$(1 + \varepsilon)$-Approximate Incremental Matching in Constant Deterministic Amortized Time*

Fabrizio Grandoni†, Stefano Leonardi‡, Piotr Sankowski§

Chris Schwiegelshohn¶, Shay Solomon‖

Abstract

We study the matching problem in the incremental setting, where we are given a sequence of edge insertions and aim at maintaining a near-maximum cardinality matching of the graph with small update time. We present a deterministic algorithm that, for any constant $\varepsilon > 0$, maintains a $(1+\varepsilon)$-approximate matching with constant amortized update time per insertion.

1 Introduction

Let $G = (V, E)$ be an n-node m-edge undirected graph. Finding a large cardinality matching in G is a fundamental optimization problem. For bipartite graphs, the currently best available time bounds are $O(m\sqrt{n})$ due to Hopcroft and Karp [21], $O(n^{\omega})$ due to Mucha and Sankowski [29] and $\tilde{O}(m^{10/7})$ due to Madry [27]. The former two algorithms have been extend to finding matchings in general (non-bipartite) graphs as well [28, 29].

In contrast to this static case (where the graph is given up-front), there has been recently a lot of interest in the *dynamic matching* problem. In dynamic setting we must maintain a (near-)optimal matching as the graph changes over time. Most of the results have been given in the *fully-dynamic* model where edges are added or deleted over time. It is known how to maintain the size of the maximum matching with $O(n^{1.495})$ worst-case update time [33]. And we known that maintaining the exact value of the maximum matching requires polynomial update time under reasonable complexity conjectures [2, 20, 26]. Hence, we turn our attention to approximate matchings. In this case we know how to maintain 2-approximate matchings with constant amortized update time [34], but

*This work was done in part while a subset of the authors was visiting the *Algorithms and Uncertainty* and *Bridging Discrete and Continuous Optimization* programs at the Simons Institute for the Theory of Computing. F. Grandoni is partially supported by the SNSF Grant 200021_159697/1 and the SNSF Excellence Grant 200020B_182865/1. S. Leonardi and C. Schwiegelshohn are partially supported by the ERC Advanced Grant 788893 AMDROMA. P. Sankowski is partially supported by ERC Consolidator Grant 772346 TUgbOAT and Polish National Science Centre grant 2014/13/B/ST6/00770.

†IDSIA, USI-SUPSI

‡Sapienza University of Rome

§University of Warsaw

¶Sapienza University of Rome

‖Tel Aviv University

algorithms achieving better-than-2 approximations all require polynomial update time. In particular, we can maintain a $(1+\varepsilon)$-approximate matching in the fully-dynamic setting with update time $O(\sqrt{m}/\varepsilon^2)$ [18], a $(3/2+\varepsilon)$-approximate matching with update time $O(m^{1/4}/\varepsilon^{2.5})$ [8], and for every sufficiently large integer K, an α_K-approximation to the matching size with update time $O(n^{2/K})$ where $\alpha_K \in (1,2)$. (We survey the existing results in detail in Section 1.2.) This suggests the question: can we achieve better approximation in some natural dynamic settings?

In this paper, we consider the *incremental model* for dynamic algorithms, where the edges of the graph can only be inserted but not deleted. We show that in this case we can give much stronger results than in the fully-dynamic model:

Theorem 1.1. *Given a sequence of edge insertions to a graph G and a constant $\varepsilon > 0$, there exists a deterministic algorithm that maintains a $(1+\varepsilon)$-approximate matching with $O_\varepsilon(1)$ amortized update time per insertion.*

We remark that by [26], maintaining a maximum matching requires polynomial amortized update time even in the incremental case assuming the 3-SUM conjecture. Hence, our result is asymptotically optimal, up to deamortization.

The only previous result for approximate matchings in the incremental model is due to Gupta [16], who gave an amortized $O(\log^2 n)$ update-time algorithm to maintain $(1+\varepsilon)$-approximate matchings in *bipartite* graphs. Hence, we improve the update time from polylogarithmic to constant. Moreover, we also extend the result from bipartite graphs to general graphs.

1.1 Our Techniques

As usual for approximate matching, our starting point is the well-known fact that a given matching M is a $1+\frac{1}{\ell}$ approximation of the maximum matching OPT if there are no length $2\ell+1$ (or shorter) augmenting paths with respect to M. Hence it is sufficient to search for a matching OPT_ℓ with the above property for $\ell = 1/\varepsilon$.

One simple way to obtain OPT_ℓ is to use the following variant of Edmonds [14] algorithm. Imagine each undirected edge $\{a,b\}$ as two oppositely directed edges ab and ba. Now our goal is to find a short *directed* augmenting path, i.e., a path $P = (a_0, \dots, a_{2q+1})$, $q \le \ell$, where each edge $\{a_i, a_{i+1}\}$ belongs to the matching iff i is odd. If we find one such P, we replace the current matching M by $M \oplus P$[1], and iterate.

One simple way to search for P is roughly as follows. For each free node v, we build an *alternating path tree* T_v recursively in the following way. Let a be a given node, starting with $a = v$. We first search for a free neighbor b of a such that the v-a path in T_v plus ab induces an augmenting path. Otherwise, we expand the subtree of T_v rooted at a by adding paths of type abc, with ab unmatched and bc matched, and continue recursively on each such node c (unless c is at level[2] 2ℓ already). In particular, if we do not find any augmenting path, T_v at the end will have at most 2ℓ levels, where even (resp., odd) levels contain matched (resp., unmatched) edges only. Observe also that all

[1] With a slight notational abuse, we use P to denote both a directed path and its undirected variant.

[2] The level $\ell_T(v)$ of node $v \in T$ is v's hop-distance from the root of T, and the level $\ell_T(ab)$ of a directed edge $ab \in T$ is the level $\ell_T(b)$ of its highest level endpoint.

nodes in T_v but the root are matched, and all nodes at odd levels have precisely one child. The above procedure has the advantage that it avoids blossom contractions[3]. In particular, it works for general graphs. Unfortunately it is also very slow – its running time is $\Omega(n^\ell)$.

Our high-level approach is to maintain a *partial* version of the above alternating path trees T_v, that can be updated very efficiently under insertion of edges. While our approach does not allow us to discover all the augmenting paths of length up to $2\ell+1$, we are able to guarantee that any node-disjoint set of missed augmenting paths of that type has relatively small cardinality w.r.t. the size of the current matching. Hence missing those augmenting paths has a negligible impact on the approximation factor.

Let us describe our approach in more detail, starting with the simpler bipartite case. We exploit two main ideas. The first critical idea is to limit the degree of nodes in each T_v to some large enough constant Δ depending on ℓ. In particular, for a given node a in the above recursive construction, in the case that we do not find an augmenting path containing a, we only add up to Δ paths of type abc. Note that now T_v contains at most $O(\Delta^\ell) = O_\varepsilon(1)$ nodes. Furthermore, with some extra work we guarantee that each directed matched edge ab appears in at most one tree T_v at level i, for each possible even value of i. To see why this is helpful, imagine that we miss some augmenting path P because one of its nodes a appears at level i in some tree T_v where a has already degree Δ. Note that in this case we might miss discovering P. However, path P can increase the matching at most by 1. We charge a fraction $1/\Delta$ of this loss to each one of the Δ matching edges that appear at level $i+2$ in the subtree rooted at a. Each matching edge can be charged by at most 2ℓ node-disjoint paths this way (using the fact that each edge appears in at most two directions and ℓ trees per direction), hence the total charge is at most $\frac{2\ell}{\Delta}$: this is $O(\varepsilon)$ for large enough Δ.

A more subtle problem arises when we do find an augmenting path P. In that case we destroy the trees T_v that intersect P and rebuild them. This operation costs $\Omega(deg(a))$ per reinserted node a, hence we cannot do that too frequently. Here we exploit our second main idea. We introduce *counters* $C_i[a]$ that are incremented each time a node a is removed from some tree T_v where it appears at level i. We stop inserting a at level i in trees when $C_i[a]$ reaches a large enough constant C depending on ℓ. This way reinsertions have constant amortized cost. Using a *global counting argument*, we can show that, for C large enough, the total loss due to (node-disjoint) augmenting paths which are not discovered because one of their nodes reached the counter threshold is $O(\varepsilon)$ times the size of the current matching. The intuition is as follows. Each previously discovered augmenting path P implies an increase of the matching size by one. We interpret this increase as 1 credit, that we uniformly distribute among all the nodes of all the trees that we need to rebuild because of P. Note that there are constantly many such trees (since the length of P is bounded and nodes are duplicated a constant number of times) and each such tree contains constantly many nodes (due to the degree and depth bound of each T_v). Hence each affected node receives a constant fraction of 1 credit. When the fractional credits accumulated at v reach a total of $\Omega(1/\varepsilon)$, these credits can be used to

[3] In some sense, each encountered blossom is *traversed* in both directions.

compensate the loss due to any future augmenting path involving v.

