Dynamic Matching Algorithms Under Vertex Updates

Hung Le
University of Massachusetts, Amherst, MA, USA

Lazar Milenković
Tel Aviv University, Israel

Shay Solomon
Tel Aviv University, Israel

Virginia Vassilevska Williams
MIT, Cambridge, MA, USA

Abstract

Dynamic graph matching algorithms have been extensively studied, but mostly under *edge updates*. This paper concerns dynamic matching algorithms under vertex updates, where in each update step a single vertex is either inserted or deleted along with its incident edges.

A basic setting arising in *online algorithms* and studied by Bosek et al. [FOCS'14] and Bernstein et al. [SODA'18] is that of dynamic approximate maximum cardinality matching (MCM) in bipartite graphs in which one side is fixed and vertices on the other side either arrive or depart via vertex updates. In the `BASIC-incremental` setting, vertices only arrive, while in the `BASIC-decremental` setting vertices only depart. When vertices can both arrive and depart, we have the `BASIC-dynamic` setting. In this paper we also consider the setting in which both sides of the bipartite graph are dynamic. We call this the `MEDIUM-dynamic` setting, and `MEDIUM-decremental` is the restriction when vertices can only depart. The `GENERAL-dynamic` setting is when the graph is not necessarily bipartite and the vertices can both depart and arrive.

Denote by K the total number of edges inserted and deleted to and from the graph throughout the entire update sequence. A well-studied measure, the *recourse* of a dynamic matching algorithm is the number of changes made to the matching per step. We largely focus on Maximal Matching (MM) which is a 2-approximation to the MCM. Our main results are as follows.

- In the `BASIC-dynamic` setting, there is a straightforward algorithm for maintaining a MM, with a total runtime of $O(K)$ and constant worst-case recourse. In fact, this algorithm never removes an edge from the matching; we refer to such an algorithm as *irrevocable*.
- For the `MEDIUM-dynamic` setting we give a strong conditional lower bound that even holds in the `MEDIUM-decremental` setting: if for any fixed $\eta > 0$, there is an irrevocable decremental MM algorithm with a total runtime of $O(K \cdot n^{1-\eta})$, this would refute the OMv conjecture; a similar (but weaker) hardness result can be achieved via a reduction from the Triangle Detection conjecture.
- Next, we consider the `GENERAL-dynamic` setting, and design an MM algorithm with a total runtime of $O(K)$ and constant worst-case recourse. We achieve this result via a *1-revocable* algorithm, which may remove just one edge per update step. As argued above, an irrevocable algorithm with such a runtime is not likely to exist.
- Finally, back to the `BASIC-dynamic` setting, we present an algorithm with a total runtime of $O(K)$, which provides an $(\frac{e}{e-1})$-approximation to the MCM.

To this end, we build on the classic "ranking" *online algorithm* by Karp et al. [STOC'90].

Beyond the results, our work draws connections between the areas of dynamic graph algorithms and online algorithms, and it proposes several open questions that seem to be overlooked thus far.

2012 ACM Subject Classification Theory of computation → Dynamic graph algorithms

Keywords and phrases maximal matching, approximate matching, dynamic algorithm, vertex updates

Digital Object Identifier 10.4230/LIPIcs.ITCS.2022.96

Funding *Hung Le*: Supported by National Science Foundation under Grant No. CCF-2121952.
Lazar Milenković: Partially supported by the Israel Science Foundation (ISF) grant No.1991/1, and by a grant from the United States-Israel Binational Science Foundation (BSF), Israel, and the United States National Science Foundation (NSF).
Shay Solomon: Supported by the Israel Science Foundation (ISF) grant No.1991/1, and by a grant from the United States-Israel Binational Science Foundation (BSF), Israel, and the United States National Science Foundation (NSF).
Virginia Vassilevska Williams: Supported in part by NSF CAREER Award 1651838, NSF Grants CCF-1909429 and CCF- 2129139, BSF grants 2016365 and 2020356, a Google Research Fellowship and a Sloan Research Fellowship.

1 Introduction

Dynamic matching algorithms have been intensively studied in recent years. The great majority of previous work considers graphs that change via *edge updates*, where in each update step a single edge is either inserted to or deleted from the graph. In dynamic graphs that are subject to *vertex updates*, where each update step consists of an insertion or a deletion of a vertex (with all its incident edges), the literature is much more sparse and, to the best of our knowledge, essentially all of it is restricted to incremental (insertion only) or decremental (deletion only) updates. This work studies matching algorithms in *fully dynamic graphs under vertex updates*.

1.1 Background

In the area of dynamic graph algorithms, one can try to optimize the *amortized* or *worst-case* update time of an algorithm, where both measures are defined with respect to a worst-case sequence of updates, and the worst-case (resp., amortized) update time designates the maximum (resp., average) update time spent by the algorithm over all update steps.

In graphs subject to vertex updates, which is our focus, when it comes to the update time, one should care about *amortized* bounds. Indeed, the update time following a vertex update must be at least linear in the degree of that vertex. Due to possibly significant differences in the degrees of updated vertices, the worst-case update time is rather meaningless. On the other hand, the amortized update time is a *normalized* notion of the total runtime of the algorithm, where we divide the total runtime by the number of *edges* ever inserted and deleted to and from graph; dividing by the number of vertex updates is far less informative. Thus, in the background survey to follow (also in graphs subject to edge updates), we usually omit the distinction between amortized and worst-case bounds. We start with dynamic graphs subject to edge updates (Section 1.1.1) and later move on to vertex updates (Section 1.1.2).

1.1.1 Edge updates

We do not cover in this survey the entire literature on matching algorithms under edge updates. We shall restrict attention to the state-of-the-art results on matchings whose size approximates that of the maximum cardinality matching (MCM) to within a factor of 2, which is the focus of this work; this includes maximal matching (MM), which provides a 2-approximate MCM.

Maximal matching (MM)

Deterministically, the only improvements to the naive MM algorithm with update time $O(n)$ are for sufficiently sparse graphs: update time $O((n+m)^{\sqrt{2}/2})$ was achieved in [24], which was improved in [27] to $O(\sqrt{m})$, where n (resp., m) denote the fixed (resp., dynamic) number of vertices (resp., edges) in the graph, respectively.

