Downloaded from orbit.dtu.dk on: Feb 14, 2025

DTU Library

Online Bipartite Matching with Amortized O(log^2 n) Replacements

Bernstein, Aaron; Holm, Jacob; Rotenberg, Eva

Published in:
Journal of the ACM

Link to article, DOI:
10.1145/3344999

Publication date:
2019

Document Version
Peer reviewed version

Link back to DTU Orbit

Citation (APA):
Bernstein, A., Holm, J., & Rotenberg, E. (2019). Online Bipartite Matching with Amortized O(log^2 n) Replacements. *Journal of the ACM*, *66*(5), Article 37. https://doi.org/10.1145/3344999

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Online Bipartite Matching with Amortized $O(\log^2 n)$ Replacements*

AARON BERNSTEIN, Rutgers University, bernstei@gmail.com
JACOB HOLM†, BARC, University of Copenhagen, jaho@di.ku.dk
EVA ROTENBERG‡, Technical University of Denmark, erot@dtu.dk

In the online bipartite matching problem with replacements, all the vertices on one side of the bipartition are given, and the vertices on the other side arrive one by one with all their incident edges. The goal is to maintain a maximum matching while minimizing the number of changes (replacements) to the matching. We show that the greedy algorithm that always takes the shortest augmenting path from the newly inserted vertex (denoted the SAP protocol) uses at most amortized $O(\log^2 n)$ replacements per insertion, where n is the total number of vertices inserted. This is the first analysis to achieve a polylogarithmic number of replacements for *any* replacement strategy, almost matching the $\Omega(\log n)$ lower bound. The previous best strategy known achieved amortized $O(\sqrt{n})$ replacements [Bosek, Leniowski, Sankowski, Zych, FOCS 2014]. For the SAP protocol in particular, nothing better than the trivial $O(n)$ bound was known except in special cases. Our analysis immediately implies the same upper bound of $O(\log^2 n)$ reassignments for the capacitated assignment problem, where each vertex on the static side of the bipartition is initialized with the capacity to serve a number of vertices.

We also analyze the problem of minimizing the maximum server load. We show that if the final graph has maximum server load L, then the SAP protocol makes amortized $O(\min\{L \log^2 n, \sqrt{n} \log n\})$ reassignments. We also show that this is close to tight because $\Omega(\min\{L, \sqrt{n}\})$ reassignments can be necessary.

CCS Concepts: • **Theory of computation → Graph algorithms analysis**; **Online algorithms**;

Additional Key Words and Phrases: Online algorithms, maximum matching, load balancing, shortest augmenting path, bipartite graphs

ACM Reference Format:
Aaron Bernstein, Jacob Holm, and Eva Rotenberg. 2019. Online Bipartite Matching with Amortized $O(\log^2 n)$ Replacements. *J. ACM* 1, 1, Article 1 (January 2019), 23 pages. https://doi.org/10.1145/3344999

1 INTRODUCTION

In the online bipartite matching problem, the vertices on one side are given in advance (we call these the servers S), while the vertices on the other side (the clients C) arrive one at a time with all their incident edges. In the standard online model the arriving client can only be matched

*This journal paper is the full version of a conference paper of SODA'18, where it won the best paper award. It differs from the extended abstract by the content of sections 3.2.1 and 6.3.

†This research is partially supported by grant DFF-0602-02499B from the Danish Council for Independent Research, and grant 16582, Basic Algorithms Research Copenhagen (BARC), from the VILLUM foundation.

‡This research was partly conducted during the third author's time as a PhD student at University of Copenhagen (DIKU).

Authors' addresses: Aaron Bernstein Rutgers University, bernstei@gmail.com; Jacob Holm BARC, University of Copenhagen, jaho@di.ku.dk; Eva Rotenberg Technical University of Denmark, erot@dtu.dk.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2019 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
0004-5411/2019/1-ART1 $15.00
https://doi.org/10.1145/3344999

immediately upon arrival, and the matching cannot be changed later. Because of this irreversibility, the final matching might not be maximum; no algorithm can guarantee better than a $(1 - 1/e)$-approximation [22]. But in many settings the irreversibility assumption is too strict: rematching a client is expensive but not impossible. This motivates the online bipartite matching problem with replacements, where the goal is to at all times match as many clients as possible, while minimizing the number of changes to the matching. Applications include hashing, job scheduling, web hosting, streaming content delivery, and data storage; see [8] for more details.

In several of the applications above, a server can serve multiple clients, which raises the question of online bipartite *assignment* with reassignments. There are two ways of modeling this:

Capacitated assignments. Each server s comes with the capacity to serve some number of clients $u(s)$, where each $u(s)$ is given in advance. Clients should be assigned to a server, and at no times should the capacity of a server be exceeded. There exists an easy reduction showing that this problem is equivalent to online matching with replacements [2]. A more formal description is given in Section 6.1.

Minimize max load. There is no limit on the number of clients a server can serve, but we want to (at all times) distribute the clients as "fairly" as possible, while still serving all the clients. Defining the load on a server as the number of clients assigned to it, the task is to, at all times, minimize the maximum server load — with as few reassignments as possible. A more formal description is given in Section 6.2

While the primary goal is to minimize the number of replacements, special emphasis has been placed on analyzing the *SAP* protocol in particular, which always augments down a shortest augmenting path from the newly arrived client to a free server (breaking ties arbitrarily). This is the most natural replacement strategy, and shortest augmenting paths are already of great interest in graph algorithms: they occur for example in Dinitz' and Edmonds and Karp's algorithm for maximum flow [9, 10], and in Hopcroft and Karp's algorithm for maximum matching in bipartite graphs [19].

Throughout the rest of the paper, we let n be the number of clients in the final graph, and we consider the *total* number of replacements during the entire sequence of insertions; this is exactly n times the amortized number of replacements. The reason for studying the vertex-arrival model (where each client arrives with all its incident edges) instead of the (perhaps more natural) edge-arrival model is the existence of a trivial lower bound of $\Omega(n^2)$ total replacements in this model: Start with a single edge, and maintaining at all times that the current graph is a path, add edges to alternating sides of the path. Every pair of insertions cause the entire path to be augmented, leading to a total of $\sum_{i=1}^{n/2} i \in \Omega(n^2)$ replacements.

1.1 Previous work

For static bipartite graphs, the best algorithms known for computing a maximum cardinality matching are the $O(m\sqrt{n})$ result by Hopcroft and Karp [19], and the $O(m^{10/7})$ result by Madry [25].

The problem of online bipartite matchings with replacements was introduced in 1995 by Grove, Kao, Krishnan, and Vitter [13], who showed matching upper and lower bounds of $\Theta(n \log n)$ replacements for the case where each client has degree two. In 2009, Chaudhuri, Daskalakis, Kleinberg, and Lin [8] showed that for any arbitrary underlying bipartite graph, if the client vertices arrive in a random order, the expected number of replacements (in their terminology, the *switching cost*) is $\Theta(n \log n)$ using SAP, which they also show is tight. They also show that if the bipartite graph remains a forest, there exists an algorithm (not SAP) with $O(n \log n)$ replacements, and a matching lower bound. Bosek, Leniowski, Sankowski and Zych later analyzed the SAP protocol for forests, giving an upper bound of $O(n \log^2 n)$ replacements [6], later improved to the optimal

$O(n \log n)$ total replacements [7]. For general bipartite graphs, no analysis of SAP is known that shows better than the trivial $O(n^2)$ total replacements. Bosek et al. [5] showed a different algorithm that achieves a total of $O(n\sqrt{n})$ replacements. They also show how to implement this algorithm in total time $O(m\sqrt{n})$, which (for $m \in \Omega(n^{7/6})$) matches the best performing combinatorial algorithm for computing a maximum matching in a static bipartite graph (Hopcroft and Karp [19]).

The lower bound of $\Omega(\log n)$ by Grove et al. [13] has not been improved since, and is conjectured by Chaudhuri et al. [8] to be tight, even for SAP, in the general case. We make significant progress towards closing that conjecture.

For the problem of minimizing the maximum load, [15] and [2] showed an approximation solution: with only $O(1)$ amortized changes per client insertion they maintain an assignment $\mathcal{A}$ such that at all times the maximum load is within a factor of 8 of optimum.

The model of online algorithms with replacements – alternatively referred to as online algorithms with recourse – has also been studied for a variety of problems other than matching. The model is similar to that of online algorithms, except that instead of trying to maintain the best possible approximation without making any changes, the goal is to maintain an optimal solution while making as few changes to the solution as possible. This model encapsulates settings in which changes to the solution are possible but expensive. The model originally goes back to online Steiner trees [20], and there have been several recent improvements for online Steiner tree with recourse [14, 17, 24, 26]. There are many papers on online scheduling that try to minimize the number of job reassignments [1, 11, 27, 28, 30, 32]. The model has also been studied in the context of flows [15, 32], and there is a very recent result on online set cover with recourse [16].

1.2 Our results

Theorem 1. *SAP makes at most $O(n \log^2 n)$ total replacements when n clients are added.*

This is a huge improvement of the $O(n\sqrt{n})$ bound by [5], and is only a log factor from the lower bound of $\Omega(n \log n)$ by [13]. It is also a huge improvement of the analysis of SAP; previously no better upper bound than $O(n^2)$ replacements for SAP was known. To attain the result we develop a new tool for analyzing matching-related properties of graphs (the balanced flow in Sections 3 and 4) that is quite general, and that we believe may be of independent interest.

Although SAP is an obvious way of serving the clients as they come, it does not immediately allow for an efficient implementation. Finding an augmenting path may take up to $O(m)$ time, where m denotes the total number of edges in the final graph. Thus, the naive implementation takes $O(mn)$ total time. However, short augmenting paths can be found substantially faster, and using the new analytical tools developed in this paper, we are able to exploit this in a data structure that finds the augmenting paths efficiently:

Theorem 2. *There is an implementation of the SAP protocol that runs in total time $O(m\sqrt{n}\sqrt{\log n})$.*

Note that this is only an $O(\sqrt{\log n})$ factor from the offline algorithm of Hopcroft and Karp [19]. This offline algorithm had previously been matched in the online setting by the algorithm of Bosek et al. [5], which has total running time $O(m\sqrt{n})$. Our result has the advantage of combining multiple desired properties in a single algorithm: few replacements ($O(n \log^2(n))$ vs. $O(n^{1.5})$ in [5]), fast implementation ($O(m\sqrt{n}\sqrt{\log n})$ vs. $O(m\sqrt{n})$ in [5]), and the most natural augmentation protocol (shortest augmenting path).

Extending our result to the case where each server can have multiple clients, we use that the capacitated assignment problem is equivalent to that of matching (see Section 6.1) to obtain:

Theorem 3. *SAP uses at most $O(n \log^2 n)$ reassignments for the capacitated assignment problem, where n is the number of clients.*

In the case where we wish to minimize the maximum load, such a small number of total reassignments is not possible. Let $\textsc{opt}(G)$ denote the minimum possible maximum load in graph G. We present a lower bound showing that when $\textsc{opt}(G) = L$ we may need as many as $\Omega(nL)$ reassignments, as well as a nearly matching upper bound (see Section 6.2).