In the case of general graphs the requirement that each directed matching edge appears in at most one copy per level i is too restrictive: indeed, it might happen that we fail to discover an augmenting path not due to the degree or counter constraints, but because of the presence of a blossom. To better understand this issue, consider the following scenario. Consider an augmenting path $u'\alpha\beta u''$, and assume that edge $\{u'', \alpha\}$ exists. With the algorithm for the bipartite case we might have $\alpha\beta$ appearing at level 2 in $T_{u''}$. This would prevent us from adding $\alpha\beta$ at level 2 in $T_{u'}$, and at the same time the path in $T_{u''}$ from u'' to β cannot be extended to an augmenting path by adding edge $\beta u''$. This specific issue can be addressed by allowing $\alpha\beta$ to appear at level 2 in two different trees (with distinct roots), but this is not sufficient in general.

In order to address the above problem, we introduce a notion of *simple path covering* that, to the best of our knowledge, is new and might be of independent interest. For two paths P and P' (represented as a sequence of nodes) starting and ending at some node s, resp., let $P \circ P'$ denote their concatenation. We show that any set $\mathcal{U}$ of simple paths of length κ ending at some node s, contains a subset $\mathcal{C}$ of size at most $(\kappa+1)^{\kappa'}$ such that the following covering property holds: given any path P' of length κ' starting at s and a path $P \in \mathcal{U}$ such that $P \circ P'$ is simple, then there exists some $P'' \in \mathcal{C}$ with the same property. Furthermore $\mathcal{C}$ can be computed efficiently with a greedy algorithm.

Intuitively, in our case $\mathcal{U}$ will be the set of alternating paths of length κ starting at some free node and ending at some node s, so that there exists some augmenting path $P \circ P'$ of length $\kappa + \kappa' \leq 2\ell + 1$ with $P \in \mathcal{U}$ as a prefix. Our construction shows that it is sufficient to maintain the cover $\mathcal{C}$ of $\mathcal{U}$ in our trees T_v. In turn, this can be achieved by allowing each directed edge ab to appear in up to $\ell^{O(\ell)}$ trees at the same level. This affects the values of the parameters Δ and C and the running time only by a constant factor (depending on ℓ).

1.2 Other Related Work

Given the existence of a polynomial lower bounds for maintaining even the value of the maximum matching (see below), approximate matching algorithms have been studied.

The Fully Dynamic Setting: Onak and Rubinfeld [31] gave an $O(1)$-approximation in amortized $O(\log^2 n)$ update time; this was improved by Bhattacharya et al. [9, 10] to a deterministic $(2+\varepsilon)$-approximation with polylogarithmic amortized update time. Extending prior work by Ivkovic and Lloyd [22] and Baswana et al. [5], Solomon [34] gave an algorithm to maintain a maximal (hence 2-approximate) matching with high probability in amortized constant update time. For worst-case update times, Bhattacharya et al. [9] showed $(2+\varepsilon)$-approximation for the fractional setting with update time $O(\log^3 n)$. This was recently extended to the integral setting independently by Arar et al. [3] and Charikar and Solomon [12]. For bipartite graphs, Bernstein and Stein [7] gave a $(3/2+\varepsilon)$-approximation with update time $O(\sqrt[4]{m})$. Building on work by Neiman and Solomon [30], Gupta and Peng [18] showed a $(1+\varepsilon)$-approximation with update time $O(\sqrt{m}/\varepsilon^2)$; this was improved for low arboricity graphs by Peleg and Solomon [32].

The Incremental Setting: The incremental setting has received much less attention. As mentioned above, Gupta [16] gave a $(1+\varepsilon)$-approximation for this setting, with amortized $O(\log^2 n)$-update time. His approach is based on the multiplicative-weight-update method, which makes it unlikely that an improved analysis will yield constant update times. Moreover, his approach is based on maintaining fractional matchings, which makes it harder to extend it to the non-bipartite case. Recently, Gupta and Khan [17] gave an algorithm for maintaining an exact matching with an amortized update time of $O(n)$, which is essentially optimal (see below for lower bounds). Other incremental models have been considered in the online algorithms literature, e.g., the *bipartite vertex-arrival* model of Karp, Vazirani, and Vazirani [24]. In this setting, Bosek et al. [11] give algorithms matching the runtime of Hopcroft-Karp [21], and Bernstein et al. [6] bound the number of edge-changes. Solomon also studied the number of edge-changes in the fully dynamic and incremental setting [35]. Edge-arrivals have also been studied in the streaming and online model: while better-than-2 results are known for random-order models [25, 4, 19], nothing better than a factor-2 approximation is known for the case of adversarial arrivals; also, see [15, 23] for some lower bounds in these settings.

Lower Bounds: Abboud and Williams [2] gave polynomial lower bounds on the update time when maintaining a maximum bipartite matching under different conjectures: their lower bounds are worst-case in the incremental or decremental case, and amortized in the fully dynamic case. Henzinger et al. [20] gave a stronger lower bound of $\Omega(m^{1/2-o(1)})$ for the mentioned cases under the OMv conjecture. Kopelowitz et al. [26] showed that maintaining a maximum matching in incremental or decremental graphs requires amortized $\Omega(n^{0.333-o(1)})$ update time assuming the 3-SUM conjecture. Dahlgaard [13] showed that even for planar bipartite graphs, no algorithm to maintain a maximum matching in the incremental setting can have amortized $O(n^{1-\varepsilon})$ update time under OMv (see also [1]).

2 Preliminaries

Let $G=(V,E)$ be an unweighted undirected graph. Given a subgraph G' (possibly described as a subset of edges), we denote by $V(G')$ and $E(G')$ its node and edge set, resp. In order to simplify the notation, we sometimes use G' instead of $V(G')$ or $E(G')$ when the meaning is clear from the context. We denote the neighborhood of a node v by $N(v)$.

A *matching* is a set of edges $M \subseteq E$ such that no two edges of M share a common node. We call the nodes in $V(M)$ matched, and the remaining nodes free or unmatched. Similarly, edges in M are matched, and the remaining edges are unmatched. By OPT we denote a matching of maximum cardinality. An *alternating path* is a path whose edges alternate between unmatched and matched ones. An *augmenting path* is an alternating path whose endpoints are both free. We denote the symmetric difference of two sets A and B by $A \oplus B := (A \cup B) \setminus (A \cap B)$. If P is an augmenting path with respect to M, then $P \oplus M$ is a matching. For a collection of node-disjoint paths $\mathcal{P}$, we use $\mathcal{P}$ also to denote the union of their edges. The following claim follows from standard matching theory.

Lemma 2.1. *Let M be a matching and let OPT denote an optimal matching. Then for any ℓ there exists a set of node-disjoint augmenting paths $\mathcal{P}_\ell$ of length at most $2\ell+1$ such that $\frac{\ell+1}{\ell} \cdot |M \oplus \mathcal{P}_\ell| \geq |OPT|$.*

Proof. Let $\mathcal{Q} := M \oplus OPT$. $\mathcal{Q}$ consists of even length alternating paths and cycles, or augmenting paths. The former two we can ignore, as they do not increase the size of the matching. Let $\mathcal{P}_\ell$ be the set of augmenting paths of length at most $2\ell+1$ of $\mathcal{Q}$. Then $\mathrm{OPT}_\ell := M \oplus \mathcal{P}_\ell$ has no augmenting paths of length at most $2\ell+1$. It is well-known (see, e.g., [21]) that the latter condition implies $\frac{\ell+1}{\ell}|\mathrm{OPT}_\ell| \geq |\mathrm{OPT}|$. $\square$

Given a rooted tree T and a node $v \in V(T)$, by $deg_T(v)$ we denote the number of children of v, and by $lev_T(v)$ the level of v (with root at level 0). We say that an edge of T is at level i if its *bottom* endpoint is at that level.

