Online Bipartite Perfect Matching With Augmentations

Kamalika Chaudhuri*, Constantinos Daskalakis†, Robert D. Kleinberg‡, and Henry Lin†
*Information Theory and Applications Center, U.C. San Diego
Email: kamalika@soe.ucsd.edu
†Division of Computer Science, U.C. Berkeley
E-mail: costis@cs.berkeley.edu, henrylin@eecs.berkeley.edu
‡Computer Science Department, Cornell University
E-mail: rdk@cs.cornell.edu

***Abstract*—In this paper, we study an online bipartite matching problem, motivated by applications in wireless communication, content delivery, and job scheduling. In our problem, we have a bipartite graph G between n clients and n servers, which represents the servers to which each client can connect. Although the edges of G are unknown at the start, we learn the graph over time, as each client arrives and requests to be matched to a server. As each client arrives, she reveals the servers to which she can connect, and the goal of the algorithm is to maintain a matching between the clients who have arrived and the servers. Assuming that G has a perfect matching which allows all clients to be matched to servers, the goal of the online algorithm is to minimize the switching cost, the total number of times a client needs to switch servers in order to maintain a matching at all times.**

Although there are no known algorithms which are guaranteed to yield switching cost better than the trivial $O(n^2)$ in the worst case, we show that the switching cost can be much lower in three natural settings. In our first result, we show that for any arbitrary graph G with a perfect matching, if the clients arrive in random order, then the total switching cost is only $O(n \log n)$ with high probability. This bound is tight, as we show an example where the switching cost is $\Omega(n \log n)$ in expectation. In our second result, we show that if each client has edges to $\Theta(\log n)$ uniformly random servers, then the total switching cost is even better; in this case, it is only $O(n)$ with high probability, and we also have a lower bound of $\Omega(n/\log n)$. In terms of the number of edges needed for each client, our result is tight, since $\Omega(\log n)$ edges are needed to guarantee a perfect matching in G with high probability. In our last result, we derive the first algorithm known to yield total cost $O(n \log n)$, given that the underlying graph G is a forest. This is the first result known to match the existing lower bound for forests, which shows that any online algorithm must have switching cost $\Omega(n \log n)$, even when G is restricted to be a forest.

I. Introduction

In this paper, we study an online bipartite matching problem, which models a scenario in which clients arrive over time and request permanent service from a set of given servers. As each client arrives, she announces a set of feasible servers capable of servicing her request, and our goal is to provide service to each client persistently by maintaining a matching at all times between clients who have arrived and servers capable of servicing their requests. We would like to assign clients to servers permanently without ever having to reassign clients to different servers, but when a new client arrives we may be forced to reassign existing clients to alternative servers to ensure that all clients can receive service. As it is often more important to provide service to all clients, the goal of our algorithm will be to maintain a matching always between arrived clients and allowed servers, while minimizing the *switching cost*, the total number of times that clients are reassigned to different servers.

Our online bipartite matching problem has a wide variety of applications spanning diverse areas, including streaming content delivery, web hosting, remote data storage, job scheduling, and hashing. We describe a few applications below, which can be modeled as an instance of the online bipartite matching problem we described above. In the following examples, we always refer to the entities requesting service as *clients* and the entities providing service as *servers* for consistency. In examples where it is not clear, we mark in parenthesis which entities are clients and which entities are servers.

- **Streaming Content Delivery, Web Hosting, and Remote Data Storage** We have a set of servers capable of streaming content online, hosting web pages, or storing data remotely. A sequence of clients arrives requesting to have their content streamed online, their web pages hosted online, or their data stored online. Due to locality, security, cost, routing policy, or other reasons, the streaming content, web page, or data from each client can only be hosted at a subset of server locations.

- **Job Scheduling** We have a set of servers with differing capabilities available to process job requests from persistent sources - jobs that need to be processed over a long or indefinite period of time (e.g. protein folding, genomic research, SETI@HOME). A sequence of persistent job requests (clients) arrive and reveal a subset of servers capable of servicing their request.
- **Hashing** We have locations in a hash table (servers) available to store data objects (clients), and a set of hash functions. Data objects arrive over time and can be assigned to a location in the hash table, if one of the hash functions maps the data object to that location.

Note that in all the examples above, it is reasonable to assume that clients can be reassigned to different servers, but at a cost. For instance, in the streaming content example, clients may not be able to access their content while their content is being transferred from one server to another. Thus, it is desirable to minimize the number of the reassignments our algorithm incurs, which causes these interruptions.

Before stating our results, we define our problem more formally. Each instance of the online bipartite matching problem is defined by:

- A bipartite graph G between clients and servers, which represents the servers to which each client can connect
- A permutation σ, which represents the sequence in which the clients arrive

The graph G is unknown at the beginning, but as clients arrive according to σ, each client reveals the servers to which she has edges. The goal of the algorithm to is to maintain a matching between arrived clients and servers at all times, while minimizing the total number of times that clients are reassigned to different servers. Alternatively, as each client arrives, our algorithm can be viewed as finding an augmenting path (a path that alternates between matched and unmatched edges), in the graph revealed so far, from the arriving client to an unmatched server. It should be clear that upper bounding the total length of the augmenting paths used by the algorithm, also upper bounds the total switching cost. As the total length of the augmenting paths is close to the switching cost, it can also be used to lower bound the switching cost in some cases as well.

For simplicity, note that we assume each server may service (or be matched to) at most one client at a time, although our results can be fully generalized to the case where servers can service multiple requests. We also assume that G consists of n clients and n servers, and G contains a perfect matching, although our results still hold essentially without these assumptions. See the Appendix for further discussion.

Although this problem was introduced over a decade ago [7], we still know surprisingly little about the optimal algorithm. For example, there is a very natural *greedy algorithm*, which provides service to each new client by using an augmenting path of minimal length, but is this greedy algorithm optimal? What is its worst-case switching cost? There are no known upper bounds to show that the greedy algorithm or any other algorithm performs better than $O(n^2)$ in the worst case, but an upper bound of $O(n^2)$ is trivial since any reasonable algorithm only switches at most $O(n)$ clients per arriving client. Furthermore, an existing lower bound only shows that the total switching cost has to be $\Omega(n \log n)$, so a large gap exists between the known upper and lower bounds.