Theorem 4. *For any positive integers n and $L \leq \sqrt{n/2}$ divisible by 4 there exists a graph $G = (C \cup S, E)$ with $|C| = n$ and $\textsc{opt}(G) = L$, along with an ordering in which the clients in C are inserted, such that any algorithm for the exact online assignment problem of minimizing maximum load requires a total of $\Omega(nL)$ changes. This lower bound holds even if the algorithm knows the entire graph G in advance, as well as the order in which the clients are inserted.*

Theorem 5. *Let C be the set of all clients inserted, let $n = |C|$, and let $L = \textsc{opt}(G)$ be the minimum possible maximum load in the final graph $G = (C \cup S, E)$. SAP at all times maintains an optimal assignment while making a total of $O(n \min\left\{L \log^2 n, \sqrt{n} \log n\right\})$ reassignments.*

1.3 High level overview of techniques

Consider the standard setting in which we are given the entire graph from the beginning and want to compute a maximum matching. The classic shortest-augmenting paths algorithm constructs a matching by at every step picking a shortest augmenting path in the graph. We now show a very simple argument that the total length of all these augmenting paths is $O(n \log n)$. Recall the well-known fact that if all augmenting paths in the matching have length $\geq h$, then the cardinality of the current matching is at most $2n/h$ from optimal [19]. Thus, the algorithm augments down at most $2n/h$ augmenting paths of length $\geq h$. Let $P_1, P_2, ..., P_k$ denote all the paths augmented down by the algorithm in decreasing order of $|P_i|$; then $k \leq n$, and $|P_i| = h$ implies $i \leq 2n/h$. But then $|P_i| \leq 2n/i$, so $\sum_{1 \leq i \leq k} |P_i| \leq 2n \sum_{1 \leq i \leq k} \frac{1}{i} = 2n(\ln(k) + O(1)) = O(n \log k) = O(n \log n)$.

In the online setting, the algorithm does not have access to the entire graph. It can only choose the shortest augmenting path from the arriving client c. We are nonetheless able to show a similar bound for this setting:

Lemma 6. *Consider the following protocol for constructing a matching: For each client c in arbitrary order, augment along the shortest augmenting path from c (if one exists). Given any h, this protocol augments down a total of at most $4n \ln(n)/h$ augmenting paths of length $> h$.*

The proof of our main theorem then follows directly from the lemma.

Proof of Theorem 1. Note that the SAP protocol exactly follows the condition of Lemma 6. Now, given any $0 \leq i \leq \log_2(n) + 1$, we say that an augmenting path is at level i if its length is in the interval $[2^i, 2^{i+1})$. By Lemma 6, the SAP protocol augments down at most $4n \ln(n)/2^i$ paths of level i. Since each of those paths contains at most 2^{i+1} edges, the total length of augmenting paths of level i is at most $8n \ln(n)$. Summing over all levels yields the desired $O(n \log^2 n)$ bound. □

The entirety of Sections 3 and 4 is devoted to proving Lemma 6. Previous algorithms attempted to bound the total number of reassignments by tracking how some property of the matching M changes over time. For example, the analysis of Gupta et al. [15] keeps track of changes to the "height" of vertices in M, while the algorithm with $O(n\sqrt{n})$ reassignments [5] takes a more direct approach, and uses a non-SAP protocol whose changes to M depend on how often each particular client has already been reassigned.

Unfortunately such arguments have had limited success because the matching M can change quite erratically. This is especially true under the SAP protocol, which is why it has only been analyzed in very restrictive settings [6, 8, 13]. We overcome this difficulty by showing that it is

enough to analyze how new clients change the structure of the graph $G = (C \cup S, E)$, without reference to any particular matching.

Intuitively, our analysis keeps track of how "necessary" each server s is (denoted $\alpha(s)$ below). So for example, if there is a complete bipartite graph with 10 servers and 10 clients, then all servers are completely necessary. But if the complete graph has 20 servers and 10 clients, then while any matching has 10 matched servers and 10 unmatched ones, it is clear that if we abstract away from the particular matching every server is 1/2-necessary. Of course in more complicated graphs different servers might have different necessities, and some necessities might be very close to 1 (say $1 - 1/n^{2/3}$). Note that server necessities depend only on the graph, not on any particular matching. Note also that our algorithm never computes the server necessities, as they are merely an analytical tool.

We relate necessities to the number of reassignments with 2 crucial arguments. **1.** Server necessities only increase as clients are inserted, and once a server has $\alpha(s) = 1$, then regardless of the current matching, no future augmenting path will go through s. **2.** If, *in any matching*, the shortest augmenting path from a new client c is long, then the insertion of c will increase the necessity of servers that already had high necessity. We then argue that this cannot happen too many times before the servers involved have necessity 1, and thus do not partake in any future augmenting paths.

1.4 Paper outline

In Section 2, we introduce the terminology necessary to understand the paper. In Section 3, we introduce and reason about the abstraction of a balanced server flow, a number that reflects the necessity of each server. In Section 4, we use the balanced server flow to prove Lemma 6, which proves our main theorem that SAP makes a total of $O(n \log^2 n)$ replacements. In Section 5, we give an efficient implementation of SAP. Finally, in Section 6, we present our results on capacitated online assignment, and for minimizing the maximum server load in the online assignment problem.

2 PRELIMINARIES AND NOTATION

Let (C, S) be the vertices, and E be the edges of a bipartite graph. We call C the *clients*, and S the *servers*. Clients arrive, one at a time, and we must maintain an explicit maximum matching of the clients. For simplicity of notation, we assume for the rest of the paper that $C \neq \emptyset$. For any vertex v, let $N(v)$ denote the neighborhood of v, and for any $V \subseteq C \cup S$ let $N(V) = \bigcup_{v \in V} N(v)$.

Theorem 7 (Hall's Marriage Theorem [18]). *There is a matching of size $|C|$ if and only if $|K| \leq |N(K)|$ for all $K \subseteq C$.*

Definition 8. Given any matching in a graph $G = (C \cup S, E)$, an alternating path is one which alternates between unmatched and matched edges. An augmenting path is an alternating path that starts and ends with an unmatched vertex. Given any augmenting path P, "flipping" the matched status of every edge on P gives a new larger matching. We call this process *augmenting down* P.

Denote by SAP the algorithm that upon the arrival of a new client c augments down the shortest augmenting path from c; ties can be broken arbitrarily, and if no augmenting path from c exists the algorithm does nothing. Chaudhuri et al. [8] showed that if the final graph contains a perfect matching, then the SAP protocol also returns a perfect matching. We now generalize this as follows:

Observation 9. *Because of the nature of augmenting paths, once a client c or a server s is matched by the SAP protocol, it will remain matched during all future client insertions. On the other hand, if a client c arrives and there is no augmenting path from c to a free server, then during the entire sequence*

of client insertions c will never be matched by the SAP protocol; no alternating path can go through c because it is not incident to any matched edges.

Lemma 10. *The SAP protocol always maintains a maximum matching in the current graph $G = (C \cup S, E)$.*

Proof. Consider for contradiction the first client c such that after the insertion of c, the matching M maintained by the SAP protocol is not a maximum matching. Let C be the set of clients before c was inserted. Since M is maximum in the graph $G = (C \cup S, E)$ but not in $G' = (C \cup S \cup \{c\}, E)$, it is clear that c is matched in the maximum matching M' of G' but not in M. But this contradicts the well known property of augmenting paths that the symmetric difference $M \oplus M'$ contains an augmenting path in M from c to a free server. □

3 THE SERVER FLOW ABSTRACTION

3.1 Defining the Server Flow

We now formalize the notion of server necessities from Section 1.3 by using a flow-based notation. The necessity of a server s will be the value $\alpha(s)$ of a balanced server flow α: We first define a server flow, then define what it means for a server flow to be balanced, and, finally, show that the balanced server flow is unique.

Definition 11. Given any graph $G = (C \cup S, E)$, define a *server flow* α as any map from S to the nonnegative reals such that there exist nonnegative $(x_e)_{e \in E}$ with:

$$\forall c \in C : \sum_{s \in N(c)} x_{cs} = 1 \qquad \forall s \in S : \sum_{c \in N(s)} x_{cs} = \alpha(s)$$

We say that such a set of x-values *realize* the server flow.

A server flow can be thought of as a fractional assignment from C to S; note, however, that is is not necessarily a fractional matching, since servers may have a load greater than 1. Note also that the same server flow may be realized in more than one way. Furthermore, if $|N(c)| = 0$ for some $c \in C$ then $\sum_{s \in N(c)} x_{cs} = 0 \neq 1$, so no server flow is possible. So suppose (unless otherwise noted) that $|N(c)| \geq 1$ for all $c \in C$.

The following theorem can be seen as a generalization of Hall's Marriage Theorem:

Lemma 12. *If $\max_{\emptyset \subset K \subseteq C} \frac{|K|}{|N(K)|} = \frac{p}{q}$, then there exists a server flow where every server $s \in S$ has $\alpha(s) \leq \frac{p}{q}$.*

Proof. Let C^* be the original set C but with q copies of each client. Similarly let S^* contain p copies of each server, and let E^* consist of all pq edges between copies of the endpoints of each edge in E.

Now let $K^* \subseteq C^*$, and let $K \subseteq C$ be the originals that the vertices in K^* are copies of. Then $|K^*| \leq q\,|K| \leq p\,|N(K)| = |N(K^*)|$, so the graph $(C^* \cup S^*, E^*)$ satisfies Hall's theorem and thus it has some matching M in which every client in C^* is matched. Now, for $cs \in E$ let

$$x_{cs} = \tfrac{1}{q} \left| \left\{ c^* s^* \in M \;\middle|\; c^* \text{ is a copy of } c \text{ and } s^* \text{ is a copy of } s \right\} \right|$$

Since for each $c \in C$ all q copies of c are matched, $\sum_{s \in N(c)} x_{cs} = \frac{q}{q} = 1$ for all $c \in C$. Similarly, since for each $s \in S$ at most p copies of s are matched, $\sum_{c \in N(s)} x_{cs} \leq \frac{p}{q}$. Thus, $(x_e)_{e \in E}$ realizes the desired server flow. □

Definition 13. We say that a server flow α is *balanced*, if additionally:

$$\forall c \in C, s \in N(c) \setminus A(c) : x_{cs} = 0 \qquad \text{where } A(c) = \arg\min_{s \in N(c)} \alpha(s)$$

That is, if each client only sends flow to its least loaded neighbours.

We call the set $A(c)$ the *active* neighbors of c, and we call an edge cs *active* when $s \in A(c)$. We extend the definition to sets of clients in the natural way, so for $K \subseteq C$, $A(K) = \bigcup_{c \in K} A(c)$.

3.2 Uniqueness of Loads in a Balanced Server Flow

While there may be more than one server flow, and while there may be many possible x-values x_{cs} that realize any given flow, we will nonetheless show that the balanced server flow α is unique.

Lemma 14. *A unique balanced server flow exists if and only if $|N(c)| \geq 1$ for all $c \in C$.*

Clearly, it is necessary for all clients to have at least one neighbor for a server flow to exist, so the "only if" part is obvious. We dedicate the rest of this section to proving that this condition is sufficient. In fact, we provide two different proofs of uniqueness, the first of which is simpler but provides less intuition for what the unique $\alpha(s)$ values signify about the structure of the graph.

3.2.1 Short proof of Lemma 14 via convex optimization. It is not hard to prove uniqueness by showing that a balanced server flow corresponds to the solution to a convex program[1]. Consider the convex optimization problem where the constraints are those of a *not necessarily balanced* server flow (Definition 11), and the objective function we seek to minimize is the sum of the squares of the server loads.