Proofs and detailed that are omitted from this extended abstract will appear in the full version of the paper.

3 The Incremental Algorithm: Bipartite Graphs

In this section we will focus on the case of bipartite graphs. This will allow us to introduce part of the main ideas, while avoiding some technical complications due to the presence of blossoms in the general case.

The graph G is represented via lists of neighbouring nodes. As usual in the incremental setting, we assume that the graph initially contains no edges. Furthermore, for simplicity, we assume that the set of nodes is known and fixed a priori (with corresponding data structures correctly initialized). The second assumption can be removed by standard doubling techniques, with an additive constant amortized cost per insertion.

Recall that for each (undirected) edge $\{a, b\}$ we consider its two oppositely directed versions ab and ba, and search for directed augmenting paths, i.e., directed paths $P = (a_0, \ldots, a_{2q+1})$ where all edges of type $\{a_{2i}, a_{2i+1}\}$ are unmatched and all edges of type $\{a_{2i+1}, a_{2i+2}\}$ are matched.

We will store multiple copies a' of the same node a in different trees T_v. In order to simplify the notation, we will simply denote one such copy by a when the meaning is clear from the context.

3.1 The Variables

In the following C, Δ, and ℓ are constant parameters depending on ε to be fixed later. The current approximate matching is denoted by M. We maintain the following variables.

- For each matched node $v \in V(M)$, its mate `mate`$[v]$ (i.e. $\{v, \texttt{mate}[v]\} \in M$). We set `mate`$[v] =$ NULL if v is free.
- For each free node $v \notin V(M)$, one *alternating path tree* T_v initially containing v only. Intuitively, these trees are used to discover (directed) augmenting paths having v as an endpoint. In the following we consider the degree $deg_{T_v}(w)$ and

level $lev_{T_v}(w)$ of w in T_v as updated implicitly. By $T_v(a)$ we denote the v-a path in T_v.

- For each level $i = 0, 1, \ldots, 2\ell$ and each node $a \in V$, the root $R_i[a] = v$ of the tree T_v containing a at level i (NULL if there is no such tree).
- For each level $i = 0, 1, \ldots, 2\ell$ and each node $a \in V$, an integer *counter* $C_i[a]$ initialized to 0. Intuitively, the sum of the counters is an estimate of the current matching size up to constant factors.

We critically maintain the following invariant for the trees T_v.

Invariant 3.1 (Tree Invariant)**.** *Each tree T_v is maximal w.r.t. the following constraints under insertion of edges:*

- **(Alternating Path)** *For each leaf $a \in T_v$, $T_v(a)$ is an even-length duplicate-free alternating path. Furthermore, no node a at even level in T_v is adjacent to a free node $b \notin T_v(a)$.*
- **(Depth)** *The depth of each T_v is at most 2ℓ.*
- **(Degree)** *The maximum degree of each T_v is at most Δ.*
- **(Counter)** *No T_v contains a node w at level i with $C_i[w] \geq C$.*
- **(Duplication)** *For each level i and node a, a can appear in at most one tree T_v at level i (hence $R_i[a]$ is well defined).*

We will assume that Invariant 3.1 holds before each edge insertion, and we will later show how to restore it after the insertion of some edge. Note that the first 2 properties are essentially the same as those in the previously described variant of Hopcroft-Karp algorithm, while the last 3 properties are a novelty of our approach.

3.2 The Procedures

Upon insertion of an edge $\{a', b'\}$ we execute the main procedure $insert(\{a', b'\})$ which is described in Figure 1. This procedure exploits two global variables P_{aug} and V_{exp}.

Variable P_{aug} is used to store any discovered augmenting path (of length at most $2\ell + 1$). Variable V_{exp} is a vector indexed by levels $i \in \{0, 1, \ldots, 2\ell + 1\}$. Each $V_{exp}[i]$, $i \geq 1$, contains a list of directed edges bc. Intuitively, each such bc is an edge that can be potentially inserted at level i in some tree T_v. Furthermore, in the case that bc belongs to (or is inserted in) some tree T_v, it is possible that the subtree rooted at c is not maximal. As a boundary case, $V_{exp}[0]$ contains pairs of type vv. Intuitively, this corresponds to nodes v for which we have to reconstruct the entire tree T_v.

Procedure $insert()$ adds $\{a', b'\}$ to G, and initializes P_{aug} and V_{exp} to the empty set[4] (lines 1-2). Then (lines 3-4) it adds $a'b'$ and $b'a'$ to $V_{exp}[i]$ for each odd level i. Intuitively, these are (unmatched) edges that wish to be added to some T_v for the first time. Finally it executes a while loop (lines 5-11) that iterates as long as at least one of P_{aug} or V_{exp} is not empty. In each execution of the loop, it first checks if $P_{aug} \neq$ NULL,

[4]For V_{exp} this means that all its entries are empty lists.

insert($\{a', b'\}$)
1: Add $\{a', b'\}$ to G
2: $P_{aug} \leftarrow \emptyset$, $V_{exp} \leftarrow \emptyset$
3: **for** $i \in \{1, 3, \ldots, 2\ell + 1\}$ **do**
4: $\quad V_{exp}[i] \leftarrow V_{exp}[i] \cup \{a'b', b'a'\}$;
5: **while** $P_{aug} \neq \emptyset \vee V_{exp} \neq \emptyset$ **do**
6: $\quad$ **if** $P_{aug} \neq \emptyset$ **then**
7: $\quad\quad$ *augment*();
8: $\quad\quad P_{aug} \leftarrow \emptyset$;
9: $\quad$ **else**
10: $\quad\quad$ Extract bc from non-empty $V_{exp}[i]$ with minimum i;
11: $\quad\quad$ *expand*(bc, i);

Figure 1: Procedures *insert*().

in which case it calls the subroutine *augment*() and then resets P_{aug} to NULL (lines 6-8). Otherwise (lines 9-11), it extracts bc from the non-empty $V_{exp}[i]$ with minimum i, and calls $expand(bc, i)$.

The subroutine *augment*() (see Figure 2) is intuitively used to *implement* the augmenting path $P_{aug} = (a_0, \ldots, a_{2q+1})$. This procedure updates the matching to $M \oplus P_{aug}$ (line 3). Furthermore, it *destroys* each tree T_v that intersect P_{aug}, which involves the following operations[5]. It increments the counter $C_i[w]$ of any node $w \in T_v$ appearing at level i (lines 4-5). Then (lines 6-7) it adds to $V_{exp}[i]$ each edge $bc \in E(T_v) \cap M$ at even level i. Note that these edges do not belong to P_{aug} (due to the update of the matching in line 3). It also adds (lines 8-9) all the edges of $P_{aug} \cap M$, in both directions, to $V_{exp}[i]$ for each even $i \geq 2$. These are newly created matching edges that might be inserted potentially at any even level. Finally, in lines 10-15, the procedure sets each involved tree T_v to $(\{v\}, \emptyset)$ if v is free and $C_0[v] < C$, and to NULL otherwise. In the first case it also adds vv to $V_{exp}[0]$ to recall that the tree T_v has to be reconstructed. The R_j's are updated in an obvious way.

The recursive subroutine $expand(c, i)$ is described in Figure 3. Intuitively, this is the subroutine that is used to construct the trees T_v, and to keep them maximal. It gets a pair bc and a level $i \geq 0$ (with $b = c$ for $i = 0$). This procedure halts if the counters of b or c for the associated level reach the threshold C, and also if $P_{aug} \neq$ NULL or $i \geq 2\ell + 2$.

Lines 3-11 apply to the case that i is odd. Their goal is to insert the unmatched edge bc at level i in up to one tree T_v if possible without violating the Tree Invariant. Intuitively, bc corresponds to some newly inserted edge $\{a', b'\}$ introduced in lines 3-4 of *insert*(). In this case b needs to be already contained in some tree T_v at level $i - 1$. Lines 5-6 check if c closes an augmenting path in T_v. If not (lines 7-11), the procedure tries to add a path of type bcd to T_v, if this is possible respecting the Tree Invariant. In that case, it calls recursively $expand(cd, i + 1)$.