A. Our Results

Although worst case analysis for this problem has not yielded fruitful results, we know that the worst case does not often occur in practice, and it may be reasonable to make certain assumptions about the graph G or the arrival order σ. For example, in the case of remote data storage, one might imagine that the graph G which determines the servers which can store a client's data is fixed, but perhaps the arrival order σ is random. Does the greedy algorithm perform provably better than $O(n^2)$ in this case? Moreover in some cases, it might be reasonable to assume that the graph G is generated randomly. For example, in the case of hashing, if $\Theta(\log n)$ random hash functions are chosen, then the set of edges between clients (data elements) and servers (hash locations) is a graph where each client has $\Theta(\log n)$ random edges to servers. Can better bounds be proved in this case as well?

In this paper, we show that indeed the switching cost can be much better under these conditions, despite the lack of worst case upper bounds better than $O(n^2)$. In the first case, we show that when G is chosen arbitrarily, but σ is chosen uniformly at random, then the greedy algorithm performs well and achieves $O(n \log n)$ switching cost with high probability. This bound is tight as we show that there is a distribution over graphs for which any deterministic algorithm must incur cost $\Omega(n \log n)$ in expectation, even if σ is chosen uniformly at random.

In the second case, where each client has $\Theta(\log n)$ random edges to servers, we show that the switching cost can be even better. In this case, we prove that the switching cost is $O(n)$ with high probability, and it nearly matches our lower bound of $\Omega(n/\log n)$. In terms

of the number of edges used to generate the graph G, our result is tight, since $\Omega(\log n)$ random edges are needed per node, in order to guarantee a perfect matching with high probability.

Lastly, we also make the first progress in over a decade in the original worst case model where σ is chosen adversarially, but G is known to be acyclic (i.e. a forest). In this setting, we derive a new algorithm which achieves cost $O(n \log n)$, which matches the existing lower bound of $\Omega(n \log n)$ for forests. Although the networks that occur in practice are not often acyclic, we view our last result as making progress towards finding an $O(n \log n)$ solution in the general worst case setting.

B. Related Work

The problem studied here was first introduced by Grove et al. [7] in a paper, which focused on a special case of the problem where each client has degree at most two. For this special class of instances, they prove that the greedy algorithm incurs a worst-case switching cost of $O(n \log n)$, and they give a matching lower bound by showing there are cases where any algorithm must have cost $\Omega(n \log n)$. The paper goes on to consider the case in which clients can connect *and disconnect* over time; for this problem, assuming that each client has degree at most two, they present a randomized algorithm which has a competitive ratio of $O(\sqrt{n})$, where the *competitive ratio* of an online algorithm is the ratio between the cost incurred by the online algorithm, which does not know the input sequence in advance, and the cost incurred by an optimal algorithm which does know the input sequence in advance. Previous to our work, the special case in which each client has degree at most two, was the only class of instances for which an optimal online algorithm was known.

Our problem is related to previous work on online load balancing with task preemption, which has been studied by many authors [1], [9], [10]. The main difference between our work and the previous work on load balancing is that our model assumes a hard capacity constraint on the servers and allows clients to be reassigned, while the previous load balancing work generally does not assume a hard capacity constraint on the servers, or does not allow reassignments. Without hard capacity constraints or reassignments, the goal in online load balancing is often to minimize the maximum load, or if reassignments are allowed, the goal is to minimize the number of reassignments while always maintaining a maximum load which is close to the optimal maximum load achievable.

For example, a series of works by Azar et al. [2], [3], [4] studied online load balancing *without* the possibility of task preemption (i.e. without reassigning clients, in our terminology). They established that when tasks arrive but never depart, the greedy algorithm, which assigns each task to the least-loaded admissible server, has $O(\log n)$ competitive ratio with respect to the maximum load. When tasks both arrive and depart, they show that the optimal competitive ratio is $O(\sqrt{n})$. When jobs arrive and depart and task preemption is allowed, the situation improves dramatically: Phillips and Westbrook [9] give an algorithm which always maintains a maximum load within a factor of $O(\log n)$ of optimal, while only incurring a reassignment cost of $O(m)$, where m is the number of arrivals and departures. Westbrook [10] also gives an algorithm which is $O(1)$-competitive with respect to the maximum load and with a reassignment cost of $O(m \log n)$. Andrews et al. [1] study online load balancing with reassignment costs in a model in which any client may be assigned to any server, but clients have arbitrary sizes and reassignment costs. (The same model was considered, in lesser generality, in [10].) They give an online algorithm which is 3.5981-competitive with respect to load and 6.8285-competitive with respect to reassignment cost.

Our work is also related to the recent work of Godfrey, who analyzes the load balancing properties of certain random processes which assign clients to servers [6]. Godfrey proves that only very weak conditions are needed on the random process, in order to ensure that the servers stay roughly balanced with high probability.

Our load-balancing scenario in Section III also has connections to hashing. A large body of theoretical and experimental research has focused on hashing schemes and dictionary data structures; a dictionary data-structure is a table containing items, and the goal is to design a data-structure which supports fast insertions and accesses, with a fairly high space-utilization. Typically, dictionaries are used in combination with a hashing scheme, which is a mapping from the items to their locations in the table. A common assumption in theoretical work is that such mappings are random. Under this assumption, the connection between hashing and our setting is as follows: if an item has d randomly chosen locations where it can be inserted into the table, then the problem of inserting a series of items into the table reduces to the problem of finding a matching in a random bipartite graph in an online manner. Under this setting, our algorithm is very similar to the Cuckoo Hashing scheme of Pagh and Rodler [8]. Cuckoo hashing, essentially, uses the following algorithm for inserting items to tables, or equivalently, matching new clients to servers. If there is an empty server adjacent to the client to be matched,

then the client is matched to this server; otherwise, it is matched arbitrarily to one of its adjacent servers, and a new matching is recursively found for the client which was previously attached to this server. In [8], [5], it was shown that if the degree of each node is $d = O(\ln \frac{1}{\epsilon})$, and if there are n clients and $n(1+\epsilon)$ servers, then the expected switching cost is $O(1)$, where the expectation is with respect to the randomness in the assignments. In contrast, our results imply that if $d = O(\log n)$, and if there are n clients and n servers, the switching cost is still $O(1)$, with high probability over the assignments. Thus, our results extend the results of [8], [5] to the case in which there is no surplus of servers over clients.