To be precise, the convex program contains a variable α_s for each server $s \in S$, and a variable x_{cs} for each edge (c, s) in the graph. Its objective is to minimize the function $\sum_{s \in S} \alpha_s^2$ subject to the constraints:

$$0 \leq x_{cs} \leq 1 \qquad \forall c \in C : \sum_{s \in N(c)} x_{cs} = 1 \qquad \forall s \in S : \sum_{c \in N(s)} x_{cs} = \alpha_s$$

It is easy to check that because we introduce a separate variable α_s for each server load, the objective function is strictly convex, so the convex program has a unique minimum with respect to the server loads α_s (but not the edge flows).

We now observe that this unique solution is a *balanced* server flow: the constraints of the convex program ensure that it is a server flow, and were it not balanced, there would be some client c, who has the neighbours s and s', and who sends non-zero flow to s' although $\alpha(s) < \alpha(s')$. This would be a contradiction, because we can decrease the objective function by increasing x_{cs} and decreasing $x_{cs'}$. We have thus proved the existence of a balanced server flow.

We must now prove uniqueness, i.e. that all balanced server flows have the same server loads. We will do this by showing that any balanced server flow optimizes the objective function of the convex function. There are many standard approaches for proving this claim, but the simplest one we know of is based on existing literature on load balancing with selfish agents. In particular, we rely on the following simple auxiliary lemma, which is a simplified version of Lemma 2.2 in [31].

Lemma 15. *Consider any balanced server flow x_{cs}, let $\alpha_s = \sum_{c \in C} x_{cs}$ be the server flow of s. Let x'_{cs} be* any *feasible server flow, and let $\alpha'_s = \sum_{c \in C} x'_{cs}$ be the resulting server loads. Then, we always have:*

$$\sum_{s \in S} \alpha_s^2 \leq \sum_{s \in S} \alpha_s \alpha'_s$$

[1]The authors thank Seffi Naor for pointing this out to us.

Proof. For any client c, let $\mu(c)$ (μ for minimum) be the minimum server load neighboring c in the balanced solution x_{cs}. That is, $\mu(c) = \min_{s \in N(c)} \alpha_s$. We then have

$$\sum_{s \in S} \alpha_s^2 = \sum_{s \in S} \sum_{c \in C} x_{cs}\alpha_s = \sum_{c \in C} \sum_{s \in S} x_{cs}\alpha_s = \sum_{c \in C} \sum_{s \in S} x_{cs}\mu(c) = \sum_{c \in C} \mu(c),$$

where the last equality follows from the fact that each client sends one unit of flow, and the before-last equality follows from the fact that the flow is balanced, so for any edge $(c, s) \in E$ with $x_{cs} \neq 0$ we have $\alpha_s = \mu(c)$.

From the definition of $\mu(c)$ it follows that for *any* edge $(c, s) \in E$, we have $\alpha_s \geq \mu(c)$. This yields:

$$\sum_{s \in S} \alpha'_s\alpha_s = \sum_{s \in S} \sum_{c \in C} x'_{cs}\alpha_s = \sum_{c \in C} \sum_{s \in S} x'_{cs}\alpha_s \geq \sum_{c \in C} \sum_{s \in S} x'_{cs}\mu(c) = \sum_{c \in C} \mu(c).$$

We thus have $\sum_{s \in S} \alpha_s^2 = \sum_{c \in C} \mu(c)$ and $\sum_{s \in S} \alpha'_s\alpha_s \geq \sum_{c \in C} \mu(c)$, which yields the lemma. □

We now argue that any balanced flow is an optimal solution to the convex program, and is thus unique. Consider any balanced flow with loads α_s. To show that α_s is optimum, we need to show that for any feasible solution α'_s we have $\sum_{s \in S} \alpha_s^2 \leq \sum_{s \in S} (\alpha')_s^2$. Equivalently, let $\boldsymbol{\alpha}$ and $\boldsymbol{\alpha'}$ be the vectors of server loads in the two solutions. We want to show that $\|\boldsymbol{\alpha}\| \leq \|\boldsymbol{\alpha'}\|$. This follows from $\|\boldsymbol{\alpha}\|^2 \leq \boldsymbol{\alpha} \cdot \boldsymbol{\alpha'} \leq \|\boldsymbol{\alpha}\| \cdot \|\boldsymbol{\alpha'}\|$, where the first inequality is a reformulation of Lemma 15, and the second is the Cauchy-Schwartz inequality.

3.2.2 Longer combinatorial proof of uniqueness. Although the reduction to convex programming is the most direct proof of uniqueness, it has the disadvantage of not providing any insight into what the unique $\alpha(s)$ values actually correspond to. We thus provide a more complicated combinatorial proof which shows that the $\alpha(s)$ correspond to a certain hierarchical decomposition of the graph.

The following lemma will help us upper and lower bound the sum of flow to a subset of servers.

Lemma 16. *If α is a balanced server flow, then*

$$\forall T \subseteq S : \left|\{c \in C \mid A(c) \subseteq T\}\right| \leq \sum_{s \in T} \alpha(s) \leq \left|\{c \in C \mid A(c) \cap T \neq \emptyset\}\right|$$

Proof. The first inequality is true because each client in the first set contributes exactly one to the sum (but there may be other contributions). The second inequality is true because every client contributes exactly one to $\sum_{s \in T} \alpha(s)$, and the inequality counts every client that contributes anything to $\sum_{s \in T} \alpha(s)$ as contributing one. □

The first step to proving that every graph has a unique server flow α is to show that the maximum value $\widehat{\alpha} = \max_{s \in S} \alpha(s)$ is uniquely defined. We start by showing that the generalization of Hall's Marriage Theorem from Lemma 12 is "tight" for a balanced server flow in the sense that there does indeed exist a set of p clients with neighbourhood of size q realizing the maximum α-value $\frac{p}{q}$. In fact, the maximally necessary servers and their active neighbours (defined below) form such a pair of sets:

Lemma 17. *Let α be a balanced server flow, let $\widehat{\alpha} = \max_{s \in S} \alpha(s)$ be the maximal necessity, let $\widehat{S} = \{s \in S \mid \alpha(s) = \widehat{\alpha}\}$ be the maximally necessary servers, and let $\widehat{K} = \{c \in C \mid A(c) \cap \widehat{S} \neq \emptyset\}$ be their active neighbours. Then $N(\widehat{K}) = \widehat{S}$ and $|\widehat{K}| = \widehat{\alpha}\,|\widehat{S}|$.*

Proof. Let $K = \{c \in C \mid A(c) \subseteq \widehat{S}\}$, and note that $K \subseteq \widehat{K}$. However, we also have $\widehat{K} \subseteq K$: By definition of $\widehat{S}$, and since we assume the server flow is balanced, $\widehat{K} \neq \emptyset$, and for every $c \in \widehat{K}$, $N(c) = A(c) \subseteq \widehat{S}$. Thus, $K = \widehat{K}$ and $N(\widehat{K}) = \widehat{S}$. Now, note that by Lemma 16

$$|\widehat{K}| = |K| \leq \widehat{\alpha}\,|\widehat{S}| \leq |\widehat{K}|\,.$$

□

We can thus show that $\widehat{\alpha}$ exactly equals the maximal quotient $\frac{|K|}{|N(K)|}$ over subsets K of clients.

Lemma 18. *Let α be a balanced server flow, and let $\widehat{\alpha} = \max_{s \in S} \alpha(s)$. Then*

$$\widehat{\alpha} = \max_{\emptyset \subset K \subseteq C} \frac{|K|}{|N(K)|}$$

Furthermore, for any $K \subseteq C$, if $|K| = \widehat{\alpha}\,|N(K)|$, then $\alpha(s) = \widehat{\alpha}$ for all $s \in N(K)$.

Proof. By definition of server flow, for $K \subseteq C$, $|K| \leq \sum_{s \in N(K)} \alpha(s) \leq \widehat{\alpha}\,|N(K)|$, so $|K| \leq \widehat{\alpha}\,|N(K)|$. Let $\widehat{K}$ be defined as in Lemma 17. Then $\widehat{\alpha} = \frac{|\widehat{K}|}{|N(\widehat{K})|} \leq \max_{\emptyset \subset K \subseteq C} \frac{|K|}{|N(K)|} \leq \widehat{\alpha}$. Finally, if $|K| = \sum_{s \in N(K)} \alpha(s) = \widehat{\alpha}\,|N(K)|$ then $\alpha(s) \leq \widehat{\alpha}$ for all $s \in S$ implies $\alpha(s) = \widehat{\alpha}$ for $s \in N(K)$. □

Corollary 19. *If $\max_{\emptyset \subset K \subseteq C} \frac{|K|}{|N(K)|} = \frac{|C|}{|S|}$ there is a unique balanced server flow.*

Proof. By Lemma 12 there exists a server flow with $\alpha(s) \leq \frac{|C|}{|S|}$ for all $s \in S$. Since $\sum_{s \in S} \alpha(s) = |C|$, any such flow must actually have $\alpha(s) = \frac{|C|}{|S|}$ for all $s \in S$, and be balanced. Uniqueness follows from Lemma 18. □

We are now ready to give a combinatorial proof of uniqueness. We will do so by showing that the $\alpha(s)$ in fact express a very nice structural property of the graph, which can be thought of as a hierarchy of "tightness" for the Hall constraint. As shown in Lemma 18, the maximum α value $\widehat{\alpha}$ corresponds to the tightest Hall constraint, i.e. the maximum possible value of $|K| / |N(K)|$. Now, there may be many sets K with $|K| / |N(K)| = \widehat{\alpha}$, so let $\widehat{C}$ be a maximal such set; we will show that $\widehat{C}$ is in fact the union of all sets K with $|K| / |N(K)| = \widehat{\alpha}$. Now, by Lemma 18, every server $s \in N(\widehat{C})$ has $\alpha(s) = \widehat{\alpha}$. We will show that in fact, because $\widehat{C}$ captured *all* sets with tightness $\widehat{\alpha}$, all servers $s \notin N(\widehat{C})$ have $\alpha(s) < \widehat{\alpha}$. Thus, because the flow is balanced, all active edges incident to $\widehat{C}$ or $\widehat{S}$ will be between $\widehat{C}$ and $\widehat{S}$; there will be no active edges coming from the outside. For this reason, any balanced server flow α on $G = (C \cup S)$ can be the thought of as the union of two completely independent server flows: the first (unique) flow assigns $\alpha(s) = \widehat{\alpha} = |\widehat{C}| / |N(\widehat{C})|$ to all $s \in \widehat{S}$, while the second is a balanced server flow on the remaining graph $G \setminus (\widehat{C} \cup \widehat{S})$. Since this remaining graph is smaller, we can use induction on the size of the graph to claim that this second balanced server flow has unique α-values, which completes the proof of uniqueness. If we follow through the entire inductive chain, we end up with a hierarchy of α-values, which can be viewed as the result of the following peeling procedure: first find the (maximally large) set C_1 that maximizes $\alpha_1 = |C_1| / |N(C_1)|$ and assign every server $s \in N(C_1)$ a value of α_1; then peel off C_1 and $N(C_1)$, find the (maximally large) set C_2 in the remaining graph that maximizes $\alpha_2 = |C_2| / |N(C_2)|$, and assign every server $s \in N(C_2)$ value α_2; peel off C_2 and $N(C_2)$ and continue in this fashion, until every server has some value α_i. These values α_i assigned to each server are precisely the unique $\alpha(s)$ in a balanced server flow.