[5] A "partial destruction" of trees would also work, but we here consider the total destruction case to simplify the presentation.

augment()

1: Let $P_{aug} = (a_0, \dots, a_{2q+1})$;
2: Let $r(P_{aug})$ be the set of roots of trees T_v containing some node in P_{aug};
3: Update `mate` according to $M \leftarrow M \oplus P_{aug}$;
4: **for** each $v \in r(P_{aug})$ and each $w \in T_v$, with $i := lev_v(w)$ **do**
5: $\quad C_i[w] \leftarrow C_i[w] + 1$;
6: **for** each $v \in r(P_{aug})$ and each $bc \in E(T_v) \cap M$ at even level i **do**
7: $\quad V_{exp}[i] \leftarrow V_{exp}[i] \cup \{bc\}$;
8: **for** each $bc \in P_{aug} \cap M$ and each $i \in \{2, 4, \dots, 2\ell\}$ **do**
9: $\quad V_{exp}[i] \leftarrow V_{exp}[i] \cup \{bc, cb\}$;
10: **for** each $v \in r(P_{aug})$ **do**
11: $\quad$ **if** v is free $\wedge$ $C_0[v] < C$ **then**
12: $\quad\quad$ Set $T_v \leftarrow (\{v\}, \emptyset)$ and update R_i's;
13: $\quad\quad$ $V_{exp}[0] \leftarrow V_{exp}[0] \cup \{vv\}$;
14: $\quad$ **else**
15: $\quad\quad$ Set $T_v \leftarrow$ NULL and update R_i's;

Figure 2: Procedure *augment*().

Lines 12-15 apply to the case that $i \geq 2$ is even and $bc \in M$. Here the procedure tries to append bc at level i in some tree T_v if this does not violate the Tree Invariant. Intuitively, this corresponds to the case that bc is either a *newly* created matching edge, or some already existing matching edge that used to belong to a tree that was destroyed by *augment*(). In both case adding bc to some tree might be needed to restore maximality.

Lines 16-24 apply to the case that i is even and $bc \in M$ (hence $i \geq 2$) or $b = c$ is free. Then, if c is contained at level i in some T_v, the procedure tries to find an augmenting path containing $T_v(c)$ (lines 18-19). If such path is not found, the procedure expands in a maximal way the subtree of T_v rooted at c (lines 20-24). This is done by adding paths of type cde, whenever possible without violating the Tree Invariant, and calling $expand(de, i+2)$ recursively.

3.3 Analysis

The slightly technical proof of this lemma is deferred to Section A in the appendix.

Lemma 3.2. *The Tree Invariant holds at the end of each execution of insert().*

Let us next analyze the approximation factor of the algorithm. We first observe the following direct consequence of Lemma 3.2.

Lemma 3.3 (Witness Lemma)**.** *Let P be an augmenting path of length at most $2\ell + 1$ undetected by the algorithm. Then one of the following two conditions holds for some $w \in V(P)$:*

1. *A copy of w appears in some T_v and $deg_{T_v}(w) = \Delta$.*
2. *A copy of w appears in some T_v at level i and $C_i[w] \geq C$.*

Proof. Assume for the sake of contradiction that there exists an augmenting path P not satisfying the two conditions. We consider the directed augmenting path $P =$

expand(bc, i)

if $i \geq 2\ell + 2 \vee P_{aug} \neq \textsc{null} \vee C_i[c] \geq C \vee C_{i-1}[b] \geq C$ **then**
    halt;
if i is odd $\wedge\ bc \notin M$ **then**
    if $v := R_{i-1}[b] \neq \textsc{null}$ **then**
        if c is free **then**
            Set $P_{aug} \leftarrow T_v(b) \circ (b, c)$ and **halt**;
        else
            Let $d = \texttt{mate}[c]$;
            if $deg_{T_v}[b] < \Delta \wedge R_{i+1}[d] = \textsc{null} \wedge C_{i+1}[d] < C \wedge c, d \notin T_v(b)$ **then**
                Add $\{bc, cd\}$ to T_v and update R_j's;
                expand$(cd, i + 1)$;
if $i \geq 2$ is even $\wedge\ bc \in M$ **then**
    for all neighbors a of b **do**
        if $v := R_{i-2}[a] \neq \textsc{null} \wedge deg_{T_v}(a) < \Delta \wedge R_i[c] = \textsc{null} \wedge b, c \notin T_v(a)$ **then**
            Add $\{ab, bc\}$ to T_v and update R_j's;
if i is even $\wedge$ ($bc \in M \vee b = c$ is free) $\wedge\ v := R_i[c] \neq \textsc{null}$ **then**
    for all neighbors d of c **do**
        if d is free **then**
            Set $P_{aug} \leftarrow T_v(c) \circ (c, d)$ and **halt**;
        else
            Let $e = \texttt{mate}[d]$;
            if $deg_{T_v}(c) < \Delta \wedge R_{i+2}[e] = \textsc{null} \wedge C_{i+1}[d] < C \wedge C_{i+2}[e] < C \wedge d, e \notin T_v(c)$ **then**
                Add $\{cd, de\}$ to T_v and update R_j's;
                expand$(de, i + 2)$;

Figure 3: Procedure *expand*(), bipartite graphs.

$(u', \alpha_1, \beta_1, \alpha_2, \beta_2, \ldots, \alpha_k, \beta_k, u'')$, with $k \leq \ell$, and let $e_i = \alpha_i \beta_i$. Observe that, by construction, whenever a node is matched, it remains matched for the rest of the algorithm.

We prove by induction that for each e_i, there exists a tree T_{x_i} containing e_i at level $2i$. This easily implies a contradiction. Indeed, $T_{x_k}(\beta_k) \circ (\beta_k, u'')$ would be an augmenting path undetected by the algorithm, contradicting the Alternating Path invariant (note that it cannot be $x_k = u''$ since bipartite graphs do not contain blossoms).

For the base case e_1, if e_1 is contained at level 2 in some tree $T_{u'}$ the claim holds with $x_1 = u'$. Otherwise, observe that the path (u', α_1, β_1) satisfies the constraints Degree, Counter, Alternating Path and Depth w.r.t. $T_{u'}$. Hence, by the maximality of $T_{u'}$ implied by Lemma 3.2, the only reason why this path is not added to $T_{u'}$ is because e_1 is already contained at level 2 in some other tree, a contradiction.

The inductive step follows analogously. Assume the claim holds up to edge e_i, $i < \ell$, and consider edge e_{i+1}. By assumption e_i is contained at level $2i$ in some tree T_{x_i}. If T_{x_i} also contains e_{i+1} at level $2i + 2$ the claim holds with $x_{i+1} = x_i$. Otherwise, observe that adding the path $(\beta_i, \alpha_{i+1}, \beta_{i+1})$ to T_{x_i} would not violate the constraints Degree, Counter, Alternating Path and Depth w.r.t. T_{x_i}. Hence, again by the maximality of T_{x_i}, edge e_{i+1} must be contained at level $2i + 2$ in some other tree $T_{x_{i+1}}$. □

Lemma 3.4. *At the end of each insert(),* $\frac{|OPT|}{|M|} \leq \frac{\ell+1}{\ell}(1 + \frac{2\ell}{\Delta} + \frac{16\ell^2 \Delta^\ell}{C})$.

Proof. Let $\mathcal{P}_\ell$ be the set of node-disjoint augmenting paths guaranteed by Lemma 2.1 w.r.t. M. We have

$$\frac{|\mathrm{OPT}|}{|M|} \leq \frac{\ell+1}{\ell}\frac{|M|+|\mathcal{P}_\ell|}{|M|} \tag{1}$$

By the Witness Lemma 3.3, we can partition $\mathcal{P}_\ell$ into the following two subsets:

- The paths $\mathcal{P}^{degree} \subseteq \mathcal{P}_\ell$ that satisfy the degree condition of Lemma 3.3.
- The remaining paths $\mathcal{P}^{count} = \mathcal{P}_\ell \setminus \mathcal{P}^{degree}$ that (have to) satisfy the counter condition of Lemma 3.3.