II. Random Arrival Order

In this section, we assume that the bipartite graph G is arbitrary, but that the clients arrive in a uniformly random order (i.e. the arrival order is uniformly distributed over all $n!$ possible orderings of the clients). Under this assumption, we prove that the natural greedy algorithm for the problem, which always augments along the shortest augmenting path available, suffers a switching cost which is $O(n \log n)$ with high probability. We also provide a lower bound demonstrating that this upper bound is the best possible in the random-ordering model.

A. Upper bound

To obtain some intuition, we start by showing the expected cost is at most $O(n \log n)$. Our analysis here is just for intuition, as we will need a more sophisticated argument to show that the cost is $O(n \log n)$ with high probability. To prove that the expected cost is $O(n \log n)$, we just need the following lemma, which upper bounds the expected cost of a client who arrives when i clients remain to arrive.

Lemma II.1 *Given that i clients remain to be connected, the expected number of servers that the greedy algorithm needs to augment in order to connect the next client is at most $\frac{n}{i}$.*

Given the lemma, it is easy to sum over all clients and bound the total expected switching cost by $\sum_{i=1}^{n} \frac{n}{i} = O(n \log n)$. We now prove the lemma.

Proof: Before we prove the lemma, let us define d_k, for $k \in \{1, \ldots, i\}$, to be the least number of servers that need to be switched if the kth remaining client arrives, out of i remaining clients. Note that when i clients remain to arrive, the expected cost of the next arriving client is $\sum_{k=1}^{i} (\frac{d_k}{i})$. Thus, if we can just show that $\sum_{k=1}^{i} d_k \leq n$, then the expected cost will then be upper bounded by $\frac{n}{i}$, and the lemma follows. To show $\sum_{k=1}^{i} d_k \leq n$, note that if one is given knowledge in advance about the remaining k clients, it is possible to find a perfect matching M between all n clients and all n servers. Furthermore, if we are given the perfect matching M, note that we can connect the i remaining clients with total switching cost $\leq n$, since we can switch the matching between clients and servers to be exactly the same as M with total cost $\leq n$. Given that the remaining i clients can be matched with $\leq n$ switches, it must be the case that there must exist i disjoint augmenting paths of total length at most n (in terms of servers augmented), which can be used to connect the remaining i clients to free servers. Since we found that we could connect the i remaining clients with i disjoint paths of total length n (in terms of servers augmented), then it must be the case that $\sum_{k=1}^{i} d_k \leq n$, since d_k is the length of the *shortest* augmenting path from the kth remaining client to a free server. Thus, the lemma follows, and the total expected cost must be $O(n \log n)$. ■

We now prove the main theorem of this section, which states the cost is bounded by $O(n \log n)$ with high probability.

Theorem II.2 *For any graph G, when the clients in G arrive in random order, the switching cost incurred by the greedy algorithm is at most $O(n \log n)$ with probability at least $1 - n^{-2}$.*

Proof: To prove the cost is $O(n \log n)$ with high probability, we couple our online matching process with a sequence of n^2 binary random variables, $X_{i,j}$ for $i, j \in \{1, \ldots, n\}$, such that $\sum_{i,j \in \{1,\ldots,n\}} X_{i,j}$ stochastically dominates the total augmentation cost. In our random process, our random variables $X_{i,j}$ have the property that $\Pr[X_{i,j} = 1] = (\frac{1}{i})$ for any $i, j \in \{1, \ldots, n\}$, although they are not completely independent. If the $X_{i,j}$ variables were completely independent, then it would be easy to apply a standard Chernoff bound to show the total cost is at most $O(n \log n)$ with high probability. In our case, we have to carefully avoid dependencies to show that $\sum_{i,j \in \{1,\ldots,n\}} X_{i,j} = O(n \log n)$ with high probability.

Before we can analyze the $X_{i,j}$ variables, we need to describe them and to define the coupling between our random online matching process and the $X_{i,j}$ variables. The main idea is to define our coupling so that for each $i \in \{1, \ldots, n\}$, the random variable $\sum_{j=1}^{n} X_{i,j}$ stochastically dominates the number servers that need to be augmented in order to connect a new arriving client, when i clients remain to be connected.

To define our coupling, when $i \in \{1, \ldots, n\}$ clients remain, let d_k represent the minimum number of servers

that need to be augmented in order to connect the kth remaining client out of i total remaining clients, for $k \in \{1, \ldots, i\}$. Recall that $\sum_{k=1}^{i} d_k \leq n$ from our previous argument. Thus, we can assign to each client $k \in \{1, \ldots, i\}$, d_k distinct binary random variables from $\{X_{i,j} \mid j \in \{1, \ldots, n\}\}$ without overlap, which can be used to represent the connection cost if the kth client arrives next. Moreover, if the kth client arrives next, then we set the d_k binary variables assigned to it to value 1, and 0 otherwise. Note that this causes each binary variable assigned to a client in $\{X_{i,j} \mid j \in \{1, \ldots, n\}\}$ to be true with probability $(\frac{1}{i})$, since each client arrives with probability exactly $(\frac{1}{i})$. For consistency, we also set variables in $\{X_{i,j} \mid j \in \{1, \ldots, n\}\}$, which are not assigned to any client, to value 1 independently at random with probability $(\frac{1}{i})$ and 0 otherwise.