Remark 20. We were unaware of this when submitting the extended abstract, but a similar hierarchical decomposition was used earlier to compute an approximate matching in the semi-streaming setting: see [12], [21]. Note that unlike those papers, we do end up relying on this decomposition for our main arguments. We only present it here to give a combinatorial alternative to the convex optimization proof above: regardless of which proof we use, once uniqueness is established, the rest of our analysis is expressed purely in terms of balanced server flows.

Proof of Lemma 14. As already noted, $|N(c)| \geq 1$ for all $c \in C$ is a necessary condition. We will prove that it is sufficient by induction on $i = |S|$. If $|S| = 1$, the flow $\alpha(s) = |C|$ for $s \in S$ is trivially

the unique balanced server flow. Suppose now that $i > 1$ and that it holds for all $|S| < i$. Now let $\widehat{\alpha} = \max_{\emptyset \subset K \subseteq C} \frac{|K|}{|N(K)|}$ and let

$$\widehat{C} = \bigcup_{K \in \mathcal{K}} K \qquad \text{where } \mathcal{K} = \left\{ K \subseteq C \;\middle|\; |K| = \widehat{\alpha}\,|N(K)| \right\}$$

Note that for any $K_1, K_2 \in \mathcal{K}$ we have

$$\begin{aligned}
\widehat{\alpha}\,|N(K_1 \cup K_2)| &\geq |K_1 \cup K_2| && \text{(by definition of } \widehat{\alpha}) \\
&= |K_1| + |K_2| - |K_1 \cap K_2| \\
&= \widehat{\alpha}\,|N(K_1)| + \widehat{\alpha}\,|N(K_2)| - |K_1 \cap K_2| && \text{(since } K_1, K_2 \in \mathcal{K}) \\
&\geq \widehat{\alpha}\,|N(K_1)| + \widehat{\alpha}\,|N(K_2)| - \widehat{\alpha}\,|N(K_1 \cap K_2)| && \text{(by definition of } \widehat{\alpha}) \\
&\geq \widehat{\alpha}\,|N(K_1 \cup K_2)| && \text{(since } |N(\cdot)| \text{ is submodular)}
\end{aligned}$$

so $K_1 \cup K_2 \in \mathcal{K}$ and thus $\widehat{C} \in \mathcal{K}$ and $|\widehat{C}| = \widehat{\alpha}\,|N(\widehat{C})|$. If $N(\widehat{C}) = S$ then $\widehat{C} = C$ (otherwise $\frac{|\widehat{C}|}{|N(\widehat{C})|} < \frac{|C|}{|S|} \leq \widehat{\alpha}$) and by Corollary 19 we are done, so suppose $\emptyset \subset N(\widehat{C}) \subset S$. Consider the subgraph G_1 induced by $\widehat{C} \cup N(\widehat{C})$ and the subgraph G_2 induced by $(C \setminus \widehat{C}) \cup (S \setminus N(\widehat{C}))$.

By Corollary 19, G_1 has a unique balanced server flow α_1 with $\alpha_1(s) = \widehat{\alpha}$ for all $s \in N(\widehat{C})$.

By our induction hypothesis, G_2 also has a *unique* balanced server flow α_2.

We proceed to show that the combination of α_1 with α_2 constitutes a unique balanced flow α of the entire graph G, defined as follows:

$$\alpha(s) = \begin{cases} \alpha_1(s) & \text{if } s \in N(\widehat{C}) \\ \alpha_2(s) & \text{otherwise} \end{cases}$$

Note first that α is a balanced server flow for G, because both G_1 and G_2 have a set of x-values that realize them, and by construction these values (together with zeroes for each edge between $C \setminus \widehat{C}$ and $N(\widehat{C})$) realize a balanced server flow for G.

For uniqueness, note that by Lemma 18 any balanced server flow α' for G must have $\alpha'(s) = \widehat{\alpha} = \alpha_1(s)$ for $s \in N(\widehat{C})$. We now show that for any $s \in S \setminus N(\widehat{C})$, any balanced server flow α' must also have $\alpha'(s) = \alpha_2(s)$; then, the uniqueness of α will follow from the uniqueness of α_1 and α_2.

Let $\widehat{S} = \{s \in S \mid \alpha'(s) = \widehat{\alpha}\}$ be the set of maximally necessary servers, and let $\widehat{K} = \{c \in C \mid A(c) \cap \widehat{S} \neq \emptyset\}$ be the set of clients with a maximally necessary server in their active neighborhood. We will show that $\widehat{K} = \widehat{C}$.

"$\subseteq$" By Lemma 17, $|\widehat{K}| = \widehat{\alpha}\,|N(\widehat{K})|$ so by definition of $\widehat{C}$, $\widehat{K} \subseteq \widehat{C}$.

"$\supseteq$" On the other hand, $|\widehat{C}| = \widehat{\alpha}\,|N(\widehat{C})|$ so by Lemma 18 we have $N(\widehat{C}) \subseteq \widehat{S}$ and in particular $A(c) \subseteq \widehat{S}$ for c in $\widehat{C}$ and thus $\widehat{C} \subseteq \widehat{K}$.

Thus, by definition of $\widehat{K}$, $A(c) \cap \widehat{S} = \emptyset$ for all $c \in C \setminus \widehat{C}$. And there are clearly no edges between $\widehat{C}$ and $S \setminus N(\widehat{C})$. But then, for any $(x_e)_{e \in E}$ realizing α', the subset $(x_{cs})_{c \in C \setminus \widehat{C}, s \in S \setminus N(\widehat{C})}$ realizes a balanced server flow in G_2, so since α_2 is the unique balanced server flow in G_2 we have $\alpha'(s) = \alpha_2(s)$ for $s \in S \setminus N(\widehat{C})$. □

3.3 How Server Loads Change as New Clients are Inserted

From now on, let α denote the unique balanced server flow. We want to understand how the balanced server flow changes as new clients are added. For any server s, let $\alpha^{\text{OLD}}(s)$ be the flow in s *before* the insertion of c, and let $\alpha^{\text{NEW}}(s)$ be the flow *after*. Also, let $\Delta\alpha(s) = \alpha^{\text{NEW}}(s) - \alpha^{\text{OLD}}(s)$.

Intuitively, as more clients are added to the graph, the flow on the servers only increases, so no $\alpha(s)$ ever decreases. We now prove this formally.

Lemma 21. *When a new client c is added, $\Delta\alpha(s) \geq 0$ for all $s \in S$.*

Proof. Let $S^* = \{s \in S \mid \alpha^{\text{NEW}}(s) < \alpha^{\text{OLD}}(s)\}$. We want to show that $S^* = \emptyset$. Say for contradiction that $S^* \neq \emptyset$, and let $\alpha^* = \min_{s \in S^*} \alpha^{\text{NEW}}(s)$. We will now partition S into three sets.

$$S^- = \{s \in S \mid \alpha^{\text{OLD}}(s) \leq \alpha^*\}$$
$$S^\Delta = \{s \in S \mid \alpha^{\text{OLD}}(s) > \alpha^* \wedge \alpha^{\text{NEW}}(s) = \alpha^*\}$$
$$S^+ = \{s \in S \mid \alpha^{\text{OLD}}(s) > \alpha^* \wedge \alpha^{\text{NEW}}(s) > \alpha^*\}$$

It is easy to see that these sets form a partition of S, and that $\emptyset \neq S^\Delta \subseteq S^*$.

Let C^Δ contain all clients with an active neighbor in S^Δ before the insertion of c. Since each client sends one unit of flow, $\sum_{s \in S^\Delta} \alpha^{\text{OLD}}(s) \leq \left|C^\Delta\right|$. Now, because we had a balanced flow before the insertion of c, there cannot be any edges in G from C^Δ to S^- (any such edge would be from a client $u \in C^\Delta$ to a server $v \in S^-$ with $\alpha^{\text{OLD}}(v) \leq \alpha^* < \alpha^{\text{OLD}}(s)$ for $s \in S^\Delta$ contradicting that u had an active neighbor in S^Δ). Moreover, in the balanced flow after the insertion of c, there are no active edges from C^Δ to S^+ (any such edge would be from a client $u \in C^\Delta$ to a server $v \in S^+$ with $\alpha^{\text{NEW}}(v) > \alpha^* = \alpha^{\text{NEW}}(s)$ for all $s \in S^\Delta$ so is not active). Thus, all active edges incident to C^Δ go to S^Δ, so $\sum_{s \in S^\Delta} \alpha^{\text{NEW}}(s) \geq \left|C^\Delta\right|$. This contradicts the earlier fact that $\sum_{s \in S^\Delta} \alpha^{\text{OLD}}(s) \leq \left|C^\Delta\right|$, since by definition of S^Δ we have $\sum_{s \in S^\Delta} \alpha^{\text{NEW}}(s) < \sum_{s \in S^\Delta} \alpha^{\text{OLD}}(s)$. □

The next lemma formalizes the following argument: Say that we insert a new client c, and for simplicity say that c is only incident to server s. Now, c will have no choice but to send all of its flow to s, but that does not imply that $\Delta\alpha(s) = 1$, since other clients will balance by retracting their flow from s and sending it elsewhere. But by the assumption that the flow was balanced before the insertion of c, all this new flow can only flow "upward" from s: it cannot end up increasing the flow on some s^- with $\alpha^{\text{OLD}}(s^-) < \alpha^{\text{OLD}}(s)$. Along the same lines of intuition, even if c has several neighbors, inserting c cannot affect the flow of servers whose original flow was less than the lowest original flow among the neighbors of s.

Lemma 22. *When a new client c is added , $\Delta\alpha(s) = 0$ for all s where $\alpha^{OLD}(s) < \min_{v \in N(c)} \alpha^{OLD}(v)$.*

Proof. Let us first consider the balanced flow *before* the insertion of c.

Let $S^+ = \left\{s \in S \;\middle|\; \alpha^{\text{OLD}}(s) \geq \min_{v \in N(c)} \alpha^{\text{OLD}}(v)\right\}$ and define $S^- = S \setminus S^+$. We want to show that $\Delta\alpha(s) = 0$ for all servers s in S^-.

Define C^+ to contain all client vertices incident to S^+; that is $C^+ = \{c \in C \mid N(c) \cap S^+ \neq \emptyset\}$. Let $C^- = C \setminus C^+$. Note that because the flow is balanced there are no edges in G from C^+ to S^- and there are no *active* edges from C^- to S^+ before the insertion of c. Thus, $\sum_{s \in S^-} \alpha^{\text{OLD}}(s) = |C^-|$.

Now consider the insertion of c. By definition of S^- the new client c has no neighbors in S^-, so it is still the case that only clients in C^- have neighbors in S^-. Thus, in the new balanced flow we still have have that $\sum_{s \in S^-} \alpha^{\text{NEW}}(s) \leq |C^-|$. But this means that $\sum_{s \in S^-} \Delta\alpha(s) \leq 0$, so if $\Delta\alpha(s_1) > 0$ for some $s_1 \in S^-$ then $\Delta\alpha(s_2) < 0$ for some $s_2 \in S^-$, which contradicts Lemma 21. □

4 ANALYZING REPLACEMENTS IN MAXIMUM MATCHING

We now consider how server flows relate to the length of augmenting paths.