Better deterministic algorithms are known for bounded *arboricity* graphs. The arboricity of an m-edge graph G, denoted by $\alpha = \alpha(G)$, is the minimum number of forests into which it can be decomposed; this parameter ranges from 1 to $\sqrt{m}$ and is a common measure of "uniform sparsity". The current state-of-the art is update time $O(\alpha + \sqrt{\alpha \log n})$, due to [20].

Allowing randomization against a (non-adaptive) *oblivious adversary*, an update time of $O(\log n)$ was achieved [2], which was subsequently improved to $O(1)$ [31]. We remark that an oblivious adversary is not allowed to learn anything about the random bits used by the algorithm, which can alternatively be viewed as if the entire sequence of updates is fixed by the adversary in advance, i.e., prior to any random choices made by the algorithm.

Can one improve the $\sqrt{m}$ deterministic update time of [27], even using randomization against an adaptive adversary? No lower bound whatsoever is known.

▶ **Question 1.1.** *Is there a deterministic algorithm, or a randomized one against an adaptive adversary, for maintaining an MM in general graphs with a sub-polynomial update time?*

Better-than-2 approximate MCM

A $(1+\epsilon)$ (resp., $(3/2+\epsilon)$)-approximate MCM can be maintained with update time $O(\sqrt{m}/\epsilon^2)$ [19] (resp., $O(m^{1/4}\epsilon^{-2.5})$) [6, 7], for any $0 < \epsilon < 1/2$. These results were generalized for bounded arboricity graphs: A $(1+\epsilon)$ (resp., $(3/2+\epsilon)$)-approximate MCM can be maintained with update time $O(\alpha/\epsilon^2)$ [28] (resp., $O(\alpha^{1/4}\epsilon^{-2.5} + \epsilon^{-6})$ [17]), where α is the arboricity bound, which, as mentioned, ranges from 1 to $\sqrt{m}$.

A randomized algorithm against an oblivious adversary for maintaining a $(2-\Omega(\epsilon))$-approximate MCM with update time $O(\Delta^{\epsilon}+\text{polylog}(n))$ was given in [3], where ϵ is a constant and the Ω-notation in the approximation bound hides a tiny constant. For bipartite graphs, similar results can be achieved without making the oblivious adversary assumption, either via a randomized algorithm against an adaptive adversary [8, 33] or via a deterministic algorithm [8, 9]. We also mention a result for graphs of bounded *neighborhood independence*[1] β: There is a randomized algorithm against an adaptive adversary for maintaining a $(1+\epsilon)$-approximate MCM with update time $O(\frac{\beta}{\epsilon^3}\log\frac{1}{\epsilon})$ [26].

For any fixed $\epsilon > 0$, a $(1+\epsilon)$-MCM can be maintained in constant (resp., polylog(n)) update time in incremental general (resp., decremental bipartite) graphs [16] (resp., [9]).

The following is a major open problem.

▶ **Question 1.2.** *Is there any algorithm for maintaining a better-than-2 approximate MCM in general graphs with a sub-polynomial update time? Further, is it possible to get a significantly better than 2 (say, 1.99) approximation with a small polynomial update time?*

[1] The *neighborhood independence (number)* of a graph G, denoted by $\beta = \beta(G)$, is the size of the largest independent set in the neighborhood of any vertex.

1.1.2 Vertex updates

For matching algorithms under vertex updates, which is the focus of this work, the literature is much more sparse. In FOCS'14, Bosek et al. [11] considered bipartite graphs in which one side is fixed (n fixed "servers") and vertices ("clients") on the other side either arrive or depart in n vertex updates; we will refer to these settings as `BASIC-incremental` and `BASIC-decremental`. They optimized two measures: The total runtime of the algorithm (or equivalently the amortized update time) and the *recourse* bound, which measures how many changes to the matching are done per step. They designed an exact (respectively, $(1+\epsilon)$-approximate) MCM algorithm with a total runtime of $O(m\sqrt{n})$ (resp., $O(m\epsilon^{-1})$), where n and m are the number of vertices and edges in the graph, matching the performance of the celebrated Hopcroft-Karp [22] algorithm for a one-time static computation. Their exact and approximate algorithms admit recourse bounds of $O(\sqrt{n})$ and $O(\epsilon^{-1})$, respectively. A different exact MCM algorithm with an amortized recourse bound of $O(\log^2 n)$ was given in SODA'18 by Bernstein et al. [4]. See also [18, 14, 12, 13, 5, 32], and the references therein.

The recourse bound

The recourse bound is a basic measure of quality, and has been subject to growing attention in recent years [18, 14, 12, 13, 4, 5, 30, 29]. In some applications such as job scheduling, web hosting, streaming content delivery and data storage, and in situations where the matched edges are hardwired in a physical sense, a replacement of a matched edge by another one may be prohibitively expensive, possibly much more than the time needed to compute these replacements. Moreover, a low recourse bound is important when the matching algorithm is used as a black-box subroutine inside a larger data structure or algorithm (see, e.g., [7, 1]).

The worst-case (resp., amortized) recourse bound of an algorithm measures the maximum (resp., average) number of changes to the *output* matching per update step, over the entire sequence of updates. The recourse bounds in the algorithms of [11] and [4], as well as in other previous works [18, 14, 12, 13, 5] are *amortized*. Under vertex updates, as mentioned, the worst-case update time is not too informative, as it is problematic to compare update times following updates of vertices with significantly different degrees; the same is not true of course for edge updates. However, the recourse bound is not necessarily affected by the degrees of the updated vertices, and so optimizing the *worst-case* recourse bound is not less natural or important than optimizing the amortized recourse bound.

It was shown in ITCS'21 [30] that any dynamic algorithm for maintaining a γ-approximate MCM with update time T, for any $\gamma \geq 1$, T and $\epsilon > 0$, can be transformed into an algorithm for maintaining a $(\gamma(1+\epsilon))$-approximate MCM with update time $T + O(1/\epsilon)$ and a worst-case recourse bound of $O(1/\epsilon)$. This transformation applies to both edge and vertex updates. Although this transformation only increases the approximation guarantee by a factor of $1+\epsilon$, the transformed algorithm does not preserve the qualitative properties of the matching maintained by the original algorithm; in particular, an MM algorithm will be transformed into an algorithm that maintains a $(2+\epsilon)$-approximate MCM, which is very different of course than an MM.