As we have defined our coupling, note that $\sum_{j=1}^{n} X_{i,j}$ always dominates the number servers that need to be augmented in order to connect a new client, when i clients remain to be connected. Furthermore, $\Pr[X_{i,j} = 1] = (\frac{1}{i})$ for any $i, j \in \{1, \ldots, n\}$, and it is not hard to see that our random process defines the random variables $X_{i,j}$ in a way such that each $X_{i,j}$ variable is independent of random variables $X_{i',j}$, where $i' \in \{1, \ldots, n\} - \{i\}$. Thus, we can apply a standard Chernoff bound to conclude that for any fixed $j \in \{1, \ldots, n\}$, $\sum_{i=1}^{n} X_{i,j} = O(\log n)$ with probability at least $(1 - n^{-3})$. Then a simple union bound implies $\sum_{i,j \in \{1,\ldots,n\}} X_{i,j} = \Theta(n \log n)$ with probability at least $(1 - n^{-2})$. Therefore, the total connection cost must be at most $O(n \log n)$ with probability at least $(1 - n^{-2})$. ■

B. Lower bound

In this section, we show using a lower bound that our upper bound is essentially optimal (proof in the appendix).

Theorem II.3 *For any online algorithm, there exists a graph G such that the algorithm incurs an expected switching cost of $\Omega(n \log n)$ when the clients of G arrive in random order.*

III. Random Connection Model

In this section we study the following scenario: there is an equal number n of servers and clients, the clients arrive sequentially, and each of them selects, at arrival, a random subset of $O(\log n)$ servers. We provide an online matching protocol for this setting which incurs total switching cost of $O(n)$ with high probability, so that the average switching cost is $O(1)$ per client. Moreover, we give a lower bound, showing that any online matching protocol requires switching cost $\Omega(n/\log n)$ with high probability, which establishes that our upper bound is tight to within a factor of $O(\log n)$. Before proceeding to the details of our results, we note that, since the number of servers and clients are equal, we need to assume that every client selects $\Omega(\log n)$ servers in order to guarantee that a matching exists, after all clients arrive.

The idea of our protocol is this: When a new client arrives, the current server-client graph is examined, and a binary subtree of the breadth-first-search tree originating at the new client is chosen carefully, in order for an augmenting path to be found along the edges of that tree. Our analysis establishes that the binary tree that is explored is likely to contain a free server at constant depth, so that the augmenting path, and hence the resulting switching cost, is constant.

Theorem III.1 *Consider the following online matching problem: There are n servers, and n clients arrive sequentially, each client selecting $O(\log n)$ servers at random. There exists a client-server assignment protocol for this online arrival model which, with high probability, succeeds in matching each client with a server with overall switching cost of $O(n)$.*

Proof: Let $\mathcal{S} = \{s_1, \ldots, s_n\}$ be the set of servers and $\mathcal{C} = \{c_1, \ldots, c_n\}$ the set of clients, and let us suppose that the clients arrive in the order $c_1, c_2, \ldots, c_n$, where t, $t = 1, \ldots, n$, is the arrival time of client c_t. Without loss of generality, we assume that $n = 2^\ell$, for some integer ℓ; we also assume that every client selects a random set of $\alpha \cdot \log_2 n$ servers from the set $\mathcal{S}$ with repetition, for some sufficiently large constant $\alpha > 0$; with minor modifications in the proof, the result extends to the case where n is not a power of 2 and the clients select servers without repetition. For this model, we show that the Online Matching Protocol described in Figure 1 satisfies the following properties with high probability. (Here and throughout this section, we interpret "with high probability" to mean "with probability at least $1 - O(1/n)$.")

- at every time step $t > 0$, the clients $c_1, \ldots, c_t$ are matched with a subset of t servers;
- the total switching cost incurred by the protocol is $O(n)$.

In the description of the protocol in Figure 1 we use the following notation. We denote by $f_t : \{c_1, \ldots, c_t\} \to \mathcal{S}$ the matching of clients to servers that the protocol maintains at time t, and by $g_t : \mathcal{S} \to \{\neg\} \cup \{c_1, \ldots, c_t\}$ the "inverse matching", defined so that $g_t(s) = c$ iff $c \in \{c_1, \ldots, c_t\}$ and $f_t(c) = s$. Finally, by $\mathcal{G}_t$ we denote the bipartite graph with node set $\{c_1, \ldots, c_t\} \sqcup \mathcal{S}$ and an

edge between client c_i, $i \leq t$, and server s iff $s \in \mathcal{S}_{c_i}$, where $\mathcal{S}_{c_i}$ is the set of servers that client c_i selects.

ONLINE MATCHING PROTOCOL
At time step $t > 0$:

1) client c_t chooses a random set $\mathcal{S}_{c_t}$ of $\alpha \cdot \log_2 n$ servers from the set $\mathcal{S}$ with repetition;
2) c_t orders $\mathcal{S}_{c_t}$ arbitrarily into a list $\mathcal{L}_{c_t}$ and sets $j_{c_t} = 1$ (to be used as an index in her list of servers);
3) $\mathcal{P}$:=BINARY_BFS(c_t, f_{t-1}, g_{t-1});
   */*upon success* BINARY_BFS *returns an augmenting path on the graph $\mathcal{G}_t$ originating at the node c_t*/*

   if $\mathcal{P} \neq \emptyset$ **then** augment matching f_{t-1} along path $\mathcal{P}$; define f_t, g_t appropriately;
   else declare FAIL;

Fig. 1. High-level description of the protocol; when client c_t joins the network an augmenting path from c_t to a free server is sought; if such path is found, the current matching is augmented along this path to include client c_t.

We will make use of the following fact about the Coupon Collector Model (proof in the appendix).

Lemma III.2 (Coupon Collector Model) *Suppose that there are n types of coupons. If every coupon has one of the n types uniformly at random, then, with probability at least $1 - (2/e)^n$, after $k \cdot n$ coupons are requested at most $n/k - 1$ types of coupons are uncollected.*

To analyze the performance of the ONLINE MATCHING PROTOCOL we are going to temporarily forget the fact that every client has a list of $\alpha \cdot \log_2 n$ servers; we will pretend instead that every client has an infinite list of servers selected uniformly at random. Under this assumption, with probability 1, the protocol does not declare FAIL and every client gets matched with a server. We establish the following lemma which concludes the proof of the theorem.