Lemma 23. *The graph $(C \cup S, E)$ contains a matching of size $|C|$, if and only if $\alpha(s) \leq 1$ for all $s \in S$.*

Proof. Let $\widehat{\alpha} = \max_{s \in S} \alpha(s)$. It follows directly from Lemma 18 that $|K| \leq |N(K)|$ for all $K \subseteq C$ if and only if $\widehat{\alpha} \leq 1$. The corollary then follows from Hall's Theorem (Theorem 7) □

It is possible that in the original graph $G = (C \cup S, E)$, there are many clients that cannot be matched. But recall that by Observation 9, if a client cannot be matched when it is inserted, then it can be effectively ignored for the rest of the algorithm. This motivates the following definition:

Definition 24. We define the set $C_M \subseteq C$ as follows. When a client c is inserted, consider the set of clients C' before c is inserted: then $c \in C_M$ if the maximum matching in $(C' \cup \{c\} \cup S, E)$ is greater than the maximum matching in $(C' \cup S, E)$. Define $G_M = (C_M \cup S, E)$.

Observation 25. *When a client $c \in C_M$ is inserted the SAP algorithm finds an augmenting path from c to a free server; this follows from the fact that SAP always maintains a maximum matching (Lemma 10). By Observation 9, if $c \notin C_M$ then no augmenting path goes through c during the entire sequence of insertions. By the same observation, once a vertex $c \in C_M$ is inserted it remains matched through the entire sequence of insertions.*

Definition 26. Let α_M denote the balanced server flow in G_M; by Lemma 14 α_M is uniquely defined.

Observation 27. *By construction G_M contains a matching of size $|C_M|$, so by Lemma 23 $\alpha_M(s) \leq 1$ for all $s \in S$. Finally, note that since $C_M \subseteq C$, we clearly have $\alpha_M(s) \leq \alpha(s)$*

Definition 28. Define an *augmenting tail* from a vertex v to be an alternating path that starts in v and ends in an unmatched server. We call an alternating path *active* if all the edges on the alternating path that are not in the matching are active.

Note that augmenting tails as defined above are an obvious extension of the concept of augmenting paths: Every augmenting path for a newly arrived client c consists of an edge (c, s), plus an augmenting tail from some server $s \in N(c)$.

We are now ready to prove our main lemma connecting the balanced server flow to augmenting paths. We show that if some server s has small $\alpha(s)$, then regardless of the particular matching at hand, there is guaranteed to be a *short* active augmenting tail from s. Since every *active* augmenting tail is by definition an augmenting tail, this implies that any newly inserted client c that is incident to s has a short augmenting path to an unmatched server.

Lemma 29 (Expansion Lemma). *Let $s \in S$, and suppose $\alpha_M(s) = 1 - \epsilon$ for some $\epsilon > 0$. Then there is an active augmenting tail for s of length at most $\frac{2}{\epsilon} \ln(|C_M|)$.*

Proof. By our definition of active edges, it is not hard to see that any server s' reachable from s by an active augmenting tail has $\alpha_M(s') \leq 1 - \epsilon$.

For $i \geq 1$, let K_i be the set of clients c reachable from s via an active alternating path that starts with a matched edge and visits at most i clients (including c). Let $k_i = |K_i|$. Note that $k_1 = 1$, $K_1 \subseteq K_2 \subseteq \ldots \subseteq K_i$, and

$$k_i = |K_i| \leq \sum_{s' \in A(K_i)} \alpha_M(s') \leq \sum_{s' \in A(K_i)} (1 - \epsilon) = |A(K_i)|\,(1 - \epsilon)$$

Thus

$$|A(K_i)| \geq \frac{k_i}{1 - \epsilon}$$

Suppose there is no active augmenting tail from s of length $\leq 2(i-1)$, then every server in $A(K_i)$ is matched, and the clients they are matched to are exactly K_{i+1}. There is a bijection between $A(K_i)$ and K_{i+1} given by the perfect matching, so we have $k_{i+1} = |A(K_i)|$ and thus $|C_M| \geq k_{i+1} \geq \frac{1}{1-\epsilon} k_i \geq (\frac{1}{1-\epsilon})^i k_1 = (\frac{1}{1-\epsilon})^i$. It follows that $i \leq \frac{\ln |C_M|}{\ln \frac{1}{1-\epsilon}} \leq \frac{1}{\epsilon} \ln |C_M|$, where the last

inequality follows from $1 - \epsilon \leq e^{-\epsilon}$. Thus for any $i > \frac{1}{\epsilon} \ln |C_M|$ there exists an active augmenting tail of length at most $2(i-1)$, and the result follows. □

We are now able to prove the key lemma of our paper, which we showed in Section 1.3 implies Theorem 1.

Lemma 6. *Consider the following protocol for constructing a matching: For each client c in arbitrary order, augment along the shortest augmenting path from c (if one exists). Given any h, this protocol augments down a total of at most $4n\ln(n)/h$ augmenting paths of length $> h$.*

Proof. Recall that $n = |C| \geq |C_M|$. The lemma clearly holds for $h \leq 4\ln(n)$ because there are at most n augmenting paths in total. We can thus assume for the rest of the proof that $h > 4\ln(n)$. Recall by Observation 25 that any augmenting path is contained entirely in G_M. Now, let $C^* \subseteq C_M$ be the set of clients whose shortest augmenting path have length at least $h+1$ when they are added. Our goal is to show that $|C^*| \leq 4n\ln(n)/h$. For each $c \in C^*$ the shortest augmenting tail from each server $s \in N(c)$ has length at least h and so by the Expansion Lemma 29, each server $s \in N(c)$ has $\alpha_M(s) \geq 1 - 2\ln(n)/h$. Let S^* be the set of all servers that at some point have $\alpha_M(s) \geq 1 - 2\ln(n)/h$; by Lemma 21, this is exactly the set of servers s such that $\alpha_M(s) \geq 1 - 2\ln(n)/h$ after all clients have been inserted. By Lemma 22, if $c \in C^*$, the insertion of c only increases the flow on servers in S^* that already had flow at least $1 - 2\ln(n)/h$. Since by Observation 27 $\alpha_M(s) \leq 1$ for all $s \in S$, the flow of each server in S^* can only increase by at most $2\ln(n)/h$. But then, since the client c contributes with exactly one unit of flow, the total number of such clients is $|C^*| \leq (2\log(n)/h)\,|S^*|$. We complete the proof by showing that $|S^*| < 2n$. This follows from the fact that each client $c \in C_M$ sends one unit of flow, so $n \geq |C_M| \geq (1 - 2\ln(n)/h)\,|S^*| > |S^*|/2$, where the last inequality follows from the assumption that $h > 4\ln(n)$. □

5 IMPLEMENTATION

In the previous section we proved that augmenting along a shortest augmenting path yields a total of $O(n\log^2 n)$ replacements. But the naive implementation would spend $O(m)$ time per inserted vertex, leading to total time $O(mn)$ for actually maintaining the matching. In this section, we show how to find the augmenting paths more quickly, and thus maintain the optimal matching at all times in $O(m\sqrt{n}\sqrt{\log n})$ total time, differing only by an $O(\sqrt{\log n})$ factor from the classic offline algorithm of Hopcroft and Karp algorithm for static graphs [19].

Definition 30. Define the height of a vertex v (server or client) to be the length of the shortest augmenting tail (Definition 28) from v. If no augmenting tail exists, we set the height to $2n$.

At a high level, our algorithm is very similar to the standard $O(m\sqrt{n})$ blocking flow algorithm. We will keep track of heights to find shortest augmenting paths of length at most $\sqrt{n}\sqrt{\log n}$. We will find longer augmenting paths using the trivial $O(m)$ algorithm, and use Lemma 6 to bound the number of such paths. Our analysis will also require the following lemma:

Lemma 31. *For any server $s \in S$, there is an augmenting tail from s to an unmatched server if and only if $\alpha_M(s) < 1$.*

Proof. If $\alpha_M(s) < 1$, then the existence of *some* tail follows directly from the Expansion Lemma 29. Now let us consider $\alpha_M(s) = 1$. Let $S_1 = \{s \in S \mid \alpha_M(s) = 1\}$. Since 1 is the maximum possible value of $\alpha_M(s)$ (Observation 27), Lemma 17 implies that there is a set of clients $C_1 \in C_M$ such that $N(C_1) = S_1$ and $|C_1| = |S_1|$. Now since every client in C_1 is matched, every server S_1 is matched to some client in C_1. Every augmenting tail from some $s \in S_1$ must start with a matched edge, so it must go through C_1, so it never reaches a server outside of $N(C_1) = S_1$, so it can never reach a free server. □

We now turn to our implementation of the SAP protocol. We will use a dynamic single-source shortest paths algorithm as a building block. We start by defining a directed graph D such that maintaining distances in D will allow us to easily find shortest augmenting paths as new clients are inserted.

Let D be the directed graph obtained from $G = (C \cup S, E)$ by directing all unmatched edges from C to S, and all matched edges from S to C, and finally adding a *sink* t with an edge from all unmatched vertices, as well as an edge from each client in C that has not yet arrived. Any alternating path in G corresponds to a directed path in $D \setminus \{t\}$ and vice-versa. In particular, it is easy to see that if P is a shortest path in D from a matched server s to the sink t, then $P \setminus \{t\}$ is a shortest augmenting tail from s to a free server. Similarly, for any client c that has arrived (so edge (c, t) is deleted) but is not yet matched, if P is the shortest path from c to t in D, then $P \setminus \{t\}$ is a shortest augmenting path for c in G. Furthermore, augmenting down this path in G corresponds (in D) to changing the direction of all edges on $P \setminus \{t\}$ and deleting the edge on P incident to t.

We can thus keep track of shortest augmenting paths by using a simple dynamic shortest path algorithm to maintain shortest paths to t in the changing graph D. We will use a modification of Even and Shiloach (See [29]) to maintain a shortest path tree in D to t from all vertices of height at most $h = \sqrt{n}\sqrt{\log n}$. The original version by Even and Shiloach worked only for undirected graphs, and only in the decremental setting where the graph only undergoes edge deletions, never edge insertions. This was later extended by King [23] to work for directed graphs. The deletions-only setting is too constrained for our purposes because we will need to insert edges into D; augmenting down a path P corresponds to deleting the edges on P and inserting the reverse edges. Fortunately, it is well known that the Even and Shiloach tree can be extended to the setting where there are both deletions and insertions, as long as the latter are guaranteed not to decrease distances; we will show that this in fact applies to our setting.

Lemma 32 (Folklore. See e.g. [3, 4]). *Let $G = (V, E)$ be a dynamic directed or undirected graph with positive integer weights, let t be a fixed sink, and say that for every vertex v we are guaranteed that the distance $\text{dist}(v, t)$ never decreases due to an edge insertion. Then we can maintain a tree of shortest paths to t up to distance d in total time $O(m \cdot d + \Delta)$, where m is the total number of edges (u, v) such that (u, v) is in the graph at any point during the update sequence, and Δ is the total number of edge changes.*

Theorem 2. *There is an implementation of the SAP protocol that runs in total time $O(m\sqrt{n}\sqrt{\log n})$.*

Proof. We will explicitly maintain the graph D, and use the extended Even-Shiloach tree structure from Lemma 32 to maintain a tree T of shortest paths to t up to distance $h = \sqrt{n}\sqrt{\log n}$. Every vertex will either be in this tree (and hence have height less than h), or be marked as a *high* vertex. When a new client c arrives, we update D (and T) by first adding edges to $N(c)$ from c, and then deleting the dummy edge from c to t. Note that because the deletion of edge (c, t) comes last, the inserted edges do not change any distances to t. We then use D and T to find a shortest augmenting path. We consider two cases.