1.2 Our contribution

As mentioned, our work studies matching algorithms in graphs that change dynamically under vertex updates, whereas most previous work in the area considers edge updates. In the area of online algorithms, on the other hand, most of the literature revolves around vertex updates. In both areas, vertex updates are easier to deal with than edge updates (further

details on that are given in Section 1.2.1). We stress that the objectives in the areas of online algorithms and dynamic graph algorithms are inherently different. In the former the objective is to optimize the *competitive ratio* of an online algorithm (the ratio of the approximation guarantee provided by the online algorithm to that of the best possible offline algorithm), each decision of the algorithm is *irrevocable*, and no runtime measure is considered. In the latter the main objective is to optimize the *update time* of the algorithm, and while the decisions are allowed to be "revoked" as many times as necessary, it is important nonetheless to optimize the recourse bound.

The previous works on dynamic matching algorithms under vertex updates were inspired by the area of online algorithms; e.g., the paper of Bosek et al. [11] is titled "online matching in offline time". In particular, as with most papers in the area of online algorithms, all previous work on dynamic matching algorithms under vertex updates considers either incremental or decremental updates. **Our paper is the first to study dynamic matching algorithms under *fully dynamic* vertex updates.** We denote by K the total number of edges inserted and deleted to and from the graph throughout the sequence of vertex updates, and note that a total runtime of $K \cdot t$, for any t, translates into an amortized update time of t.

We first observe that in bipartite graphs where one side is fixed, i.e., consists of n fixed vertices that we may view as "servers", and the other side is *fully dynamic*, i.e., vertices on that side that we may view as "clients" arrive and depart via vertex updates, hereafter the `BASIC-dynamic` setting, there is a trivial MM algorithm with a total runtime of $O(K)$ and constant worst-case recourse. In fact, this algorithm resembles an online algorithm in that it never removes an edge from the matching; we refer to such an algorithm as *irrevocable*. Throughout the paper, we shall distinguish between two types of edges deleted from the matching: Those due to the algorithm, which will be referred to as *removals* (only from the matching), and those due to the adversary (the entity performing the updates), which will be referred to as *deletions* (they get deleted from the graph, but then also from the matching). Further details are given in Section 2.

We then consider bipartite graphs in which both sides are fully dynamic, hereafter the `MEDIUM-dynamic` setting. For this setting we prove that, based on popular conjectures, no irrevocable algorithm with a total runtime of $O(K)$ may exist.

▶ **Theorem 1.** *An irrevocable MM algorithm with a total runtime of $O(K \cdot n^{1-\eta})$, for* any *fixed $\eta > 0$, would refute the the online matrix-vector multiplication (OMv) conjecture. Moreover, a runtime of $O(K \cdot m^{\eta})$, for* some *fixed $\eta > 0$, would refute the Triangle Detection conjecture.*

We also extend the hardness result provided by Theorem 1 to the decremental setting, called `MEDIUM-decremental`, where the update sequence only contains vertex deletions. In this setting, we can fix the graph and its initial maximal matching, and the dynamic algorithm must be able to maintain the maximal matching under vertex deletions. Alternatively, our results apply to an "almost-decremental" sequence of updates, which starts with a batch of insertions and proceeds with a batch of deletions. The decremental setting is more restricted than the fully dynamic setting, which makes our lower bound stronger.

Next, we consider general graphs under vertex updates, hereafter the `GENERAL-dynamic` setting. We say that an algorithm is *i-revocable*, for an integer parameter $i \geq 0$, if it may remove at most i edges per update step in the worst-case. (An irrevocable algorithm is 0-revocable.)

▶ **Theorem 2.** *For any sequence of updates in the `GENERAL-dynamic` setting, one can maintain an MM via a deterministic 1-revocable algorithm with a total runtime of $O(K)$.*

Any i-revocable MM algorithm may add at most $2i + 1$ edges to the matching per step, and so the algorithm provided by Theorem 2 has a constant worst-case recourse. Theorem 2 and Theorem 1 establish a strong separation between 1-revocable and irrevocable MM algorithms. Recall that in the edge update setting, the fastest deterministic update time is $\sqrt{m}$; see Question 1.1.

Finally, back to the `BASIC-dynamic` setting, we prove the following theorem in Section 5.

▶ **Theorem 3.** *For any sequence of updates in the `BASIC-dynamic` setting, one can maintain a $(\frac{e}{e-1})$-approximation (in expectation) to the MCM with a total runtime of $O(K)$.*

To achieve this result, we build on the classic "ranking" *online algorithm* by Karp et al. [25]. More specifically, we demonstrate how to efficiently maintain a matching corresponding to the ranking algorithm under vertex updates. Given any fixed ranking over the servers, our algorithm performs *deterministically* within total runtime $O(K)$, but the approximation factor may approach 2. To get an $(\frac{e}{e-1})$-approximation (in expectation), we use a random ranking over the servers, which is withheld from the adversary. Thus our algorithm is randomized and it works against an oblivious adversary. Recall that in the edge update setting, the fastest update time (including algorithms against an oblivious adversary) is polynomial in n; see Question 1.2.

1.2.1 Conceptual highlights

Beyond the results reported above, we next discuss the conceptual contribution of this work.

A more refined measure of recourse

We have already discussed the importance of the recourse bound. The number of changes to the matching is comprised of the number of edge removals and edge insertions. Removing an edge from the matching is arguably much more problematic or expensive than adding an edge to the matching, when it comes to real-life applications. Recall that an i-revocable algorithm may remove at most i edges from the matching per update in the worst-case; we identified this parameter, which we may refer to as the *revocable parameter*, as a pivotal measure of MM algorithms: While there is a 1-revocable MM algorithm with constant amortized update time (Theorem 2), based on popular conjectures, any irrevocable algorithm must incur a polynomial update time. For MM algorithms, this parameter provides a more refined measure than the standard recourse bound, since any i-revocable MM algorithm has a recourse of at most $3i + 1$; in particular, both irrevocable and 1-revocable MM algorithms have constant recourse.

It would be interesting to explore this measure further. For example, what is the relationship between irrevocable MM algorithms and maximum matching algorithms with no bound on the revocable parameter? Both problems admit high conditional lower bounds, but can we reduce one of the problems to the other? What about a $(2 + \epsilon)$-approximation algorithm that does not maintain an MM: Is there an irrevocable $(2 + \epsilon)$-approximate MCM algorithm with constant amortized update time? The hardness result of Theorem 1 exploits the maximality of the matching but not the approximation guarantee. We note that the $(\frac{e}{e-1})$-approximation algorithm of Theorem 3 may incur a large revocable parameter. Is it possible to achieve a similar approximation and update time via a 1-revocable, or even an irrevocable, algorithm?