Lemma III.3 *Under the assumption of infinite server-lists, the following are satisfied with high probability, i.e. with probability at least $1 - O(1/n)$:*

1) the total augmentation cost incurred by the protocol is $O(n)$;
2) no client explores more than $\alpha \cdot \log_2 n$ servers from her infinite list of servers.

Proof: To show Part 1 of the lemma, let us divide

BINARY_BFS
input: client c_t, current matching f_{t-1}, inverse matching g_{t-1};
output: augmenting path $\mathcal{P}$ on the graph $\mathcal{G}_t$ starting at node c_t, or $\emptyset$;

1) let $\sigma := \mathcal{L}_{c_t}(j_{c_t})$;
2) **if** $g_{t-1}(\sigma) = \neg$ **then** $\mathcal{P} := \langle (c_t, \sigma) \rangle$; **return** $\mathcal{P}$;
   else initialize a queue data structure $\mathcal{Q}$ containing the element $g_{t-1}(\sigma)$;
3) **while** $\mathcal{Q} \neq \emptyset$
   a) $c := deQueue(Q)$;
   b) **for** $r = 1, 2$
      if $j_c < \alpha \log_2 n$ **then**
      i) $\sigma := \mathcal{L}_c(j_c)$; $j_c := j_c + 1$;
      ii) **if** $g_{t-1}(\sigma) = \neg$ **then**
          let $\mathcal{P}$ be the path from node c_t to node σ on the BFS tree created by the process;
          return $\mathcal{P}$;
      else push $g_{t-1}(\sigma)$ into $\mathcal{Q}$;
   else declare FAIL;

Fig. 2. When a client c_t joins the network, process BINARY_BFS explores the graph $\mathcal{G}_t$ in a Breadth-First-Search fashion in order to find an augmenting path from c_t to a free server. However, each time a client node is encountered by the BFS process, only two of its adjacent edges are explored; moreover, the process has memory: the next time that the same client is encountered a different pair of edges will get explored.

the arrival times of the clients into $\log_2 n + 1 \equiv \ell + 1$ progressively smaller intervals $\mathcal{I}_j = \{b_j, \ldots, e_j\}$, $j = 1, \ldots, \log_2 n + 1$, where each interval j has $\lceil \frac{n}{2^j} \rceil$ time steps. In particular, we define:

- $b_j = n - n/2^{j-1} + 1$, for all $j = 1, \ldots, \log_2 n + 1$;
- $e_j = n - \lfloor n/2^j \rfloor$, for all $j = 1, \ldots, \log_2 n + 1$;

Let T_i be the BFS-tree constructed by the BINARY_ BFS process when client c_i arrives and denote by $|V_i|$ the number of client nodes in T_i. We show the following lemma.

Lemma III.4 *Under the infinite server-lists assumption, with high probability over the random choices of servers by the clients,*

$$\textit{for all } j \in \{1, \ldots, \log_2 n + 1\} : \sum_{i \in \mathcal{I}_j} |V_i| \leq 2^j n.$$

Proof: Let us fix any $j \in \{1, \ldots, \log_2 n + 1\}$. Let K_j be the number of times Steps 1 and 3(b)i of BINARY_BFS are invoked between the arrival of client c_{b_j} and until client c_{e_j} is matched with a server. Observe

that, every time the Steps 1 and 3(b)i of BINARY_BFS are invoked, the identity of server σ is independent of the identities of the servers revealed in the preceding invocations of Steps 1 and 3(b)i of BINARY_BFS. Also, observe that, if the number of distinct servers that the invocations of Steps 1 and 3(b)i of BINARY_BFS have returned over the course of the protocol is at least $\lceil n - \frac{n}{2^j} \rceil$, then the clients of the set $\{c_i\}_{i \in \cup_{t \leq j} \mathcal{I}_t}$ have all been matched with servers. From Lemma III.2 it follows that, with probability at least $1 - (2/e)^n$, after $2^j n$ invocations of Steps 1 and 3(b)i of BINARY_BFS, $\lceil n - \frac{n}{2^j} \rceil$ distinct servers will be discovered. Hence, with probability at least $1 - (2/e)^n$,

$$K_j \leq 2^j n. \tag{1}$$

It is easy to see that

$$\sum_{i \in \mathcal{I}_j} |V_i| \leq 2 \left\lceil \frac{n}{2^j} \right\rceil + \frac{K_j - 2 \left\lceil \frac{n}{2^j} \right\rceil}{2}.$$

which by (1) implies

$$\sum_{i \in \mathcal{I}_j} |V_i| \leq \left\lceil \frac{n}{2^j} \right\rceil + 2^{j-1} n \leq 2^j n.$$

The result follows from a union bound. ■

From Lemma III.4 it follows that, for all $j = 1, \ldots, \log_2 n + 1$,

$$\frac{1}{|\mathcal{I}_j|} \sum_{i \in \mathcal{I}_j} |V_i| \leq \frac{1}{|\mathcal{I}_j|} 2^j n \leq 2^{2j}$$

$$\Rightarrow \log_2 \left\{ \frac{1}{|\mathcal{I}_j|} \sum_{i \in \mathcal{I}_j} |V_i| \right\} \leq 2j. \tag{2}$$

By Jensen's inequality and the concavity of $\log_2$ it follows that

$$\log_2 \left\{ \frac{1}{|\mathcal{I}_j|} \sum_{i \in \mathcal{I}_j} |V_i| \right\} \geq \frac{1}{|\mathcal{I}_j|} \sum_{i \in \mathcal{I}_j} \log_2 |V_i|.$$

By combining the above with (2), it follows that

$$\sum_{i \in \mathcal{I}_j} \log_2 |V_i| \leq 2j \left\lceil \frac{n}{2^j} \right\rceil,$$

and summing over j it follows that

$$\sum_{i=1}^{n} \log_2 |V_i| \leq n \sum_{j=1}^{\log_2 n} \frac{2j}{2^j} + 2(\log_2 n + 1) = O(n).$$

Finally, observe that, when a client c_i joins the network, the augmenting path chosen by the protocol has length $O(1 + \log_2 |V_i|)$. It follows that the total augmentation cost paid by the protocol is $O(n)$.