The first case is when c is not high. Then T contains a shortest path P from c to t.

The second case is when c is high. In this case we can just brute-force search for a shortest path P from c to t in time $O(m)$. *If we do not find a path from c to t, then we remove all servers and clients encountered during the search, and continue the algorithm in the graph with these vertices removed.*

In either case, if a shortest path P from c to t is found, we augment down P and then make the corresponding changes to D: we first reverse the edges on $P \setminus \{t\}$ in order starting with the edge closest to c, and then we delete the edge (s, t) on P incident to t (because the server s is now matched). Each edge reversal is done by first inserting the reversed edge, and then deleting the

original. Note that since P is a shortest path, none of these edge insertions change the distances to t.

Correctness: We want to show that our implementation chooses a shortest augmenting path at every step. This is clearly true if we always find an augmenting path, but otherwise becomes a bit more subtle as we delete vertices from the graph after a failed brute-force search. We must thus show that any vertex deleted in this way cannot have participated in any future augmenting path.

To see this, note that when our implementation deletes a server $s \in S$, there must have been no augmenting path through s at the time that s was deleted. By Lemma 31, this implies that $\alpha_M(s) = 1$. But then by Lemma 21 we have $\alpha_M(s) = 1$ for all future client insertions as well. (Recall that by Observation 27 we never have $\alpha_M(s) > 1$.) Thus by Lemma 31 there is never an augmenting path through s after this point, so s can safely be deleted from the graph. Similarly, if a client c is deleted from the graph, then all of its neighboring servers had no augmenting tails at that time, so they all have $\alpha_M(s) = 1$, so there will never be an augmenting path through c.

Running time: There are three factors to consider.

(1) the time to follow the augmenting paths and maintain D.
(2) the time to maintain T.
(3) the time to brute-force search for augmenting paths.

Item 1 takes $O(m + n\log^2 n)$ time because we need $O(1)$ time to add each of the m edges and to follow and reverse each edge in the augmenting paths, and by Theorem 1 the total length of augmenting paths is $O(n\log^2 n)$.

For Item 2, it is easy to see that the total number of edges ever to appear in D is $m = O(|E|)$; D consists only of dummy edges to the sink t, and edges in the original graph oriented in one of two directions. By Item 1, the number of changes to D is $O(m + n\log^2 n)$. Thus by Lemma 32 the total time to maintain T is $O(mh + n\log^2 n)$.

For Item 3 we consider two cases. The first is brute-force searches which result in finding an augmenting path. These take a total of $O(mn\log(n)/h)$ time because by Lemma 6 during the course of the entire algorithm there are at most $O(n\log(n)/h)$ augmenting paths of length $\geq h$, and each such path requires $O(m)$ time to find. The second case to consider is brute-force searches that do not result in an augmenting path. These take total time $O(m)$ because once a vertex participates in such a search, it is deleted from the graph with all its incident edges.

Summing up, the total time used is $O(mh + n\log^2 n + mn\log(n)/h + m)$, which for our choice of $h = \sqrt{n}\sqrt{\log n}$ is $O(m\sqrt{n}\sqrt{\log n})$. □

6 EXTENSIONS

In many applications of online bipartite assignments, it is natural to consider the extension in which each server can serve multiple clients. Recall from the introduction that we examine two variants: capacitated assignment, where each server comes with a fixed capacity which we are not allowed to exceed, and minimizing maximum server load, in which there is no upper limit to the server capacity, but we wish to minimize the maximum number of clients served by any server. We show that there is a substantial difference between the number of reassignments: Capacitated assignment is equivalent to uncapacitated online matching with replacements, but for minimizing maximum load, we show a significantly higher lower bound.

6.1 Capacitated assignment

We first consider the version of the problem where each server can be matched to multiple clients. Each server comes with a positive integer capacity $u(s)$, which denotes how many clients can be

matched to that server. The greedy algorithm is the same as before: when a new client is inserted, find the shortest augmenting path to a server s that currently has less than $u(s)$ clients assigned.

Theorem 3. *SAP uses at most $O(n \log^2 n)$ reassignments for the capacitated assignment problem, where n is the number of clients.*

Proof. There is a trivial reduction from any instance of capacitated assignment to one of uncapacitated matching where each server can only be matched to one client: simple create $u(s)$ copies of each server s. This reduction was previously used in [2]. When a client c is inserted, if there is an edge (c, s) in the original graph, then add edges from c to every copy of s. It is easy to see that the number of flips made by the greedy algorithm in the capacitated graph is exactly equal to the number made in the uncapacitated graph, which by Theorem 1 is $O(n \log^2 n)$. (Note that although the constructed uncapacitated graph has more servers than the original capacitated graph, the number of clients n is exactly the same in both graphs.) □

6.2 Minimizing the maximum server load

In this section, we analyze the online assignment problem of minimizing maximum load. Here, servers have an unlimited capacity, but we wish to minimize the maximum server load.

Definition 33. Given a bipartite graph $G = (C \cup S, E)$, an assignment $\mathcal{A} : C \to S$ assigns each client c to a server $\mathcal{A}(c) \in S$. Given some assignment $\mathcal{A}$, for any $s \in S$ let the *load* of s, denoted $\ell_{\mathcal{A}}(s)$, be the number of clients assigned to s; when the assignment $\mathcal{A}$ is clear from context we just write $\ell(s)$. Let $\ell(\mathcal{A}) = \max_{s \in S} \ell_{\mathcal{A}}(s)$. Let $\textsc{opt}(G)$ be the minimum load among all possible assignments from C to S.

In the online assignment problem, clients are again inserted one by one with all their incident edges, and the goal is to maintain an assignment with minimum possible load. More formally, define $G_t = (C_t \cup S, E_t)$ to be the graph after exactly t clients have arrived, and let $\mathcal{A}_t$ be the assignment at time t. Then we must have that for all t, $\ell(\mathcal{A}_t) = \textsc{opt}(G_t)$. The goal is to make as few changes to the assignment as possible.

[15] and [2] showed how to solve this problem with approximation: namely, with only $O(1)$ amortized changes per client insertion they can maintain an assignment $\mathcal{A}$ such that for all t, $\ell(\mathcal{A}_t) \leq 8\textsc{opt}(G_t)$. Maintaining an approximate assignment is thus not much harder than maintaining an approximate maximum matching, so one might have hoped that the same analogy holds for the exact case, and that it is possible to maintain an optimal assignment with amortized $O(\log^2 n)$ changes per client insertion. We now present a lower bound disproving the existence of such an upper bound. The lower bound is not specific to the greedy algorithm, and applies to any algorithm for maintaining an assignment $\mathcal{A}$ of minimal load. In fact, the lower bound applies even if the algorithm knows the entire graph G in advance; by contrast, if the goal is only to maintain a maximum matching, then knowing G in advance trivially leads to an online matching algorithm that never has to rematch any vertex.

Theorem 4. *For any positive integers n and $L \leq \sqrt{n/2}$ divisible by 4 there exists a graph $G = (C \cup S, E)$ with $|C| = n$ and $\textsc{opt}(G) = L$, along with an ordering in which the clients in C are inserted, such that any algorithm for the exact online assignment problem of minimizing maximum load requires a total of $\Omega(nL)$ changes. This lower bound holds even if the algorithm knows the entire graph G in advance, as well as the order in which the clients are inserted.*

The main ingredient of the proof is the following lemma:

Lemma 34. *For any positive integer L divisible by 4, there exists a graph $G = (C \cup S, E)$ along with an ordering in which clients in C are inserted, such that $|C| = L^2$, $|S| = L$, $\textsc{opt}(G) = L$, and any algorithm for maintaining an optimal assignment $\mathcal{A}$ requires $\Omega(L^3)$ changes to $\mathcal{A}$.*

Proof. Let $S = \{s_1, s_2, ..., s_L\}$. We partition the clients in C into L blocks $C_1, C_2, ..., C_L$, where all the clients in a block have the same neighborhood. In particular, clients in C_L only have a single edge to server s_L, and clients in C_i for $i < L$ have an edge to s_i and s_{i+1}.

The online sequence of client insertions begins by adding $L/2$ clients to each block C_i. The online sequence then proceeds to alternate between *down-heavy* epochs and *up-heavy* epochs, where a down-heavy epoch inserts 2 clients into blocks $C_1, C_2, ..., C_{L/2}$ (in any order), while an up-heavy epoch inserts 2 clients into blocks $C_{L/2+1}, ..., C_L$. The sequence then terminates after $L/2$ such epochs: $L/4$ down-heavy ones and $L/4$ up-heavy ones in alternation. Note that a down-heavy epoch followed by an up-heavy one simply adds two clients to each block. Thus the final graph has $|C_i| = L$ for each i, so the graph $G = (C \cup S, E)$ satisfies the desired conditions that $|C| = L^2$ and $\textsc{opt}(G) = L$.

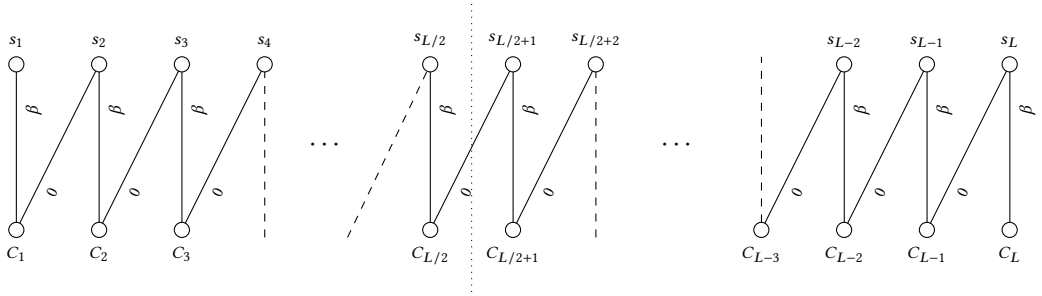

Fig. 1. Number of assignments of each type after first $L/2$ clients added to each block, and after each up-heavy phase. Each C_i has β clients. Each server has β clients assigned.

We complete the proof by showing that all the client insertions during a single down-heavy epoch cause the algorithm to make at least $\Omega(L^2)$ changes to the assignment; the same analysis applies to the up-heavy epochs as well. Consider the kth down-heavy epoch of client insertions. Let $\beta = L/2 + 2(k-1)$ and consider the graph $G^{\text{OLD}} = (C^{\text{OLD}} \cup S, E^{\text{OLD}})$ before the down-heavy epoch: it is easy to see that every block C_i has exactly β clients, that $\textsc{opt}(G^{\text{OLD}}) = \beta$, and that there is exactly one assignment $\mathcal{A}^{\text{OLD}}$ that adheres to this maximum load: $\mathcal{A}^{\text{OLD}}$ assigns all clients in block C_i to server s_i (see Figure 1).

Now, consider the graph $G^{\text{NEW}} = (C^{\text{NEW}} \cup S, E^{\text{NEW}})$ after the down-heavy epoch. The blocks $C_1, C_2, ..., C_{L/2}$ now have $\beta + 2$ clients, while the blocks $C_{L/2+1}, ..., C_L$ still only have β. We now show that $\textsc{opt}(G^{\text{NEW}}) = \beta + 1$. In particular, recall that $\beta \geq L/2$ and consider the following type of assignment $\mathcal{A}^{\text{NEW}}$: for $i \leq L/2$, $\mathcal{A}^{\text{NEW}}$ assigns $\beta + 2 - i \geq 2$ clients from C_i to s_i and i clients in C_i to s_{i+1}; for $L/2 < i \leq L$, $\mathcal{A}^{\text{NEW}}$ assigns $\beta + i - L \geq 0$ clients in C_i to s_i, and $L - i$ clients from C_i to s_{i+1}. (In particular, all β clients in C_L are assigned to s_L, which is necessary as there is no server s_{L+1}). It is easy to check that for every $s \in S$, $\ell_{\mathcal{A}^{\text{NEW}}}(s) = \beta + 1$ (see Figure 2).