Better-than-2 approximate MCM

As mentioned in Section 1.1.1, whether or not one can maintain a better-than-2 approximate MCM in sub-polynomial update time under edge updates is a major open problem (see Question 1.2). On the other hand, Theorem 3 shows that one can maintain a $(\frac{e}{e-1})$-approximate MCM in constant update time under vertex updates. To achieve this result, we maintained a matching corresponding to the ranking *online algorithm* of [25]. This draws a natural connection between the areas of online algorithms and dynamic graph algorithms. We anticipate that the area of dynamic graph algorithms could benefit significantly by borrowing more from the area of online algorithms, and it could also contribute to it of course.

Recall that the result of Theorem 3 applies to the `BASIC-dynamic` setting. It is unclear if our approach can be extended to the `MEDIUM-dynamic` setting, let alone to the `GENERAL-dynamic` setting; each of these settings adds another level of difficulty. Note also that in the `GENERAL-dynamic` setting, which concerns general rather than bipartite graphs, the approximation factor of $1-1/e$ does no longer apply. However, a generalization of the ranking algorithm [23] gives an approximation of roughly 1.919 (i.e., $\frac{1}{0.5211}$); one can hope to get this approximation in the `GENERAL-dynamic` setting. Achieving a better-than-2 approximate-MCM in these settings is not only an intriguing question on its own right, but could also be a stepping stone towards resolving Question 1.2, first in bipartite graphs and ultimately in general graphs.

2 An irrevocable MM in the `BASIC-dynamic` setting

In this section, we describe a simple irrevocable maximal matching algorithm for the `BASIC-dynamic` setting, which works in total runtime of $O(K)$. Recall that we use K to denote the total number of edges inserted and deleted to and from the graph throughout the entire update sequence.

In what follows, we denote by $\deg_i(v)$ and $N_i(v)$ the degree and set of neighbors of a vertex v in the graph after update step i, respectively; when the update step i is clear from the context, we omit the subscript. In the `BASIC-dynamic` setting, vertices on the fixed side S of the bipartition are often called servers while vertices on the dynamically changing side C are called clients.

Upon insertion of a client u, we scan its neighborhood and look for a free server in $N(u)$ to be matched with u. If there is such a server, denoted by v (v is chosen arbitrarily if there are multiple options), we add edge (u, v) to the maintained matching $\mathcal{M}$; otherwise, the algorithm "notifies" all the servers in $N(u)$ that u is free: For each server x in S , we maintain a doubly linked list $F(x)$ of free neighbors, and to notify the neighboring servers that u is free, the algorithm appends u to their linked lists. Whenever the algorithm appends some client u to the list $F(x)$ of a neighbor $x \in N(u)$, it stores the pointer to that entry in $F(x)$ with vertex u; this pointer is later used to erase the corresponding entry (if needed) in $O(1)$ time. See also the pseudocode of procedure HandleInsertion(u) in Algorithm 1.

When a client u gets deleted, we remove the occurrences of u from the lists $F(x)$, for all servers $x \in N(u)$. If the deleted client was matched, we first need to delete (u, v) from the matching $\mathcal{M}$, where v is the mate of u. This renders server v free, so the algorithm needs to rematch it with a free client in $N(v)$. To this end the algorithm checks if $F(v)$ is empty, and if not, the vertex at the front of the list, denoted by w, is removed from $F(v)$, and gets matched with v. See procedure HandleDeletion in Algorithm 1.

Clearly, Algorithm BasicMM is an irrevocable algorithm for maintaining an MM. Moreover, it is readily verified that the total runtime of this algorithm is $O(K)$.

Algorithm 1 BasicMM.

```
1:  procedure HandleDeletion(u)
2:      if mate(u) ≠ ∅ then                                   ▷ let v = mate(u)
3:          delete (u, v) from M
4:          if F(v) ≠ ∅ then
5:              take the first entry w from F(v) and add (v, w) to M
6:              Notify(w, matched)
7:      else
8:          Notify(u, free)
9:  procedure HandleInsertion(u)
10:     scan the neighbors of u                               ▷ in O(deg(v)) time
11:     if no free neighbor is found then Notify(u, free)
12:     else
13:         let v be an arbitrary free neighbor of u; add (u, v) to M
14: procedure Notify(u, State)                                ▷ State ∈ {free, matched}
15:     if State = free then
16:         for x ∈ N(u) do
17:             append u to F(x)          ▷ keep the pointer from u to its entry in F(x)
18:     else                                                  ▷ State = matched
19:         for x ∈ N(u) do
20:             remove u from F(x)        ▷ in O(1) time, using the pointer stored with u
```

▶ **Corollary 4.** *For any sequence of updates in the* `BASIC-dynamic` *setting, one can maintain an MM via a deterministic irrevocable algorithm with a total runtime of* $O(K)$.

3 Lower bounds for irrevocable MM in the `MEDIUM` setting

In this section we prove a strong lower bound of amortized time $\Omega(n^{1-\epsilon})$ per edge update for any constant $\epsilon < 1$, assuming that the OMv conjecture holds (Conjecture 5). This essentially means that for each vertex update, one cannot do much better than scanning the entire vertex neighborhood. Our lower bound even holds for `MEDIUM-decremental` setting; in this setting, we require that the decremental algorithm is able to handle any update sequence that consists of vertex deletions applied to any graph G and any maximal matching $\mathcal{M}$ of G.