To prove part 2 of the lemma, we make use of the following well known facts.

Lemma III.5 (Coupon Collector Model) *In a coupon collector model with n coupon types, the probability that all coupons are not collected after $2n \log_e n$ steps is at most $\frac{1}{n}$.*

Lemma III.6 (Balls in Bins) *In the balls and bins model, if m balls are thrown into n bins, then, with probability at least $1 - \frac{1}{n}$, the maximum load of a bin is $O(\frac{m}{n} + \log n)$.*

From Lemma III.5, it follows that, with probability at least $1 - \frac{1}{n}$, the total number of invocations of Steps 1 and 3(b)i of BINARY_BFS throughout the course of the protocol is at most $2n \log_e n$. Since every client participates exactly once in an invocation of Step 1, to conclude the proof of Part 2, it is enough to show that, with high probability, no client c participates more than $O(\log n)$ times in an invocation of Step 3(b)i. We argue this by coupling the process of selecting clients for invoking Step 3(b)i with the process of throwing $m = n \log n$ balls into n bins. For our purposes the bins are the clients and a ball, thrown at the time of an invocation of a Step 1 or a Step 3(b)i of BINARY_BFS, is received by a client (bin) c if the server selected by the invocation is matched with client c; as a result of this receipt, c will be the next client to participate in an invocation of Step 3(b)i, at which time another ball will be thrown, etc. Note that a ball might not be received by any client; in particular, for any ball that gets thrown the probability that a client c receives it is at most $\frac{1}{n}$. Hence, the maximum load that a client receives in our model is dominated by the maximum load of a bin in a balls in bins process whereby $2n \log_e$ 'n balls are thrown into n bins; the latter is $O(\log n)$ with probability at least $1 - \frac{1}{n}$ by Lemma III.6. This concludes the proof of part 2 of the lemma. ■

■

We establish next a lower bound of $\Omega(n/\log n)$ on the switching cost of any online protocol. Hence, the protocol described in the proof of Theorem III.1 is optimal up to a factor of $O(\log n)$. The proof is postponed to the appendix.

Theorem III.7 *In the setting of Theorem III.1, any online matching protocol has switching cost of $\Omega(n/\log n)$, with high probability.*

IV. Online Bipartite Matching on Forests

In this section, we provide an algorithm that achieves a switching cost of $O(n \log n)$, when the connection graph on n clients and n servers is a forest, and contains a perfect matching between clients and servers. The main result of this section is as follows.

Theorem IV.1 *Let G be the underlying graph of connections between clients and servers. If G is a forest, then, for any arrival order of clients, there is an algorithm which has a switching cost of $O(n \log n)$.*

Before we describe the algorithm, we first make some preliminary observations, which provide a foundation for stating our algorithm. Note that at the start of our online process, we have n server nodes and no edges, and thus our connection graph contains n connected components consisting of a single server node each. As new clients arrive with their edges, these connected components become merged. In particular, one should note that when a new client i arrives with $d_i \geq 2$ edges, the client's d_i edges must connect to d_i *distinct* components in the graph, since our final graph must be a forest. Thus a client arriving with $d_i \geq 2$ edges causes d_i components to merge into a single connected component.

As our process proceeds, our online algorithm monitors these connected components, and designates a node from each connected component to be the *root* of the component. The root and size of each component are key elements in deciding the augmenting path to be used to connect a new client. At the beginning of the process, each server node starts as the root node of its own connected component; this will change as clients arrive and components merge. Now to define how our algorithm maintains the root nodes and selects augmenting paths to connect each new arriving client i, we have three different cases:

- **Case** I: If client i has a single edge, then this edge connects to a single connected component C and thus all potential augmenting paths from i must pass through component C. Among the set of potential augmenting paths, choose the augmenting path that stays furthest away from the root of C.
- If a client i has more than one edge, then we have two cases to consider, depending on the set of connected components to which client i can find an augmenting path. Let S_i denote the set of connected components to which client i can find an augmenting path, and let T_i denote the set of connected components to which client i has an edge.
  - **Case** II: If the set S_i contains only one connected component, and that connected component is the unique largest component of T_i, then follow the rule in case I.
  - **Case** III: Otherwise, choose any augmenting path into the smallest component in S_i.

  At the end of both case II and case III, all components in T_i have merged into a single component T. Assign T to have the same root node as the largest component in T_i, breaking ties arbitrarily if needed.

We now prove Theorem IV.1, by bounding the total switching cost of our algorithm.

Proof of Theorem IV.1: We first bound the total cost of augmentations arising from case III by $O(n \log n)$. Note that each time a server switches clients via case III, its component size at least doubles, and thus each server can switch clients at most $O(\log n)$ times via case III augmentations. Therefore, the total cost arising from case III augmentations is $O(n \log n)$.

Lemma IV.2 shows that the switching cost arising from case I and case II augmentations is at most $O(n \log n)$. The theorem thus follows by combining Lemma IV.2 with our bound on the total cost of augmentations arising from case III. ■

We now bound the switching cost arising from case I and case II augmentations. Here we need to do a bit more work (see proof in the appendix).

Lemma IV.2 *Let G be the underlying graph of connections between clients and servers. If G is a forest, then, for any arrival order of clients, the total switching cost of case I and case II augmentations is at most $O(n \log n)$.*

V. Conclusion

In conclusion, we study three variants of the online bipartite matching problem. First, we show that when the underlying bipartite graph is arbitrary, and the clients arrive in a random order, the greedy algorithm, which always uses the shortest available switching path, achieves a switching-cost of $O(n \log n)$ with high probability. We also study the problem when the arrival order is adversarial and the underlying graph is a forest, and provide a $O(n \log n)$ algorithm for this case. Finally, we study the problem in random bipartite graphs of degree $O(\log n)$ and show an algorithm which achieves $O(n)$ switching-cost.

The main open question is to find an algorithm which achieves a switching cost of $O(n \log n)$ for any arbitrary bipartite graph between clients and servers, and for an

adversarial arrival order of clients. In particular, it would be interesting to find a proof that the greedy algorithm achieves a switching-cost of $O(n \log n)$ in any bipartite graph, for any arrival order of the clients, or to find a counterexample that suggests otherwise.