We now argue that $\mathcal{A}^{\text{NEW}}$ is in fact the only type of assignment $\mathcal{A}$ in G^{NEW} with $\ell(\mathcal{A}) = \beta + 1$. Consider any assignment $\mathcal{A}$ for C^{NEW} with $\ell(\mathcal{A}) = \beta + 1$. Observe that since the total number of clients in C^{NEW} is exactly $(\beta + 1)L$, we must have that every server $s \in S$ has $\ell(s) = \beta + 1$ in $\mathcal{A}$. We now argue by induction that for $i \leq \beta/2$, $\mathcal{A}$ assigns assigns $\beta + 2 - i$ clients from C_i to s_i and i clients in C_i to s_{i+1} (exactly as $\mathcal{A}^{\text{NEW}}$ does). The claim holds for $i = 1$ because the only way s_1 can end up with load $\beta + 1$ is if $\beta + 1$ clients from C_1 are assigned to it. Now say the claim is

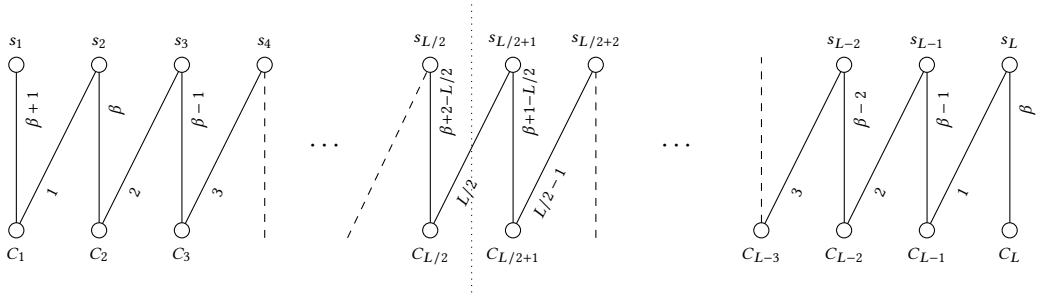

Fig. 2. Number of assignments of each type after each down-heavy phase. Each C_i has $\beta + 2$ clients for $1 \le i \le L/2$ and β clients for $L/2 + 1 \le i \le L$. Each server has $\beta + 1$ clients assigned.

true for some $i < \beta/2$. By the induction hypothesis, s_{i+1} has i clients from C_i assigned to it. Since s_{i+1} must have total load $\beta + 1$, and all clients assigned to it come from C_i or C_{i+1}, s_{i+1} must have $\beta + 1 - i = \beta + 2 - (i + 1)$ clients assigned to it from C_{i+1}.

We now prove by induction that for all $L/2 < i \le L$, $\mathcal{A}$ assigns $\beta + i - L$ clients in C_i to s_i, and $L - i$ clients from C_i to s_{i+1}, which proves that $\mathcal{A}$ is of type $\mathcal{A}^{\text{NEW}}$. The claim holds for $i = L/2 + 1$ because we have already shown that in the above paragraph that $L/2$ clients assigned to $s_i = s_{L/2+1}$ come from $C_{L/2}$, so since $\ell(s_i) = \beta + 1$, it must have $\beta + 1 - L/2 = \beta + i - L$ clients from C_i assigned to it. Now, say that the claim is true for some $i > L/2$. Then by the induction step s_{i+1} has $L - i$ clients assigned to it from C_i, so since $\ell(s_{i+1}) = \beta + 1$, it has $\beta + (i + 1) - L$ clients assigned to it from C_{i+1}, as desired. The remaining $L - (i + 1)$ clients in C_{i+1} must then be assigned to s_{i+2}.

We have thus shown that the online assignment algorithm is forced to have assignment $\mathcal{A}^{\text{OLD}}$ before the down-heavy epoch, and an assignment of type $\mathcal{A}^{\text{NEW}}$ afterwards. We now consider how many changes the algorithm must make to go from one to another. Consider block C_i for some $L/2 < i \le L$. Note that because the epoch of client insertions was down-heavy, $|C_i| = \beta$ before and after the epoch. Now, in $\mathcal{A}^{\text{OLD}}$ all of the clients in C_i are matched to s_i. But in $\mathcal{A}^{\text{NEW}}$, $L - i$ of them are matched to s_{i+1}. Thus, the total number of reassignments to get from $\mathcal{A}^{\text{OLD}}$ to $\mathcal{A}^{\text{NEW}}$ is at least $\sum_{L/2<i\le L}(L - i) = \Omega(L^2)$. Since there are $L/4$ down-heavy epochs, the total number of reassignments over the entire sequence of client insertions is $\Omega(L^3)$. □

Proof of Theorem 4. Recall the assumption of the Theorem that $n/2 \ge L^2$. Simply let the graph G consist of $\lfloor n/L^2 \rfloor$ separate instances of the graph in Lemma 34, together with sufficient copies of $K_{1,1}$ to make the total number of clients n. The algorithm will have to make $\Omega(L^3)$ changes in each such instance, leading to $\Omega(L^3 \lfloor n/L^2 \rfloor) = \Omega(nL)$ changes in total. □

We now show a nearly matching upper bound which is off by a $\log^2 n$ factor. As with the case of matching, this upper bound is achieved by the most natural SAP algorithm, which we now define in this setting. Since the insertion of a client may cause $\text{OPT}(G)$ to change, whenever a new client is inserted, the greedy algorithm must first compute $\text{OPT}(G)$ for the next client set. Note that the algorithm does not do any reassignments at this stage, it simply figures out what the max load should be. $\text{OPT}(G)$ can easily be computed in polynomial time: for example we could just compute the maximum matching when every server has capacity b for every $b = 1, 2, \ldots, |C|$, and then $\text{OPT}(G)$ is the minimum b for which every client in C is matched; for a more efficient approach see [2]. Now, when a new client c is inserted, the algorithm first checks if $\text{OPT}(G)$ increases. If yes, the maximum allowable load on each server increases by 1 so c can just be matched to an arbitrary neighbor. Otherwise, SAP finds the shortest alternating path from c to a server s with

$\ell(s) < \textsc{opt}(G)$: an augmenting path is defined exactly the same way as in Definition 8, though there may now be multiple matching edges incident to every server. The proof of the upper bound will rely on the following very simple observation:

Observation 35. *For the uncapacitated problem of online maximum matching with replacements, let us say that instead of starting with $C = \emptyset$, the algorithm starts with some initial set of clients $C_0 \subset C$ already inserted, and an initial matching between C_0 and S. Then the total number of replacements made during all future client insertions is still upper bounded by the same $O(n \log^2 n)$ as in Theorem 1, where n is the number of clients in the final graph (so n is $|C_0|$ plus the number of clients inserted).*

Proof. Intuitively, we could simply let our protocol start by unmatching all the clients in C_0, and then rematching them according the SAP protocol, which would lead to $O(n \log^2 n)$ replacements. In fact this initial unmatching is not actually necessary. Recall that the proof of Theorem 1 follows directly from the key Lemma 6, which in term follows from the expansion argument in Lemma 29. The expansion argument only refers to server necessities, not to the particular matching maintained by the algorithm, so it will hold no matter what initial matching we start with. □

Theorem 5. *Let C be the set of all clients inserted, let $n = |C|$, and let $L = \textsc{opt}(G)$ be the minimum possible maximum load in the final graph $G = (C \cup S, E)$. SAP at all times maintains an optimal assignment while making a total of $O(n \min\left\{L \log^2 n, \sqrt{n} \log n\right\})$ reassignments.*

Proof. Let us define epoch i to contain all clients c such that after the insertion of c we have $\textsc{opt}(G) = i$. We now define n_i as the total number of clients added by the end of epoch i (so n_i counts clients from previous epochs as well). Extend the reduction in the proof of Theorem 3 from [2] as follows: between any two epochs, add a new copy of each server, along with all of its edges. For the following epoch, say, the ith epoch, Observation 35 tells us that regardless of what matching we had at the beginning of the epoch, the total number of reassignments performed by SAP during the epoch will not exceed $O(n_i \log^2 n_i) \subseteq O(n \log^2 n)$. We thus make at most $O(nL \log^2 n)$ reassignments in total, which completes the proof if $L < \sqrt{n}/\log n$. If $L \geq \sqrt{n}/\log n$, we make $O(n\sqrt{n} \log n)$ reassignments during the first $\sqrt{n}/\log n$ epochs. In all future epochs, note that a server at its maximum allowable load has at least $\sqrt{n}/\log n$ clients assigned to it, so there are at most $\sqrt{n} \log n$ such servers, and whenever a client is inserted the shortest augmenting path to a server below maximum load will have length $O(\sqrt{n} \log n)$. This completes the proof because there are only n augmenting paths in total. □

6.3 Approximate semi-matching

Though our result on minimizing maximum load *exactly* is nearly tight, we conclude this section on extensions with a short and cute improvement for *approximate* load balancing, which follows from the Expansion Lemma (Lemma 29).

We study a setting similar to that of [2], in which one wishes to minimize not only the maximum load, but the p-norm $|X|_p = (\sum_{s \in S} l(s)^p)^{\frac{1}{p}}$, where $l(s)$ is the load of the server s in the assignment X, for every $p \geq 1$.

First, observe that a lower bound on the p-norm comes from our necessity values α from Section 3.

Lemma 36. *For any assignment X, $(\sum_{s \in S} \alpha(s)^p)^{\frac{1}{p}} \leq |X|_p$.*

Proof. For $p = 1$, we even have equality, as we simply count the number of clients. For $p > 1$, the proof is almost identical to that of uniqueness in Section 3.2.1. Namely, assume $p > 1$, and

consider the convex program with the same constraints as in Section 3.2.1

$$0 \le x_{cs} \le 1 \qquad \forall c \in C : \sum_{s \in N(c)} x_{cs} = 1 \qquad \forall s \in S : \sum_{c \in N(s)} x_{cs} = \alpha_s$$

and where the modified objective is to minimize $\sum_{s \in S} \alpha_s^p$.

Since the objective function is strictly convex, this program always has a unique minimal solution with respect to the server loads α_s. We now observe that this solution α_s is a balanced server flow: The constraints ensure that it is a server flow, and were it not balanced, there would be some client c, who has the neighbours s and s', and who sends non-zero flow to s' although $\alpha(s) < \alpha(s')$. This would be a contradiction, because we can decrease the objective function by increasing x_{cs} and decreasing $x_{cs'}$. Since the balanced server flow is unique, the lemma follows. □

In Section 6.2, we saw that even for the ∞-norm, we cannot maintain loads below $\lceil \alpha(s) \rceil$ when limiting ourselves to logarithmic recourse. This motivates the use of approximation, and motivates the following definition:

Definition 37 ($(1+\varepsilon)$-approximate semi-matching). For each server s in the current graph, let $L(s) = \lceil (1+\varepsilon)\alpha(s) \rceil$. We say that a semi-matching is $(1+\varepsilon)$-approximate, if each server s is assigned at most $L(s)$ clients.