We next upper bound the size of these two sets in terms of $|M|$ and of the constant parameters of the algorithm. For a path $P \in \mathcal{P}^{degree}$, let us choose arbitrarily exactly one copy w_P of some node of P that appears in some tree T_v with degree Δ at some (even) level i. Let M_P be the Δ matching edges that descend from w_P and appear at level $i+2$. We charge each one of these edges by an amount $1/\Delta$. Intuitively, this corresponds to a distribution of the increase of the matching size (by 1) due to P. Observe that each directed matching edge at some level i can be charged at most once by the node disjointness of the augmenting paths $\mathcal{P}_\ell$ and by the Duplication constraint. Hence each matching edge is charged by at most $2\ell/\Delta$. It follows that

$$|\mathcal{P}^{degree}| \leq \frac{2\ell}{\Delta}|M|. \tag{2}$$

Consider next $\mathcal{P}^{count}$. Each augmenting path P discovered by the algorithm increases the matching size by precisely 1. The corresponding total increase of counters equals the number $n(P)$ of (copies of) nodes in the trees T_v, $v \in r(P)$, destroyed because of P. One has

$$\begin{aligned} n(P) = \sum_{v\in r(P)} |V(T_v)| &\leq \sum_{v\in r(P)} 4\Delta^\ell \\ = |r(P)| \cdot 4\Delta^\ell &\leq (2\ell(2\ell-1)+2)\cdot 4\Delta^\ell \\ &\leq 16\ell^2\Delta^\ell. \end{aligned} \tag{3}$$

In the first inequality above we used the fact that each T_v contains at most $4\Delta^\ell$ nodes for $\Delta \geq 2$, and in the second-last inequality the fact that $r(P)$ contains the (free) endpoints of P plus at most 2ℓ entries for each one of the $2\ell-1$ matched nodes in P. We can conclude that the sum of the counters is $\sum_{v\in V}\sum_{i=0}^{2\ell} C_i[v] \leq 16\ell^2\Delta^\ell \cdot |M|$.

Let V_C be the nodes with counters set to at least C. Observe that $|V_C|$ cannot exceed the sum of all counters divided by C, since no counter exceeds that value. In the worst case each $w \in V_C$ hits a distinct path in $\mathcal{P}^{count}$. Therefore,

$$\begin{aligned} |\mathcal{P}^{count}| \leq |V_C| &\leq \frac{1}{C}\sum_{v\in V}\sum_{i=0}^{2\ell} C_i[v] \\ &\leq \frac{1}{C}\cdot 16\ell^2\Delta^\ell\,|M|. \end{aligned} \tag{4}$$

Altogether, we achieve

$$\begin{aligned}
&\frac{|OPT|}{|M|} \\
\overset{(1)}{\leq}\ & \frac{\ell+1}{\ell}\frac{|M|+|\mathcal{P}^{degree}|+|\mathcal{P}^{count}|}{|M|} \\
\overset{(2)+(4)}{\leq}\ & \frac{\ell+1}{\ell}\left(1+\frac{2\ell}{\Delta}+\frac{16\ell^2\Delta^\ell}{C}\right).
\end{aligned}$$

□

It remains to bound the running time of the algorithm.

Lemma 3.5. *The amortized running time per insertion is* $O(\ell^2 \cdot C + \ell^3 \cdot \Delta^\ell)$.

Proof. We analyze the cost of the different procedures, excluding the cost of the corresponding calls to subroutines.

Procedure *augment*() can be executed at most m times. The cost of each such execution on P_{aug} is asymptotically dominated by the total number $n(P_{aug})$ of nodes contained in trees T_v with $v \in r(P_{aug})$, that is $O(\ell^2 \cdot \Delta^\ell)$ by (3).

In procedure *insert*() lines 1-4 cost $O(\ell)$ per edge insertion. Each execution of the while loop (lines 5-11) costs $O(\ell)$. There are at most m such executions where $P_{aug} \neq$ NULL, and each such execution adds at most $O(\ell^2\Delta^\ell)$ entries to V_{exp} by the same argument as before. Hence lines 5-11 have a total cost of at most $O(\ell^3\Delta^\ell \cdot m)$.

It remains to consider the total cost of the procedure $expand(bc, i)$. Let $deg(v)$ denote the degree of node v in the final graph. Lines 3-11 cost $O(\ell)$, and are executed twice for each odd level i and for each newly inserted edge $\{a', b'\}$. Hence their total cost is $O(\ell^2 \cdot m)$.

Each execution of lines 12-15 costs $O(\ell \cdot deg(b))$. Let us charge this cost to b. Note that b cannot be charged more than C times for each even level i by the Counter Invariant. Indeed, each call to $expand(bc, i)$ for some c, excluding possibly the first time that b is added to some tree due to line 11, implies that edge bc was contained at level i in some destroyed tree T_w. The latter event in turn implies the increment of $C_{i-1}[b]$. Hence the total cost of these lines is $\sum_{i=1,3,\ldots,2\ell-1}\sum_{b\in V} O(C\ell \cdot deg(b)) = O(C\ell^2 \cdot m)$.

Each execution of lines 16-24 costs $O(\ell \cdot deg(c))$. Let us charge this cost to c. By the same argument as above, c is charged at most C times for each even level i. Hence the total cost of these lines is $\sum_{i=0,2,\ldots,2\ell}\sum_{c\in V} O(C\ell \cdot deg(b)) = O(C\ell^2 \cdot m)$. The claim follows. □

The proof of Theorem 1.1 in the bipartite case follows easily.

Proof of Theorem 1.1. (Bipartite Case) W.l.o.g. assume that $1/\varepsilon$ is integer and $\varepsilon \leq 1$. Let us choose $\ell = \frac{4}{\varepsilon}$, $\Delta = \frac{8\ell}{\varepsilon}$ and $C = \frac{64\ell^2\Delta^\ell}{\varepsilon}$. From Lemma 3.5 the amortized time per insertion is $O((1/\varepsilon)^{O(1/\varepsilon)})$. From Lemma 3.4, the approximation factor is at most $\frac{\ell+1}{\ell}(1+\frac{2\ell}{\Delta}+\frac{16\ell^2\Delta^\ell}{C}) \leq (1+\frac{\varepsilon}{4})(1+\frac{2\varepsilon}{4}) \leq 1+\varepsilon$. The claim follows. □

4 The Incremental Algorithm: General Graphs

In this section we deal with the case of general graphs. In Section 4.1 we describe our simple-paths covering algorithm. In Section 4.2 we sketch the changes of the bipartite-case algorithm and analysis that are needed to address general graphs.

4.1 Simple-Paths Covering

We say that a simple path P' (respectively, P) is a *valid suffix* (resp., *valid prefix*) of another simple path P (resp., P') if the concatenated path $P \circ P'$ is a (valid) simple path; the concatenated path $P \circ P'$ is valid and simple iff the two paths intersect at a single vertex: the first and last vertex along paths P' and P, resp. Let $\mathcal{U} = \mathcal{U}_s$ be a set of paths all ending at some arbitrary vertex s. We say that a path P' is *covered* by $\mathcal{U}$ if at least one path $P \in \mathcal{U}$ is a valid prefix of P'; for any integer $i \geq 1$, denote by $Cover_i(\mathcal{U})$ the set of length-i paths covered by $\mathcal{U}$. A subset $\mathcal{C} = \mathcal{C}_s$ of paths from $\mathcal{U}$ is called an *i-cover* of $\mathcal{U}$ if any length-i path covered by $\mathcal{U}$ is also covered by $\mathcal{C}$ as well, i.e., $Cover_i(\mathcal{U}) = Cover_i(\mathcal{C})$. We remark that in our application $\mathcal{U}$ will refer to a set of paths of a given graph, while P' is merely interpreted as any sequence of distinct nodes.

Fix two integers $\kappa, \kappa' \geq 1$, a vertex s, and any set $\mathcal{U}$ of length-κ (simple) paths ending at s. In what follows we present and analyze a simple algorithm, Algorithm *GreedyCover*, for efficiently computing a κ'-cover $\mathcal{C}$ of $\mathcal{U}$. The cover $\mathcal{C}$ computed by this algorithm will be referred to as the *greedy cover* (for $\mathcal{U}$). Although the greedy cover is not necessarily of minimum size, we will show that its size depends only on κ and κ' (and not on $|\mathcal{U}|$). It is a-priori unclear and perhaps counterintuitive that such a simple-paths cover exists for any path set $\mathcal{U}$ (even for $\kappa' = 1$), even regardless of the time needed for constructing it.

The order of paths in the input path set $\mathcal{U}$ determines the output path set $\mathcal{C}$; we thus assume that the paths of $\mathcal{U}$ are stored in some linked list, denoted by $\overrightarrow{\mathcal{U}}$, according to a predetermined order. Similarly, the output path set $\mathcal{C}$ is stored in some linked list, denoted by $\overrightarrow{\mathcal{C}}$; it is technically convenient to guarantee that the paths will be stored in $\overrightarrow{\mathcal{C}}$ according to their order in $\overrightarrow{\mathcal{U}}$. We shall henceforth refer to $\overrightarrow{\mathcal{U}}$ and $\overrightarrow{\mathcal{C}}$ as the input and output path *sequences* or *lists*, where $\overrightarrow{\mathcal{C}} = GreedyCover(\overrightarrow{\mathcal{U}}, \kappa')$.