REFERENCES

[1] Matthew Andrews, Michel X. Goemans, and Lisa Zhang. Improved bounds for on-line load balancing. In *Computing and Combinatorics, Second Annual International Conference (COCOON)*, pages 1–10, 1996.

[2] Yossi Azar, Andrei Broder, and Anna Karlin. On-line load balancing. In *Proc. 33rd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 218–225, 1993.

[3] Yossi Azar, Bala Kalyanasundaram, Serge Plotkin, Kirk Pruhs, and Orli Waarts. Online load balancing of temporary tasks. In *Proc. 2nd International Workshop on Algorithms and Data Structures (WADS)*, pages 119–130, 1993.

[4] Yossi Azar, Joseph Naor, and Raphael Rom. The competitiveness of on-line assignments. In *Proc. 3rd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 203–210, 1992.

[5] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. Space efficient hash tables with worst case constant access times. In *Proc. 20th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 271–282, 2003.

[6] Brighten Godfrey. Balls and bins with structure: Balanced allocations on hypergraphs. In *Proc. 19th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2008.

[7] Edward F. Grove, Ming-Yang Kao, P. Krishnan, and Jeffrey Scott Vitter. Online perfect matching and mobile computing. In *Proc. 4th International Workshop on Algorithms and Data Structures (WADS)*, pages 194–205, 1995.

[8] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *Proc. 9th Annual European Symposium on Algorithms (ESA)*, pages 121–133, 2001.

[9] Steven Phillips and Jeffrey Westbrook. Online load balancing and network flow. In *Proc. 25th ACM Symposium on Theory of Computing (STOC)*, pages 402–411, 1993.

[10] Jeffrey Westbrook. Load balancing for response time. In *Proc. 3rd Annual European Symposium on Algorithms (ESA)*, pages 355–368, 1995.

APPENDIX

A. Discussion

Although we assume the total number of arriving clients is the same as the total number of servers, it is not difficult to see our proofs still hold if n clients arrive and there are $m > n$ servers. We omit formal details, but the bounds on the switching cost are the same for this case when measuring the switching costs relative to n, the number of clients.

We also assume that each server may service (i.e. be matched to) at most one client, but one should note that our result also holds when servers can service multiple clients. In this more general setting, when n clients are to be matched to m servers who can service $s_1, s_2, \ldots, s_m$ clients each, our bounds also hold and are exactly the same as the unit capacity case, when measuring the cost relative to n the number of clients. The bounds from the unit capacity case also apply here, since we can reduce this more general problem to the unit capacity case by creating s_i server nodes for each server $i \in \{1, \ldots, m\}$. By treating each server i as s_i unit capacity servers, and creating edges appropriately, it is not hard to see that any bounds for the unit capacity case also yield equivalent bounds for the general capacity case in terms of n the number of clients that arrive.

Lastly, we assume that there is a matching at each step of the algorithm. We make this assumption because if a new client i arrives and no matching exists between the arrived clients and the servers, then we might as well ignore client i. We feel no remorse ignoring client i, since no algorithm could have matched i and the other arrived clients. By ignoring the clients who we cannot possibly serve (without disconnecting other clients), we are then left with an instance where at each step a matching exists between the arrived clients and the existing servers.

B. Skipped Proofs

In this section, we present the details of some of the proofs from previous sections of the paper.

Proof of Theorem II.3: Using Yao's lemma, it suffices to specify a distribution on input instances such that for any deterministic online algorithm, the expected switching cost incurred by the algorithm on one of these instances is $\Omega(n \log n)$. (Here, the expectation is over the random choice of instance and the random ordering of clients.) We define our instance distribution as follows: form a random bipartite graph between n clients and n servers by choosing a random permutation π of $\{1, 2, \ldots, n\}$ and matching client i to servers $\pi(i)$ and $\pi(i+1)$, where

$\pi(n+1)$ is interpreted to mean $\pi(1)$. (In other words, the bipartite graph is a random $2n$-cycle on the set of clients and servers.)

This input distribution is invariant under permutations of the clients, so we may assume without loss of generality that the clients' arrival order is $1, 2, \ldots, n$. At the arrival time of client k, the servers belong to $n-k+1$ different connected components (including isolated servers) and each of these connected components is a path. Client k is adjacent to endpoints of two of these paths, namely, the arcs of the cycle which extend from client k to the first higher-numbered client encountered when going around the cycle in either direction. The theorem now follows from a sequence of observations outlined below.

1) Conditioning on the cyclic ordering of all clients besides k, client k is equally likely to be spliced anywhere into the cycle. In particular, it has probability $1/3$ of being spliced into the middle third of an arc between higher-numbered clients.
2) The average length of the arcs between higher-numbered clients is $(n-1)/(n-k)$, so the expected distance from client k to the nearest higher-numbered client is at least $n/3k$.
3) The input distribution is invariant under the operation of cutting out an arc of the cycle (with servers at both endpoints) and reattaching it backwards. Therefore, conditional on the states of the two paths to which client $k < n$ is connected at the time of its arrival (i.e., the sets of clients and servers in those paths and the current matching between them), each of the ways for k to connect to these paths is equally likely. (Client k connects to these two paths by choosing an endpoint of each. Thus, the number of ways for k to connect to the two paths is the product of the number of endpoints of the paths. As each path has either 1 or 2 endpoints, this number is at most 4.) Thus, conditional on the states of the two paths at the time of client k's arrival, with probability $1/4$ client k connects to each of these paths at the endpoint which is furthest from the free server on that path. If so, the switching cost is bounded below by half the length of the shorter path.
4) Combining (2) and (3), we find that the expected cost incurred at the arrival time of client k is at least $\frac{n}{24k}$. Summing over k, we get the stated lower bound.