Assigning $(1+\varepsilon)\alpha(s)$ clients to server s would indeed give a $(1+\varepsilon)$-approximation for every p-norm. Unfortunately, $(1+\varepsilon)\alpha(s)$ may not be an integer, which is why we apply the natural ceiling operation.

As a further justification for this definition, consider the special case where all necessities are $\ge \frac{1}{\varepsilon}$. Then, for all $s \in S$ we have

$$\frac{\lceil (1+\varepsilon)\alpha(s) \rceil}{\alpha(s)} < \frac{(1+\varepsilon)\alpha(s) + 1}{\alpha(s)} = 1 + \varepsilon + \frac{1}{\alpha(s)} \le 1 + 2\varepsilon$$

and for any $(1+\varepsilon)$-approximate assignment X, where we let $l(s)$ denote the number of clients assigned to server $s \in S$, we have:

$$\left(\sum_{s \in S} l(s)^p \right)^{\frac{1}{p}} \le \left(\sum_{s \in S} \lceil (1+\varepsilon)\alpha(s) \rceil^p \right)^{\frac{1}{p}} < \left((1+2\varepsilon)^p \sum_{s \in S} \alpha(s)^p \right)^{\frac{1}{p}} = (1+2\varepsilon) \left(\sum_{s \in S} \alpha(s)^p \right)^{\frac{1}{p}}$$

But, as already noted, the p-norm of the α-vector is a lower bound on any assignment, including the optimal assignment X_{OPT}, so $|X|_p \le (1+2\varepsilon)\,|X_{\text{OPT}}|$.

In the following, let n denote the number of clients that have arrived thus far.

Theorem 38. *$(1+\varepsilon)$-approximate semi-matching has worst-case $O(\frac{1}{\varepsilon} \log n)$ reassignments with SAP.*

The proof of this theorem relies again on the Expansion Lemma. In this case, however, we do not use the α-values as part of an amortization argument, but only to bound the lengths of the shortest augmenting paths.

Proof. Given our graph G, let G' denote a similar graph with $L(s)$ copies of each server. Then any maximum matching in G' corresponds to an $(1+\varepsilon)$-approximate semi-matching in G. Now, note that each client-set K in G' has a neighborhood of at least $(1+\varepsilon)$ times its own size:

$$|N_{G'}(K)| = \sum_{s \in N_G(K)} L(s) \ge (1+\varepsilon) \sum_{s \in N_G(K)} \alpha_G(s) \ge (1+\varepsilon)\,|K|$$

Where the last inequality follows from the fact that the neighborhood of K receives at least all the flow from K, and thus, at least K flow. Thus, by Lemma 18 we can upper bound the highest alpha-value $\widehat{\alpha}$ in G' by

$$\widehat{\alpha} = \max_{\emptyset \subset K \subseteq C} \frac{|K|}{|N_{G'}(K)|} \leq \frac{1}{1+\varepsilon} = 1 - \frac{\varepsilon}{1+\varepsilon}$$

By setting $\varepsilon' = \frac{\varepsilon}{1+\varepsilon}$, all servers of G' have necessity $\leq 1 - \varepsilon'$. The Expansion Lemma (Lemma 29) then gives that any active augmenting tail have length at most $\frac{2}{\varepsilon'} \ln(n) = 2\frac{1+\varepsilon}{\varepsilon} \ln n$, which is $O(\frac{1}{\varepsilon} \log n)$. □

7 CONCLUSION

We showed that in the online matching problem with replacements, where vertices on one side of the bipartition are fixed (the servers), while those the other side arrive one at a time with all their incoming edges (the n clients), the shortest augmenting path protocol maintains a maximum matching while only making amortized $O(\log^2 n)$ changes to the matching per client insertion. This almost matches the $\Omega(\log n)$ lower bound of Grove et al. [13]. Ours is the first paper to achieve polylogarithmic changes per client; the previous best of Bosek et al. required $O(\sqrt{n})$ changes, and used a non-SAP strategy [5]. The SAP protocol is especially interesting to analyze because it is the most natural greedy approach to maintaining the matching. However, despite the conjecture of Chaudhuri et al. [8] that the SAP protocol only requires $O(\log n)$ amortized changes per client, our analysis is the first to go beyond the trivial $O(n)$ bound for general bipartite graphs; previous results were only able to analyze SAP in restricted settings. Using our new analysis technique, we were also able to show an implementation of the SAP protocol that requires total update time $O(m\sqrt{n}\sqrt{\log n})$, which almost matches the classic offline $O(m\sqrt{n})$ running time of Hopcroft and Karp [19].

The main open problem that remains is to close the gap between our $O(\log^2 n)$ upper bound and the $\Omega(\log n)$ lower bound. This would be interesting for any replacement strategy, but it would also be interesting to know what the right bound is for the SAP protocol in particular. Another open question is to remove the $\sqrt{\log n}$ factor in our implementation of the SAP protocol. Note that both of these open questions would be resolved if we managed to improve the bound in Lemma 6 from $O(n \ln(n)/h)$ to $O(n/h)$. (In the implementation of Section 5 we would then set $h = \sqrt{n}$ instead of $h = \sqrt{n}\sqrt{\log n}$.)

ACKNOWLEDGMENTS

The first author would like to thank Cliff Stein for introducing him to the problem. The authors would like to thank Seffi Naor for pointing out to us that uniqueness of server loads can be proved via convex optimization (Section 3.2.1), and to thank Martin Skutella and Guilaume Sagnol for very helpful pointers regarding the details of this proof.

REFERENCES

[1] M. Andrews, M. X. Goemans, and L. Zhang. Improved bounds for on-line load balancing. *Algorithmica*, 23(4):278–301, Apr 1999.

[2] A. Bernstein, T. Kopelowitz, S. Pettie, E. Porat, and C. Stein. Simultaneously load balancing for every p-norm, with reassignments. In *Proceedings of the 8th Conference on Innovations in Theoretical Computer Science (ITCS)*, 2017.

[3] Aaron Bernstein and Shiri Chechik. Deterministic decremental single source shortest paths: beyond the O(mn) bound. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 389–397, 2016.

[4] Aaron Bernstein and Liam Roditty. Improved dynamic algorithms for maintaining approximate shortest paths under deletions. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2011, San Francisco, California, USA, January 23-25, 2011*, pages 1355–1365, 2011.

[5] B. Bosek, D. Leniowski, P. Sankowski, and A. Zych. Online bipartite matching in offline time. In *55th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 384–393. IEEE Computer Society, 2014.

[6] B. Bosek, D. Leniowski, P. Sankowski, and A. Zych. Shortest augmenting paths for online matchings on trees. In *Approximation and Online Algorithms: 13th International Workshop, WAOA 2015, Patras, Greece, September 17-18, 2015. Revised Selected Papers*, pages 59–71, Cham, 2015. Springer International Publishing.

[7] B. Bosek, D. Leniowski, A. Zych, and P. Sankowski. The shortest augmenting paths for online matchings on trees. *CoRR*, abs/1704.02093, 2017.

[8] K. Chaudhuri, C. Daskalakis, R. D. Kleinberg, and H. Lin. Online bipartite perfect matching with augmentations. In *The 31st Annual IEEE International Conference on Computer Communications (INFOCOM)*, pages 1044–1052. IEEE, 2009.

[9] E. A. Dinic. Algorithm for Solution of a Problem of Maximum Flow in a Network with Power Estimation. *Soviet Math Doklady*, 11:1277–1280, 1970.

[10] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, April 1972.

[11] Leah Epstein and Asaf Levin. *Robust Algorithms for Preemptive Scheduling*, pages 567–578. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[12] Ashish Goel, Michael Kapralov, and Sanjeev Khanna. On the communication and streaming complexity of maximum bipartite matching. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 468–485, 2012.

[13] E. F. Grove, M.-Y. Kao, P. Krishnan, and J. S. Vitter. Online perfect matching and mobile computing. In S. G. Akl, F. Dehne, J.-R. Sack, and N. Santoro, editors, *Algorithms and Data Structures*, pages 194–205. Springer, Berlin,, 1995.

[14] Albert Gu, Anupam Gupta, and Amit Kumar. The power of deferral: Maintaining a constant-competitive steiner tree online. In *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing*, STOC '13, pages 525–534, New York, NY, USA, 2013. ACM.

[15] A. Gupta, A. Kumar, and C. Stein. Maintaining assignments online: Matching, scheduling, and flows. In *Proceedings of the Twenty-fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '14, pages 468–479, Philadelphia, PA, USA, 2014. Society for Industrial and Applied Mathematics.

[16] Anupam Gupta, Ravishankar Krishnaswamy, Amit Kumar, and Debmalya Panigrahi. Online and dynamic algorithms for set cover. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2017, pages 537–550, New York, NY, USA, 2017. ACM.

[17] Anupam Gupta and Amit Kumar. Online steiner tree with deletions. In *Proceedings of the Twenty-fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '14, pages 455–467, Philadelphia, PA, USA, 2014. Society for Industrial and Applied Mathematics.

[18] P. Hall. On representatives of subsets. *Journal of the London Mathematical Society*, s1-10(1):26–30, 1935.

[19] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[20] Makoto Imase and Bernard M. Waxman. Dynamic steiner tree problem. *SIAM Journal on Discrete Mathematics*, 4(3):369–384, 1991.

[21] Michael Kapralov. Better bounds for matchings in the streaming model. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1679–1697, 2013.

[22] R. M. Karp, U. V. Vazirani, and V. V. Vazirani. An optimal algorithm for on-line bipartite matching. In *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*, STOC '90, pages 352–358, New York, NY, USA, 1990. ACM.

[23] Valerie King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 81–91. IEEE Computer Society, 1999.

[24] Jakub Łącki, Jakub Ocwieja, Marcin Pilipczuk, Piotr Sankowski, and Anna Zych. The power of dynamic distance oracles: Efficient dynamic algorithms for the steiner tree. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 11–20, 2015.

[25] A. Madry. Navigating central path with electrical flows: From flows to matchings, and back. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 253–262, Oct 2013.

[26] Nicole Megow, Martin Skutella, José Verschae, and Andreas Wiese. The power of recourse for online mst and tsp. *SIAM Journal on Computing*, 45(3):859–880, 2016.

[27] S. Phillips and J. Westbrook. On-line load balancing and network flow. *Algorithmica*, 21(3):245–261, Jul 1998.

[28] Peter Sanders, Naveen Sivadasan, and Martin Skutella. Online scheduling with bounded migration. *Math. Oper. Res.*, 34(2):481–498, 2009.
[29] Yossi Shiloach and Shimon Even. An on-line edge-deletion problem. *J. ACM*, 28(1):1–4, January 1981.
[30] Martin Skutella and José Verschae. A robust PTAS for machine covering and packing. In Mark de Berg and Ulrich Meyer, editors, *Algorithms - ESA 2010, 18th Annual European Symposium, Liverpool, UK, September 6-8, 2010. Proceedings, Part I*, volume 6346 of *Lecture Notes in Computer Science*, pages 36–47. Springer, 2010.
[31] Subhash Suri, Csaba D. Tóth, and Yunhong Zhou. Selfish load balancing and atomic congestion games. *Algorithmica*, 47(1):79–96, 2007.
[32] Jeffery Westbrook. Load balancing for response time. *Journal of Algorithms*, 35(1):1 – 16, 2000. Announced at ESA'95.