The Online Boolean Matrix-Vector Multiplication (OMv) conjecture was introduced by Henzinger et al. [21] for proving conditional hardness of various (dynamic) problems. In the OMv problem, we are given an $n \times n$ Boolean matrix M and a sequence of n Boolean vectors arriving one by one $v_1, v_2 \ldots, v_n$. For each vector v_t where $t \in [1, n]$, we need to output the Boolean product $M \cdot v_t$ before the next vector comes. The OMv conjecture is the following:

▶ **Conjecture 5** (OMv Conjecture [21])**.** *For any constant $\epsilon > 0$, there is no algorithm that solves the OMv problem with an error probability of at most $1/3$ and runs in $O(n^{3-\epsilon})$ time.*

The same paper [21] introduced another problem, called the Online Vector-Matrix-Vector Multiplication (OuMv) problem. In this problem, we are given an $n \times n$ Boolean matrix M and a sequence of n *pairs of* Boolean vectors arriving one by one $(u_1, v_1), (u_2, v_2) \ldots, (u_n, v_n)$. The task is, for each time step $t \in [1, n]$, to output the Boolean product $u_t^\intercal M v_t$, before the next vector pair arrives. Henzinger et al. [21] showed that if the OMv conjecture holds, there is no algorithm for OuMv with total running time $O(n^{3-\epsilon})$ for any fixed $\epsilon > 0$ even if one is allowed to preprocess M in polynomial time before any vector pair arrives.

▶ **Theorem 6** (Theorem 2.4 [21]). *For any constant $\epsilon > 0$, assuming that the OMv conjecture holds, there is no algorithm with a polynomial preprocessing that solves the OuMv problem with an error probability of at most $1/3$ and runs in $O(n^{3-\epsilon})$ time.*

We first present a lower bound for the fully dynamic setting, as the proof is simpler and conveys some intuition for the decremental setting. The lower bound proof for the decremental setting follows the same principle, but we introduce an additional idea. In particular, we introduce the Stop-OuMv problem (Definition 13), which is a variant of of the OuMv problem. We show that the Stop-OuMv and OuMv are equivalent under truly subcubic time reductions. We then prove a lower bound for the decremental setting via a reduction from the Stop-OuMv problem.

Hardness from triangle detection

We also achieved a weaker lower bound, under the Triangle Detection conjecture:

▶ **Conjecture 7.** *There is a constant $0 < \delta$, such that in the Word RAM model with words of $O(\log n)$ bits, any algorithm requires $m^{1+\delta-o(1)}$ time in expectation to detect whether an m edge graph contains a triangle.*

Our lower bound holds for the decremental setting as well; due to space constraints, this lower bound is omitted.

3.1 The fully dynamic setting

Our proof of the lower bound for irrevocable MM in the `MEDIUM-dynamic` setting is via a reduction from the OuMv problem. In particular, we show that:

▶ **Theorem 8.** *If for some fixed $\epsilon > 0$, there is an algorithm that maintains an irrevocable maximal matching in the `MEDIUM-dynamic` setting with error probability at most $1/3$ in $O(K \cdot n^{1-\epsilon})$ time, then the OuMv problem can be solved in $O(n^{3-\epsilon})$ time with error probability of at most $1/3$, using $O(n^2)$ preprocessing time.*

Theorem 8 and Theorem 8 imply that, unless the OMv conjecture fails, there is no algorithm with running time $O(K \cdot n^{1-\epsilon})$ time and error probability at most $1/3$ that maintains an irrevocable maximal matching in the `MEDIUM-dynamic` setting.

Henceforth, we focus on proving Theorem 8. We construct the dynamic graph in 2 phases, starting from an empty graph. In the first phase, we are given an $n \times n$ matrix M. We construct a bipartite graph $G = (V, E)$ by adding vertices in a specific order such that the maximal matching maintained by the algorithm is perfect. The running time of the first phase is $O(n^2)$, which we charge to the preprocessing time of the algorithm for solving the OuMv problem. The second phase has n time steps, and in each time step t when we receive a pair of vectors (u_t, v_t), we delete and insert vertices to $G = (V, E)$ such that the total number of edges incident to the deleted/inserted vertices is $O(n)$. We argue that depending on the size of the maximal matching, we can determine whether $u_t^\intercal M v_t$ is 0 or 1. Since the total number of edges inserted/deleted over n time steps is $K = O(n^2)$, an algorithm of running time $O(K \cdot n^{1-\epsilon})$ for irrevocable MM implies an algorithm of running time $O(n^{3-\epsilon})$ for the OuMv problem with preprocessing time $O(n^2)$, as claimed.

We now describe each phase in turn.

Phase 1

In this phase, given an $n \times n$ matrix M, we incrementally construct a bipartite graph $G = (V, E)$ as follows. The vertex set V is the union of four vertex sets $V = V_L^* \cup V_L \cup V_R \cup V_R^*$, each has n vertices. It follows that $|V| = 4n$. There is an edge between a vertex pair $(i_L, j_R) \in V_L \times V_R$ if and only if $M[i_L, j_R] = 1$. For each vertex $i_L \in V_L$, there is a corresponding vertex $i_L^* \in V_L^*$ of degree 1 that is connected to i_L. That is (i_L, i_L^*) is the only edge incident to i_L^*. Similarly, for each vertex $j_R \in V_R$, there is a corresponding vertex $j_R^* \in V_R^*$ of degree 1 that is connected to j_R. Clearly G is a bipartite graph. We would like the maximal matching M maintained by the algorithm by the end of phase 1 to be $\{(i_L^*, i_L) : i_L \in V_L\} \cup \{(j_R^*, j_R) : j_R \in V_R\}$. To realize this assumption, we add vertices to G in the following order. First, we add all vertices of V_L^* and V_R^* one by one. We then add vertices of V_L and finally add vertices of V_R. Observe that, every time a vertex $i_L \in V_L$ is added, the algorithm will match it to the corresponding vertex $i_L^* \in V_L^*$, since it has no other neighbor. Similarly, when vertex $j_R \in V_R$ is added, the algorithm will match it to the corresponding vertex $j_R^* \in V_R^*$, since its neighbors in V_L are already matched and the algorithm is not allowed to change the matching. We conclude that:

▶ **Observation 9.** *The total number of edges of G after Phase 1 is $O(n^2)$. Also, the maximal matching after Phase 1 is $\{(i_L^*, i_L) : i_L \in V_L\} \cup \{(j_R^*, j_R) : j_R \in V_R\}$, which has size $2n$.*

Phase 2

This phase has n time steps, and in each time step t, a pair of vectors (u_t, v_t) arrives. First, for each $j \in [1, n]$ such that $v_t[j] = 1$, we *activate* the j-th vertex $j_R \in V_R$ by deleting its neighbor j_R^* in V_R^*. Let y be the number of 1-entries of v_t. Then, for each $i \in [1, n]$, we activate the i-th vertex $i_L \in V_L$ by deleting its counterpart $i_L^* \in V_L$ and query the size of the maximal matching. We call vertices in $V_L \cup V_R$ whose neighbors are deleted *active vertices*. We show in the following lemma that by querying the size of the maximal matching, we can determine the value of $u_t^\intercal M v_t$. Let x_i be the number of 1-entries of u_t whose indices are in $[1, i]$. Let $\mathcal{M}_i$ be the maximal matching following the deletion of i_L^*.