■

Proof of Lemma III.2: Let S be any set of $n - \frac{n}{k} = \left(1 - \frac{1}{k}\right) n$ coupon types. Then, if $k \cdot n$ coupons are requested,

$$\Pr[\text{all } k \cdot n \text{ coupons are from the set of types } S] \leq \left(1 - \frac{1}{k}\right)^{kn} \leq e^{-n}.$$

It follows that, if $k \cdot n$ coupons are requested, then

$$\begin{aligned}\Pr\left[\text{at least } \frac{n}{k} \text{ coupons are uncollected}\right] &\leq \sum_{S, |S| = \left(1-\frac{1}{k}\right)n} \Pr\left[\begin{array}{l}\text{all } k \cdot n \text{ coupons are from} \\ \text{the set of types } S\end{array}\right] \\ &\leq \left(\frac{2}{e}\right)^n,\end{aligned}$$

where the summation ranges over all sets S of coupon types of size $|S| = \left(1 - \frac{1}{k}\right) n$. ■

Proof of Theorem III.7: Let $\mathcal{S} = \{s_1, \ldots, s_n\}$ be the set of servers and $\mathcal{C} = \{c_1, \ldots, c_n\}$ the set of clients, and let us suppose that the clients arrive in the order $c_1, c_2, \ldots, c_n$, where t, $t = 1, \ldots, n$, is the arrival time of client c_t. Suppose also that every client selects a random set of $D := \alpha \cdot \log_2 n$ servers with repetition; as in the proof of Theorem III.1, with minor modifications the argument extends to the case where the selection happens without repetition. Denoting by $\mathcal{S}_{c_t}$ the set of servers that client c_t chooses, let us define the following collection of events for $t = 1, \ldots, n$:

$$\mathcal{A}_t := \begin{array}{l}\text{"When client } c_t \text{ arrives all servers in } \mathcal{S}_{c_t} \\ \text{are occupied by other clients."}\end{array}$$

Recall that an online matching protocol needs to maintain a matching of the clients with servers at all times. Hence, if the event $\mathcal{A}_t$ happens, the protocol must incur an extra cost of at least 1 in period t in order to service client c_t. Moreover, observe that

$$\Pr[\mathcal{A}_t] = \left(\frac{t-1}{n}\right)^D.$$

Therefore, the expected switching cost incurred by the protocol is

$$\begin{aligned}\mathbb{E}[\text{switching cost}] &\geq \sum_{t=1}^{n} \left(\frac{t-1}{n}\right)^D = \frac{1}{n^D} \sum_{t=1}^{n-1} t^D \\ &\geq \frac{1}{n^D} \int_{t=0}^{n-2} t^D dt = \frac{1}{n^D} \frac{(n-2)^{D+1}}{D+1} \\ &= \frac{(n-2)^D}{n^D} \frac{n-2}{D+1} = \Omega\left(\frac{n}{\log n}\right).\end{aligned}$$

So, in expectation, every online matching protocol has cost $\Omega(n/\log n)$. To show that this is also true with high

probability, note that the events $\{\mathcal{A}_t\}_{t=1}^{n}$ are independent. The result follows from an easy Chernoff bound. ■

Proof of Lemma IV.2: Note that for this lemma, we do not need to distinguish between case I and II augmentations; we just need to note that these augmentations seek to maximize the distance from the root node, and for servers involved in case I or case II augmentations, their root node does not change as a result of these augmentations.

Although we do not need to distinguish between these two types of augmentations anymore, we do need to further classify the client/server switches that arise as a result of these types of augmentations. For a server v which switches clients as a result of a case I or II augmentation, we classify v's switch as either an *upward* switch, a *downward* switch, or a *peak* switch, depending on the direction of the augmentation/switch relative to the root node. To classify these switches, suppose that a case I or case II augmenting path passes through a client u, a server v, and a client w in sequence and thus results in edge (u, v) being added to the matching and edge (w, v) being deleted from it. We say node v experiences an *upward* switch if client u is strictly closer to the component root than client w, and v experiences a *downward* switch if client u is strictly further from the component root than client w. If both client u and w are the same distance from the root, then server v experiences a *peak* switch. To complete the proof of the cost upper bound, we prove the following three statements:

- Each node experiences at most one downward switch.
- Each node experiences at most $O(\log n)$ upward switches.
- There are at most $O(n)$ peak switches in total.

Before we prove the three statements, we first define some terminology. For a set U of clients and servers, we say that U is *finished* if every client in U has already arrived, U contains an equal number of clients and servers, and no client in U is adjacent to a server in the complement of U. We say that a client or server is finished if there exists a finished set U that contains the client or server. Note that once a set U becomes finished it remains finished in the future. Also, all the clients in a finished set U must be matched to servers in U and there can be no augmenting paths passing through a node of U.

Now to prove the first statement, suppose that a server node v engages in a downward switch which results in its becoming connected to a client u. After this switch takes place, let U be the set consisting of v, u, and all the clients and servers reachable from u by an augmenting path that does not pass through v. It must be the case that every server in U is matched to a client in U, because otherwise u would have an augmenting path that stays strictly further from the root than any path passing through v, and therefore u would not have switched to v in the most recent step of the algorithm. Moreover, every client in U is matched to a server in U and no client in U is adjacent to a server $v' \notin U$, because this would imply the existence of an augmenting path from u to v' that does not pass through v, contradicting the assumption that $v' \notin U$. Thus, the set U is finished; in particular, this means v is finished, thus confirming that v can engage in at most one downward switch throughout the execution of the algorithm.

To prove the second statement, note that once a server v engages in an upward switch, if it engages in another switch before its component root changes then this switch must be a downward switch, which would finish v. Thus, the number of upward switches involving v is bounded above by the number of times the root of the component containing v changes. However, a server's root node can change at most $O(\log n)$ times, since the component size at least doubles each time the root changes. As a result, we have that each server may undergo at most $O(\log n)$ upward switches.

To prove that there are at most $O(n)$ peak switches, note that each augmentation of case I or case II, consists of a sequence of zero or more upward switches, a peak switch, and a sequence of zero or more downward switches. Thus, the second statement follows since each augmentation contains at most one peak switch and since there are at most n augmentations.

Thus, we have proven all three statements, and we have that the total cost of case I and II augmentations is $O(n \log n)$. ■