Algorithm *GreedyCover* is recursive. The base of the recursion if $\kappa' = 1$, in which case the algorithm works as follows. Write the input path sequence $\overrightarrow{\mathcal{U}} = \overrightarrow{\mathcal{U}}_s$ as $(P_1, \ldots, P_u)$, with $u = |\mathcal{U}|$. Write $P_1 = (v_1, \ldots, v_{\kappa+1} = s)$. The algorithm scans $\overrightarrow{\mathcal{U}}$ once per each vertex of P_1 except $v_{\kappa+1}$: For each vertex v_i, $i = 1, \ldots, \kappa$, let $P(v_i)$ be the first path in $\overrightarrow{\mathcal{U}}$ starting with P_2 that does not go through v_i, setting $P(v_i) = \text{NULL}$ if none exists. The output path sequence $\overrightarrow{\mathcal{C}} = \overrightarrow{\mathcal{C}}_s$ is obtained by taking all non-NULL paths in $\{P_1, P(v_1), P(v_2), \ldots, P(v_\kappa)\}$ according to their original order in $\overrightarrow{\mathcal{U}}$, leaving a single occurrence of each path in $\overrightarrow{\mathcal{C}}$.

For $\kappa' > 1$ the algorithm proceeds as follows. Write the input path sequence $\overrightarrow{\mathcal{U}} = \overrightarrow{\mathcal{U}}_s$ as $(P_1, \ldots, P_u)$, with $u = |\mathcal{U}|$. The algorithm computes a κ'-cover for $\overrightarrow{\mathcal{U}}$ recursively, where $(\kappa' - i)$-covers are computed at the ith recursion level, for $i = 0, 1, \ldots, \kappa' - 1$. The recursion bottoms at 1-covers, which are computed using the already described algorithm for $\kappa' = 1$. Write $P_1 = (v_1, \ldots, v_{\kappa+1} = s)$. The algorithm scans $\overrightarrow{\mathcal{U}}$ once per each vertex of P_1 except $v_{\kappa+1}$: For each vertex v_i, $i = 1, \ldots, \kappa$, it first computes the path subsequence of $\overrightarrow{\mathcal{U}}$ that consists of all paths starting at P_2 that do not go through v_i, denoted by $\overrightarrow{\mathcal{U}(v_i)}$, and then invokes the algorithm recursively to compute a $(\kappa' - 1)$-cover for $\overrightarrow{\mathcal{U}(v_i)}$. The output path sequence $\overrightarrow{\mathcal{C}}$ is obtained as follows: First compute the path set that consists of P_1 as well as every path in any of the $(\kappa' - 1)$-covers computed recursively, and then place all those paths in $\overrightarrow{\mathcal{C}}$ according to their order in $\overrightarrow{\mathcal{U}}$, leaving a single occurrence of each path in $\overrightarrow{\mathcal{C}}$.

Lemma 4.1. *The running time of GreedyCover*$(\mathcal{U}, \kappa')$ *is* $O((\kappa + 1)^{\kappa'} \cdot |\mathcal{U}|)$.

Proof. We prove by induction on κ' that the runtime of the algorithm is bounded by $c((\kappa+1)^{\kappa'} \cdot |\mathcal{U}|)$, for a sufficiently large constant c. For $\kappa' = 1$ the running time is trivially $O(\kappa \cdot |\mathcal{U}|)$.

We next assume the correctness of the inductive statement for $\kappa' - 1$ and prove it for κ', with $\kappa' \geq 2$. By induction hypothesis, the runtime of recursively computing each of the $(\kappa' - 1)$-covers is bounded by $c((\kappa+1)^{\kappa'-1} \cdot |\mathcal{U}|)$. Since there are κ such $(\kappa' - 1)$-covers, the overall runtime of these recursive computations is bounded by

$$\begin{aligned} &\kappa(c((\kappa + 1)^{\kappa'-1})|\mathcal{U}|) \\ &= c((\kappa + 1)^{\kappa'}|\mathcal{U}|) - c((\kappa + 1)^{\kappa'-1}|\mathcal{U}|). \end{aligned}$$

The time needed for computing the κ subsequences $\overrightarrow{\mathcal{U}(v_1)}, \ldots, \overrightarrow{\mathcal{U}(v_\kappa)}$ of $\overrightarrow{\mathcal{U}}$ is naively bounded by $O(\kappa \cdot |\mathcal{U}|)$. Clearly, the time needed for computing $\overrightarrow{\mathcal{C}}$ given the $(\kappa' - 1)$-covers obtained by the recursive computations is linear in the sum of sizes of those covers, which is naively bounded by $\kappa \cdot |\mathcal{U}|$, disregarding the time needed for guaranteeing that each path will have a single occurrence in $\overrightarrow{\mathcal{C}}$. But the latter time is easily bounded by $O(k \cdot |\mathcal{U}|)$ as well. Since $\kappa' \geq 2$, it follows that the overall runtime of the algorithm is bounded by

$$\begin{aligned} c((\kappa + 1)^{\kappa'} \cdot |\mathcal{U}|) - c((\kappa + 1)^{\kappa'-1} \cdot |\mathcal{U}|) \\ +O(\kappa \cdot |\mathcal{U}|) \\ \leq c((\kappa + 1)^{\kappa'} \cdot |\mathcal{U}|) \end{aligned}$$

for a sufficiently large constant c. □

Lemma 4.2. *GreedyCover*$(\mathcal{U}, \kappa')$ *outputs a (feasible)* κ'*-cover* $\mathcal{C}$ *of* $\mathcal{U}$ *of size at most* $(\kappa + 1)^{\kappa'}$.

Proof. Let us first bound the size of $\mathcal{C}$. For $\kappa' = 1$, this size is trivially at most $\kappa + 1$. We next assume the correctness of the inductive statement for $\kappa' - 1$ and prove it for κ', with $\kappa' \geq 2$. By induction hypothesis, each of the $(\kappa' - 1)$-covers computed recursively is of size bounded by $(\kappa + 1)^{\kappa'-1}$. Since there are κ such $(\kappa' - 1)$-covers, their union contains at most $\kappa \cdot (\kappa + 1)^{\kappa'-1} \leq (\kappa + 1)^{\kappa'} - 1$ paths. The computed κ'-cover $\mathcal{C}$ contains the paths in this union as well as P_1, hence its size is bounded by $(\kappa + 1)^{\kappa'}$.

Consider next the correctness of the algorithm. Let us start with $\kappa' = 1$. Consider any length-1 path $P' = (s, t)$ covered by $\mathcal{U}$, and let $P \in \mathcal{U}$ be a valid prefix of P'. We argue that P' is covered by $\mathcal{C}$. If P_1 is a valid prefix for P', we are done. Otherwise P_1 must go through t. Let $i \in [\kappa]$ be such that $v_i = t$. Since $P \in \mathcal{U}$ is a valid prefix of P', $P(v_i) \neq$ NULL. By definition, $P(v_i)$ is a simple path ending at s that does not go through $v_i = t$, hence $P(v_i) \in \mathcal{C}$ is a valid prefix of P'.