▶ **Lemma 10.** *$u_t^\intercal M v_t = 1$ if and only if there exists $i \in [n]$ such that $|\mathcal{M}_i| = 2n - y - x_i + 1$ and for every $1 \le i' < i$, $|\mathcal{M}_{i'}| = 2n - y - x_{i'}$.*

Proof. Observe that when the algorithm deletes i_L^*, there are $y + x_i$ vertices that have been deleted from $V_L^* \cup V_R^*$ in total. Thus, $|\mathcal{M}_i| \ge 2n - y - x_i$. The algorithm may add a new matching edge between a vertex in V_L and a vertex in V_R after deleting vertices from $V_L^* \cup V_R^*$. We argue that the number of such edges is at most 1, and that this happens only when $u_t^\intercal M v_t = 1$. Recall that $u_t^\intercal M v_t = 1$ is either 0 or 1. Furthermore, $u_t^\intercal M v_t = 1$ if and only if there exists a unique pair of indices $(i, j) \in [n] \times [n]$ such that $u_t[i] = M[i, j] = v_t[j] = 1$. It follows that, before i_L is activated, there is no edge from active vertices in V_L to active vertices in V_R. Thus, $|\mathcal{M}_{i'}| = 2n - y - x_{i'}$ for every $1 \le i' < i$. When i_L^* is deleted, i_L is free and hence, will be matched to j_R. Thus, $|\mathcal{M}_i| = 2n - y - x_i + 1$. ◀

Lemma 10 implies that we are able to decide whether $u_t^\intercal M v_t = 1$ by querying the size of the maximal matching, and find the first index $i \in [n]$ such that $|\mathcal{M}_i| = 2n - y - x_i + 1$ (we stop the phase at that i). If we cannot find such an index i, then $u_t^\intercal M v_t = 0$.

Next, we need to "undo" the changes of the maximal matching to prepare for the arrival of the next vector pair (u_{t+1}, v_{t+1}). If $u_t^\intercal M v_t = 0$, all we need to do is to insert the deleted vertices of $V_L^* \cup V_R^*$ back into the graph, and the algorithm will match all of them with their

counterparts in $V_L \cup V_R$. If $u_t^\intercal M v_t = 1$, let i be the first index such that $|\mathcal{M}_i| = 2n - y - x_i + 1$. We delete i_L from V_L, then insert the deleted vertices of $V_L^* \cup V_R^*$ back, and finally insert i_L back into the graph. Note that i_L will not be matched to any $j_R \in V_R$ since each j_R was matched to its counterpart in j_R^*, which was inserted before i_L. Thus, i_L will be matched to i_L^* and the algorithm can proceed to the next time step.

▶ **Observation 11.** *At each time step $t \in [n]$, the number of edges deleted and inserted is $O(n)$.*

Proof. Since the algorithm deletes at most n vertices of degree 1 (in $V_L^* \cup V_R^*$) and at most one vertex in V_L, the number of deleted edges is at most $2n$. The number of inserted edges is exactly the number of of deleted edges. Thus, the observation follows. ◀

Proof of Theorem 8. By Observation 9, the running time in Phase 1 is $O(n^2)$, which we charge to the preprocessing time of M. Phase 2 has n time steps, and by Lemma 10, we can correctly decide whether $u_t^\intercal M v_t = 1$ for each time step t by querying the size of the maximal matching. By Observation 11, the total number of edges over the entire update sequence is $K = O(n^2)$, including the edges updated in Phase 1. It follows that if the running time to maintain an irrevocable MM is $O(Kn^{1-\epsilon}) = O(n^{3-\epsilon})$, then the total running time to solve OuMv is $O(n^{3-\epsilon})$. ◀

3.2 The decremental setting

Our goal is to show that, such an algorithm with running time $O(K \cdot n^{1-\epsilon})$ where K is the total number of edges deleted implies that the OuMv problem can be solved in $O(n^{3-\epsilon})$ time.

▶ **Theorem 12.** *If for some $\epsilon > 0$, there is an algorithm that maintains an irrevocable maximal matching for any graph G and its maximal matching in the `MEDIUM-decremental` setting with error probability at most $1/3$ in $O(K \cdot n^{1-\epsilon})$ time, where K is the overall number of edges being deleted from the graph throughout the entire sequence of updates, then the OMv conjecture is false.*

Recall that in the lower bound proof in the fully dynamic setting (Theorem 8), in each time step $t \in [n]$ in Phase 2, we use vertex insertions to restore the matching before the next pair of vectors (u_{t+1}, v_{t+1}) arrives. In particular, we need to insert back vertices of $V_L^* \cup V_R^*$ and a vertex $i_L \in V_L$.

We would like to use auxiliary vertices to simulate all re-insertions of nodes i_L^* by deletions. Here is an idea. In our fully dynamic reduction, we removed vertices i_L^* to free up their matches i_L in the matching. At the end of the phase, we want to re-insert i_L^* so it can be matched to i_L again. The idea is that instead of a single i_L^*, we will have a copy $i_{j,L}^*$ for each $j = 1, \ldots, n$, one for each phase. At the beginning of the t-th phase, i_L is matched to some $i_{t,L}^*$; in particular, the initial matching contains $(i_L, i_{1,L}^*)$. During the phase $i_{t,L}^*$ is deleted to make i_L available, and at the end of the phase we want i_L to get matched to $i_{t+1,L}^*$, while $i_{t,L}^*$ stays deleted (rather than being re-inserted). Since we have at most n phases, we would only need n^2 total nodes $i_{j,L}^*$. To make $i_{t+1,L}^*$ available to be matched, in the initial graph, for every $j > 1$, each $i_{j,L}^*$ has a single node $i_{j,L}^{**}$ connected only to it, and the edge $(i_{j,L}^*, i_{j,L}^{**})$ is in the initial matching for all $j > 1$. Then to make $i_{j,L}^*$ available, it suffices to delete $i_{j,L}^{**}$.

