, pages 1681–1689, 2012.
2. K. Andreev and H. Räcke. Balanced graph partitioning. *Theory of Computing Systems*, 39(6):929–939, 2006.
3. C. Avin, L. Cohen, and S. Schmid. Competitive clustering of stochastic communication patterns on the ring. In *Proc. 5th International Conference on Networked Systems (NETYS)*, 2017.
4. C. Avin, A. Loukas, M. Pacut, and S. Schmid. Online balanced repartitioning. In *Proc. 30th International Symposium on Distributed Computing (DISC)*, 2016.
5. Y. Bartal, M. Charikar, and P. Indyk. On page migration and other relaxed task systems. *Theoretical Computer Science*, 268(1):43–66, 2001. Also appeared in *Proc. of the 8th SODA*, pages 43–52, 1997.
6. M. Bienkowski, A. Feldmann, J. Grassler, G. Schaffrath, and S. Schmid. The wide-area virtual service migration problem: A competitive analysis approach. *IEEE/ACM Transactions on Networking (ToN)*, 2014.
7. D. L. Black and D. D. Sleator. Competitive algorithms for replication and migration problems. 1989.
8. A. Borodin, N. Linial, and M. E. Saks. An optimal on-line algorithm for metrical task system. *Journal of the ACM*, 39(4):745–763, 1992. Also appeared in *Proc. of the 19th STOC*, pages 373–382, 1987.
9. L. Epstein, C. Imreh, A. Levin, and J. Nagy-György. Online file caching with rejection penalties. *Algorithmica*, 71(2):279–306, 2015.
10. U. Feige and R. Krauthgamer. A polylogarithmic approximation of the minimum bisection. *SIAM Journal on Computing*, 31(4):1090–1118, 2002.
11. A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, 1991.
12. A. Fiat, Y. Rabani, and Y. Ravid. Competitive k-server algorithms. *J. Comput. Syst. Sci.*, 48(3):410–428, 1994.
13. W. Franck. An optimal search problem. *SIAM review*, 7(4):503–512, 1965.
14. R. Krauthgamer and U. Feige. A polylogarithmic approximation of the minimum bisection. *SIAM Review*, 48(1):99–130, 2006.
15. L. A. McGeoch and D. D. Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6(6):816–825, 1991.
16. M. Mendel and S. S. Seiden. Online companion caching. *Theoretical Computer Science*, 324(2–3):183–200, 2004.
17. M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, New York, NY, USA, 2005.
18. T. Pöschel, W. Ebeling, and H. Rosé. Guessing probability distributions from small samples. *Journal of statistical physics*, 80(5-6):1443–1452, 1995.
19. P. V. Ramanan, D. J. Brown, C. C. Lee, and D. T. Lee. On-line bin packing in linear time. *Journal of Algorithms*, 10(3):305–326, 1989.
20. S. S. Seiden. On the online bin packing problem. *Journal of the ACM*, 49(5):640–671, 2002.
21. D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.
22. L. Vaquero, F. Cuadrado, D. Logothetis, and C. Martella. Adaptive partitioning for large-scale dynamic graphs. In *Proc. 4th Annual Symposium on Cloud Computing (SOCC)*, pages 35:1–35:2, 2013.
23. N. E. Young. On-line caching as cache size varies. In *Proc. of the2ndACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 241–250, 1991.