We next assume the correctness of the inductive statement for $\kappa' - 1$ and prove it for κ', with $\kappa' \geq 2$. Consider any length-κ' path $P' = (s = u_1, u_2, \ldots, u_{\kappa'+1})$ covered by $\mathcal{U}$, and let $P \in \mathcal{U}$ be a valid prefix of P'. We argue that P' is covered by $\mathcal{C}$. Recalling that $P_1 \in \mathcal{C}$, the case that P_1 is a valid prefix of P' is immediate. We henceforth assume that P' goes through at least one vertex, denoted by v, among the first κ vertices $v_1, \ldots, v_\kappa$ of P_1; write v as both v_i and u_j, with $i \in [\kappa], j \in [2, \kappa' + 1]$. The fact that $P \in \mathcal{U}$ is a valid prefix of P' implies that P does not go through v_i, and therefore $P \in \overrightarrow{\mathcal{U}(v_i)}$, which means that P' is covered by $\mathcal{U}(v_i)$. Now consider the length-$(\kappa' - 1)$ path $\tilde{P}$ obtained from P' by removing vertex u_j from it, i.e., $\tilde{P} = (s = u_1, \ldots, u_{j-1}, u_{j+1}, \ldots, u_{\kappa'+1})$ if $j \leq \kappa'$ and $\tilde{P} = (s = u_1, \ldots, u_{\kappa'})$ if $j = \kappa' + 1$.[6] Since P is a valid prefix of P', it is also a valid prefix of $\tilde{P}$. By induction hypothesis and since $P \in \overrightarrow{\mathcal{U}(v_i)}$ is a valid prefix of $\tilde{P}$, it follows that $\tilde{P}$ is covered by the $(\kappa' - 1)$-cover computed recursively for $\overrightarrow{\mathcal{U}(v_i)}$, denoted by $\overrightarrow{\mathcal{C}_i}$; let Π be a path in $\overrightarrow{\mathcal{C}_i}$ that is a valid prefix of $\tilde{P}$. Since Π belongs to $\mathcal{U}(v_i)$, it does not go through $v_i = u_j$, hence Π is also a valid prefix of P'. Noting that $\Pi \in \mathcal{C}_i \subseteq \mathcal{C}$ concludes the proof. $\square$

The following observation, implied by the description of the algorithm, will be useful in the sequel.

Observation 4.3. *Let $\kappa, \kappa' \geq 1$, let $\overrightarrow{U} = \overrightarrow{U_s}$ be any sequence of κ-length paths all ending at an arbitrary vertex s, and let $\overrightarrow{C} = \overrightarrow{C_s} = GreedyCover(\overrightarrow{U}, \kappa')$. Then $GreedyCover(\overrightarrow{C}, \kappa') = \overrightarrow{C}$, and more generally:*

- *For any supersequence $\overrightarrow{C'} = \overrightarrow{C'_s}$ of $\overrightarrow{C}$ in which all elements of $\overrightarrow{C}$ appear at the start in $\overrightarrow{C'}$, $GreedyCover(\overrightarrow{C'}, \kappa')$ returns a supersequence of $\overrightarrow{C}$ in which all elements of $\overrightarrow{C}$ appear at the start.*
- *For any subsequence $\overrightarrow{C'} = \overrightarrow{C'_s}$ of $\overrightarrow{C}$, we have $GreedyCover(\overrightarrow{C'}, \kappa') = \overrightarrow{C'}$.*

4.2 Algorithm and Analysis for General Graphs

In this section we sketch how to update the algorithm and analysis to address the case of general graphs. The details will appear in the full version of the paper.

As mentioned in the introduction, we need to allow nodes to appear at the same level in multiple trees if we want to detect augmenting paths despite the presence of blossoms. However, we critically need that the number of copies of a given node is bounded by some function ρ of ℓ only. This way, by scaling the constants Δ and C properly (by a

[6] The paths are not restricted to an underlying graph, so any sequence of vertices without repetitions forms a simple path.

factor depending on ρ), we can still have a $1+\varepsilon$ approximation in constant amortized time by essentially the same analysis as in the bipartite case.

Having at hand our simple-paths covering notion and algorithm, the solution is relatively straightforward modulo a number of small technical details. Intuitively, consider an augmenting path $P_{aug} = (v_0, v_1, \ldots, v_{2q+1})$, $2q+1 \leq 2\ell+1$, whose nodes are below the degree and counter threshold. We would like to guarantee that P_{aug} or some other augmenting path intersecting with it is discovered by the algorithm. Consider node v_i, $1 \leq i \leq 2q$, and let $P = (v_0, v_1, \ldots, v_i)$ and $P' = (v_i, v_{i+1}, \ldots, v_{2q+1})$ be the corresponding prefix and suffix of P_{aug}, resp. In particular, P and P' have length $\kappa = i$ and $\kappa' = 2q+1-i$, resp. For our goals it is sufficient to guarantee that v_i belongs to some tree T_w such that $T_w(v_i) \circ P'$ is a valid augmenting path. In turn, this property is guaranteed if we ensure the following. Let $P_i(v_i)$ be the collection of (simple alternating) paths of length κ that start at the root w of some tree T_w and end at a copy of v_i. It is sufficient to guarantee that $P_i(v_i)$ is a κ'-cover with respect to a proper set of paths $\mathcal{U}$ of length κ that includes P. In particular, this implies that there exists some $P'' \in P_i(v_i)$ such that $P'' \circ P'$ is a valid augmenting path.

It is therefore sufficient to modify the Tree Invariant in order to incorporate the above notion of κ'-covers, and modify the algorithm so that the new invariant is maintained. Using our GreedyCover algorithm to update the paths, we can ensure that the number of paths of type $P_i(v_i)$ (hence the number of copies of each node at a given level i) never exceeds a constant $\rho = \ell^{O(\ell)}$. In turn this implies an increase of the running time by a constant factor depending on ρ due to maintaining the mentioned κ'-covers dinamically.

The proof of Theorem 1.1 for general graphs follows, modulo technical details.

References

[1] Amir Abboud and Søren Dahlgaard. Popular conjectures as a barrier for dynamic planar graph algorithms. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pages 477–486, 2016.

[2] Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 434–443, 2014.

[3] Moab Arar, Shiri Chechik, Sarel Cohen, Cliff Stein, and David Wajc. Dynamic matching: Reducing integral algorithms to approximately-maximal fractional algorithms. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, pages 7:1–7:16, 2018.

[4] Sepehr Assadi, MohammadHossein Bateni, Aaron Bernstein, Vahab S. Mirrokni, and Cliff Stein. Coresets meet EDCS: algorithms for matching and vertex cover on massive graphs. *CoRR*, abs/1711.03076, 2017.

[5] Surender Baswana, Manoj Gupta, and Sandeep Sen. Fully dynamic maximal matching in o(log n) update time. *SIAM J. Comput.*, 44(1):88–113, 2015.

[6] Aaron Bernstein, Jacob Holm, and Eva Rotenberg. Online bipartite matching with amortized replacements. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 947–959, 2018.

[7] Aaron Bernstein and Cliff Stein. Fully dynamic matching in bipartite graphs. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part I*, pages 167–179, 2015.

[8] Aaron Bernstein and Cliff Stein. Faster fully dynamic matchings with small approximation ratios. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 692–711, 2016.

[9] Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. Deterministic fully dynamic data structures for vertex cover and matching. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 785–804, 2015.

[10] Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. New deterministic approximation algorithms for fully dynamic matching. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 398–411, 2016.

[11] Bartlomiej Bosek, Dariusz Leniowski, Piotr Sankowski, and Anna Zych. Online bipartite matching in offline time. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 384–393, 2014.

[12] Moses Charikar and Shay Solomon. Fully dynamic almost-maximal matching: Breaking the polynomial worst-case time barrier. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, pages 33:1–33:14, 2018.

[13] Søren Dahlgaard. On the hardness of partially dynamic graph problems and connections to diameter. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, pages 48:1–48:14, 2016.

[14] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[15] Ashish Goel, Michael Kapralov, and Sanjeev Khanna. On the communication and streaming complexity of maximum bipartite matching. In *Proceedings of the 23rd*

Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pages 468–485, 2012.

[16] Manoj Gupta. Maintaining approximate maximum matching in an incremental bipartite graph in polylogarithmic update time. In *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014, December 15-17, 2014, New Delhi, India*, pages 227–239, 2014.

[17] Manoj Gupta and Shahbaz Khan. Simple dynamic algorithms for maximal independent set and other problems. *CoRR*, abs/1804.01823, 2018.

[18] Manoj Gupta and Richard Peng. Fully dynamic (1+epsilon)approximate matchings. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*, pages 548–557, 2013.

[19] Guru Prashanth Guruganesh and Sahil Singla. Online matroid intersection: Beating half for random arrival. In *Integer Programming and Combinatorial Optimization - 19th International Conference, IPCO 2017, Waterloo, ON, Canada, June 26-28, 2017, Proceedings*, pages 241–253, 2017.

[20] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 21–30, 2015.

[21] 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.

[22] Zoran Ivkovic and Errol L. Lloyd. Fully dynamic maintenance of vertex cover. In *Graph-Theoretic Concepts in Computer Science, 19th International Workshop, WG '93, Utrecht, The Netherlands, June 16-18, 1993, Proceedings*, pages 99–111, 1993.