Doing this, we simulate all re-insertions except those of i_L by deletions at the cost of having $O(n^2)$ vertices; the number of edges stays $O(n^2)$. However, it seems difficult to simulate the re-insertion of i_L. To handle this, we use a different idea.

Here, we instead exploit the fact that in the fully dynamic construction the cases when $u_t M v_t = 0$ were easy to handle, and that the difficulties came when we found that $u_t M v_t = 1$.

We introduce a variant of the OuMv problem, which we call the Stop-OuMv problem, where we can stop the online update sequence of pairs of vectors whenever we encounter the first time step t such that $u_t^\intercal M v_t = 1$. Namely, the goal of the Stop-OuMv problem is to detect the first time step $\ell \in [n]$ where $u_\ell^\intercal M v_\ell = 1$. While Stop-OuMv seems easier than OuMv , we will show that the two problems are equivalent under truly subcubic reductions. Thus, it suffices to reduce Stop-OuMv to our decremental MM problem.

We proceed similarly to our fully dynamic reduction. The key idea is that whenever we detect that $u_t^\intercal M v_t = 1$, we do not need to worry about the matching at future time steps. As a result, there is no need to insert i_L back and hence, we can avoid vertex insertion entirely. The remaining technical challenge is to show that an $O(n^{3-\epsilon})$-time algorithm for the Stop-OuMv problem falsifies the OMv conjecture. (The reduction from Stop-OuMv to OuMv is trivial.)

We start by giving the formal definition of the Stop-OuMv problem.

▶ **Definition 13** (Stop-OuMv Problem)**.** *We are given an $n \times n$ Boolean matrix M and a sequence of n* pairs of *Boolean vectors arriving one by one $(u_1, v_1), (u_2, v_2) \ldots, (u_n, v_n)$. At each time step $t \in [1, n]$, we compute the Boolean product $u_t^\intercal M v_t$, and if $u_t^\intercal M v_t = 0$, we proceed to the next vector pair; otherwise, we stop and return that $u_t^\intercal M v_t = 1$.*

Later in the section we prove the following theorem, giving an equivalence between OMv and Stop-OuMv .

▶ **Theorem 14.** *For any constant $\epsilon > 0$, assuming that the OMv conjecture holds, there is no algorithm with polynomial preprocessing that solves the Stop-OuMv problem with an error probability of at most $1/3$ and runs in $O(n^{3-\epsilon})$ time.*

We are now ready to prove Theorem 12, assuming Theorem 14.

Proof of Theorem 12. We will show that a decremental algorithm as stated in Theorem 12 implies an algorithm with running time $O(n^{3-\epsilon})$ for the Stop-OuMv problem, thereby falsifying the OMv conjecture by Theorem 14.

Given a matrix M of the Stop-OuMv problem, we first construct a graph $G = (V, E)$ and its maximal matching $\mathcal{M}$. Specifically, V is the union of six vertex sets $V = V_L^{**} \cup V_L^* \cup V_L \cup V_R \cup V_R^* \cup V_R^{**}$. V_L and V_R have n vertices each, and there is an edge between a vertex pair $(i_L, j_R) \in V_L \times V_R$ if and only if $M[i_L, j_R] = 1$. For each vertex $i_L \in V_L$, we add n vertices $\{i_{1,L}^*, \ldots, i_{n,L}^*\}$ to V_L^*; each vertex in this set is connected to i_L by an edge. We apply the same construction to each vertex $j_R \in V_R$; the new neighbors of j_R are added to V_R^*. Finally, for each vertex $i_{t,L}^* \in V_L^*$ for $t > 1$, we add a vertex $i_{t,L}^{**}$ in V_L^{**} and connect $i_{t,L}^{**}$ to $i_{t,L}^*$ by an edge. We apply the same construction to each vertex in V_R^*, but new vertices are are added to V_R^{**}. This completes the construction of G. See Figure 1 for an illustration.

The maximal matching $\mathcal{M}$ we choose is $\{(i_{1,L}^*, i_L) : i_L \in V_L\} \cup \{(j_{1,R}^*, j_R) : j_R \in V_R\}$ plus all the edges incident to $V_L^{**} \cup V_R^{**}$, except for those who are matched to vertices in $V_L \cup V_R$. This can be accomplished by first inserting the edges of $\mathcal{M}$, and then the rest of the edges.

Note by the construction that vertices in $V_L^{**} \cup V_R^{**}$ have degree 1 in G and those in $V_L^* \cup V_R^*$ have degree 2.

▶ **Observation 15.** *Let m be the number of 1-entries of the matrix M. Then $|\mathcal{M}| = 2n^2$, and G has $m + 4n^2 - 2n$ edges and $4n^2$ vertices.*

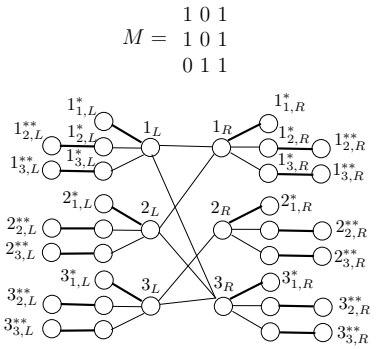

Figure 1 Here we see an example construction of the reduction graph obtained from the 3×3 matrix M. The edges in the initial matching $\mathcal{M}$ are shown in bold.

Since $m = O(n^2)$, by Observation 15, G and $\mathcal{M}$ can be constructed in $O(n^2)$ time; we charge this running time to the preprocessing time of the Stop-OuMv problem.

Next, for each time step $t \in [n]$, the pair of vectors (u_t, v_t) arrives. We will inductively maintain the following invariant:

> **Matching Invariant:** At the beginning of time step t, for every vertex $i_L \in V_L$ ($j_R \in V_L$), there is an edge $(i_L, i^*_{t,L})$ (resp. $(j_R, j^*_{t,R})$) in the maximal matching.