[23] Michael Kapralov. Better bounds for matchings in the streaming model. In *Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1679–1697, 2013.

[24] Richard M. Karp, Umesh V. Vazirani, and Vijay V. Vazirani. An optimal algorithm for on-line bipartite matching. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*, pages 352–358, 1990.

[25] Christian Konrad, Frédéric Magniez, and Claire Mathieu. Maximum matching in semi-streaming with few passes. In *Proceedings of the 16th Workshop on Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM)*, pages 231–242, 2012.

[26] Tsvi Kopelowitz, Seth Pettie, and Ely Porat. Higher lower bounds from the 3sum conjecture. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 1272–1287, 2016.

[27] Aleksander Madry. Navigating central path with electrical flows: From flows to matchings, and back. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*, pages 253–262, 2013.

[28] Silvio Micali and Vijay V. Vazirani. An $\emptyset(\sqrt{|V|}|E|)$ algorithm for finding maximum matching in general graphs. In *Proceedings of the 21st Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 17–27, 1980.

[29] Marcin Mucha and Piotr Sankowski. Maximum matchings via Gaussian elimination. In *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 248–255, 2004.

[30] Ofer Neiman and Shay Solomon. Simple deterministic algorithms for fully dynamic maximal matching. *ACM Trans. Algorithms*, 12(1):7:1–7:15, 2016.

[31] Krzysztof Onak and Ronitt Rubinfeld. Maintaining a large matching and a small vertex cover. In *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, pages 457–464, 2010.

[32] David Peleg and Shay Solomon. Dynamic (1 + epsilon)-approximate matchings: A density-sensitive approach. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 712–729, 2016.

[33] Piotr Sankowski. Faster dynamic matchings and vertex connectivity. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*, pages 118–126, 2007.

[34] Shay Solomon. Fully dynamic maximal matching in constant update time. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pages 325–334, 2016.

[35] Shay Solomon. Dynamic approximate matchings with an optimal recourse bound. *CoRR*, abs/1803.05825, 2018.

A Proof of the Tree Invariant Lemma 3.2

Let us show that the Tree Invariant is maintained by *insert*().

Lemma A.1. *If procedure expand(bc, i) ends with $P_{aug} = $ NULL and with c belonging to some T_v, then the subtree of T_v rooted at c satisfies the Tree Invariant constraints and is maximal w.r.t. those constraints.*

Proof. Let us assume that c finally belongs to some tree T_v and $P_{aug} = \text{NULL}$, otherwise there is nothing to show. We remark that *expand*() might be called on some edge bc that does not belong to any T_v initially, and might fail to insert c in any such tree. We prove the claim by induction on decreasing values of i. The claim trivially holds whenever $i = 2\ell + 1$. Indeed in that case it cannot happen that $P_{aug} = \text{NULL}$ and at the same time c is added to some tree T_v. Similarly, the claim holds for $i = 2\ell$ since in that case the subtree rooted at c contains c only.

Suppose next that the claim is true up to level $i + 1$ and consider level i. For odd i, bc must be some newly inserted edge. In that case *expand*(), if possible, adds a path of type bcd to some tree T_v, and then calls $expand(cd, i + 1)$. The claim follows by induction. For even i, *expand*() adds bc at level i in some tree T_v, if needed, then adds a maximal set of paths of type cde to T_v, and for each such path it calls $expand(de, i + 2)$. The claim follows by inductive hypothesis. □

Proof of Lemma 3.2. We prove the claim by induction on the number of insertions. The claim is trivially true before the first insertion. Next assume the claim holds for the first $j - 1$ insertions, and let $\{a', b'\}$ be the j-th inserted edge. Observe that, whenever we add an edge to any tree T_v, this is via a call to *expand*(). By definition, the latter procedure augments trees without violating the constraints of the Tree Invariant until it terminates or finds an augmenting path involving T_v (that leads to the destruction of T_v). Therefore, it is sufficient to show that the trees T_v are maximal w.r.t. the Tree Invariant constraints at the end of the execution of $insert(\{a', b'\})$.

Assume by contradiction that this is not the case, in particular there exists a non-maximal tree T_v at the end of the procedure. Note that this implies that v is free at that time.

We distinguish two cases. Suppose first that vv is inserted in $V_{exp}[0]$ at least once, and let t be the last iteration when vv is extracted from $V_{exp}[0]$. Upon execution of $expand(vv, 0)$ at iteration t, we cannot find an augmenting path involving v. Indeed otherwise the following call to *augment*() would match v. Thus, by Lemma A.1, T_v is maximal, a contradiction.

We can therefore assume that the non-maximal tree T_v at the end of the while loop involves a pair vv which never appears in $V_{exp}[0]$. Let T_v^{start} and T_v^{end} be the status of T_v at the beginning of the first iteration of the while loop and at the end of the procedure, respectively. Since T_v is never destroyed by *augment*(), it can be updated only by *expand*(), than can only add edges and nodes to T_v. Thus, for any intermediate status T'_v of T_v, one has:

$$T_v^{start} \subseteq T'_v \subseteq T_v^{end}. \tag{5}$$

If no augmenting path is ever discovered, then by inductive hypothesis the only possibility for a tree T_w to be non-maximal is that T_w should include $a'b'$ at some odd level i for the newly inserted edge $\{a', b'\}$. However the calls to $expand(a'b', i)$ guarantee that $a'b'$ is inserted in at most one such tree T_w if possible, and in that case the subtree of T_w rooted at b' is later augmented in a maximal way by Lemma A.1. Hence there cannot exist a non-maximal tree T_v, a contradiction.

We can therefore assume that some first augmenting path P_{aug} is discovered. This path clearly contains the edge $a'b'$ or $b'a'$. We can therefore conclude that T_v^{start} does not contain nodes a' nor b', since otherwise it would be destroyed by the first call to $augment()$. This in turn implies that T_v^{start} is maximal at the beginning of the first iteration of the while loop, since the unmatched edges $a'b'$ and $b'a'$ cannot be added to it.

By assumption, T_v^{end} is not maximal. In particular, there must exist a tuple (a, b, c, i), with abc not contained in T_v^{end} and $i \geq 2$ even, such that: (i) $C_i[a], C_{i+1}[b], C_{i+2}[c] < C$ and $deg_{T_v^{end}}(a) < \Delta$, (ii) $a \in T_v^{end}$ at level i and $b, c \notin T_v^{end}(a)$, (iii) $\{a, b\} \notin M$ and $\{b, c\} \in M$, (iv) bc is not contained at level $i + 2$ in some tree T_w at the end of the procedure.

Let T_v^t be the status of T_v at any discrete time slot t between the beginning of the first while loop and the end of the procedure. By the previous discussion there must exist one such time slot t so that T_v^t violates precisely one of the analogues of the conditions (i)-(iv), while $T_v^{t'}$ satisfies all of them for any $t' > t$. We next distinguish 4 subcases, depending on the condition (x) that is violated by T_v^t.

Case (x)=(i). This case cannot occur since counters can only increase over time, and the same holds for $deg_{T_v}(a)$ by (5).

Case (x)=(ii). Then $a \in T_v^{t+1}$. This involves a call of type $expand(wa, i)$, that must add abc to T_v at some later point since the conditions (i), (iii) and (iv) are satisfied at any time $t' \geq t + 1$ by definition. This contradicts the assumptions.

Case (x)=(iii). This means that an execution of $augment()$ at time $t + 1$ either (1) turns edge $\{a, b\}$ from matched to unmatched, or (2) turns edge $\{b, c\}$ from unmatched to matched. However (1) cannot occur since it would imply the destruction of T_v (given that $a \in T_v$ at that time). Assuming (2), by construction bc is added to $V_{exp}[i + 2]$ and hence $insert()$ executes $expand(bc, i + 2)$ at some later time. At that point the tuple (a, b, c, i) satisfies all the conditions (i)-(iv), which implies that $expand(bc, i + 2)$ must add bc to a maximal number of trees T_w at level i. This contradicts the assumptions.

Case (x)=(iv). This implies that bc is removed at time $t+1$ from some tree T_w where it was contained at level $i + 2$. This however implies that at some later point $insert()$ executes $expand(bc, i+2)$. At that point all conditions (i)-(iv) hold, hence $expand()$ must add bc to a maximal number of trees T_w at level i. This contradicts the assumptions. □