Clearly, the Matching Invariant holds when $t = 1$ by the construction of $\mathcal{M}$. For each $j \in [n]$ such that $v_t[j] = 1$, we *activate* the j-th vertex $j_R \in V_R$ by deleting its neighbor $j^*_{t,R} \in V^*_R$. We apply the same operation for each $i \in [n]$ such that $u_t[i] = 1$. We call vertices in $V_L \cup V_R$ whose neighbors are deleted *active vertices*. Let z_t be the total number of 1-entries of u_t and v_t. We call the first time step ℓ where $u_\ell^\intercal M v_\ell = 1$ the *critical time step*. Observe that in each time step $t \le \ell$, we delete exactly z_t edges from the initial maximal matching $\mathcal{M}$. Furthermore, the algorithm will add exactly one edge to $\mathcal{M}$ at the critical time step ℓ and the matching edge is (i_L, j_R) where (i_L, j_R) is the (unique) pair such that $u_t[i_L] = M[i_L, j_R] = v_t[j_R] = 1$. Thus, we have:

▶ **Observation 16.** *Let $\mathcal{M}_t$ be the matching $\mathcal{M}$ after time step t. Then $|\mathcal{M}_t| = 2n^2 - \sum_{k=1}^{t} z_k$ for any $t < \ell$ and $|\mathcal{M}_t| = 2n^2 - \sum_{k=1}^{t} z_t + 1$ when $t = \ell$.*

From Observation 16, our idea to detect the critical time step is to maintain $Z_t = \sum_{k=1}^{t} z_k$ after each time step. At time step t, after activating vertices in $V_L \cup V_R$ as described above, we query the size of the maximal matching $\mathcal{M}_t$ and check it against $2n^2 - Z_t$. If $|\mathcal{M}_t| = 2n^2 - Z_t$, then $u_t^\intercal M v_t = 0$. Otherwise, $|\mathcal{M}_t| = 2n^2 - Z_t + 1$, and it follows that $t = \ell$. The algorithm deletes every vertex of G afterward.

To prepare for the arrival of the vector pair in the next time step $t + 1$, we need to maintain the Matching Invariant. We do so by *deactivating* the active vertices in $V_L \cup V_R$. Note that we only need to deactivate vertices when $t < \ell$, and in this case, there is no edge between two active vertices since $u_t^\intercal M v_t = 0$. Specifically, we deactivate an active vertex $i_L \in V_L$ by deleting vertex $i^{**}_{t+1,L}$ in V^{**}_L. This deletion leaves $i^*_{t+1,L}$ free, and the algorithm will match i_L to $i^*_{t+1,L}$. We deactivate an active vertex $j_R \in V_R$ in the same way. Thus, Matching Invariant is maintained.

We now bound the total running time of solving the Stop-OuMv problem. Clearly $K = O(n^2)$ by Observation 15. Maintaining $\{Z_t\}_{t=1}^{n}$ takes $O(n^2)$ time. Thus, the runtime of the algorithm, by the assumption of Theorem 12, is $O(Kn^{1-\epsilon}) + O(n^2) = O(n^{3-\epsilon})$. ◀

Finally, we close this section by proving Theorem 14 by a reduction from OuMv problem.

Reducing OuMv to Stop-OuMv: Proof of Theorem 14.

Suppose that we have an algorithm for solving the Stop-OuMv problem with a polynomial preprocessing time $p(n)$ and total computation time $O(n^{3-\epsilon})$ for some fixed $\epsilon > 0$, we will show that the OuMv can be solved in $O(n^{3-2\epsilon/3})$ time by using $O(n^{5/3}p(n^{2/3}))$ preprocessing time; the error probability of both algorithms is $1/3$. Thus, Theorem 14 follows from Theorem 6.

Given an instance of the OuMv problem with an $n \times n$ matrix M, we partition M into $n^{2(1-a)}$ blocks of size $n^a \times n^a$ for $a = 2/3$. Then we construct an instance of Stop-OuMv for each block of M.

Next, for each time step $t \in [n]$ where a pair of vectors (u_t, v_t) arrives, we split u_t and v_t into blocks of length n^a. For each pair $(x, y) \in [n^{1-a}] \times [n^{1-a}]$, we give the x-th block of u_t, denoted by $u_t(x)$, and y-th block of v_t, denoted by $v_t(y)$, to the Stop-OuMv data structure for the (x, y)-block of M, denoted by $M(x, y)$. If the Stop-OuMv data structure for $M(x, y)$ outputs 0, we continue for the next pair. Otherwise, we return that $u_t^\intercal M v_t = 1$ and initiate a new Stop-OuMv data structure for block $M(x, y)$. Furthermore, whenever the current Stop-OuMv data structure for $M(x, y)$ "overflows", that is, we has given exactly n^a vector pairs to the data structure, we initiate a new Stop-OuMv data structure for $M(x, y)$.

We first bound the preprocessing time. Observe that for each block $M(x, y)$, the total number of Stop-OuMv data structures needed is at most n. Thus, we can initialize exactly n Stop-OuMv data structures for each block $M(x, y)$ before any vector pair arrives. The total time is $O(n^{1+2(1-a)}p(n^a)) = O(n^{5/3})p(n^{2/3})$, which we will charge to the preprocessing time of OuMv algorithm. We note that the number of Stop-OuMv data structures actually used by the algorithm for each block $M(x, y)$ could be much smaller than n, and hence, many copies of the Stop-OuMv data structures for $M(x, y)$ might go unused.

We now bound the online running time. Since the number of vector pairs given to the Stop-OuMv data structures for each block $M(x, y)$ is n over the entire sequence, we need at most n^{1-a} Stop-OuMv data structures for $M(x, y)$ to handle overflows. Furthermore, we need at most n additional Stop-OuMv data structures over all pairs (x, y) due to that $u_t^\intercal(x)M(x, y)v_t(y) = 1$. Thus, the total running time of all Stop-OuMv data structures is:

$$(n^{(1-a)} \cdot n^{2(1-a)} + n) \cdot O(n^{a(3-\epsilon)}) = 2n \cdot O(n^{(2/3)(3-\epsilon)}) = O(n^{3-2\epsilon/3}),$$

as claimed.

4 A 1-revocable MM in the `GENERAL-dynamic` setting

In this section, we present an MM algorithm for the `GENERAL-dynamic` setting that works in total time $O(K)$.

We start presenting the algorithms by introducing some notation that shall be used throughout this section. For each vertex v we maintain an estimated degree $\deg'(v) = \deg'_i(v)$: We set $\deg'(v)$ to $\deg(v) = |N(v)|$ once v is inserted to the graph (line 8 in Algorithm 2), and from that point onwards we update v's estimated degree whenever it differs from its degree by a factor of 2 (line 15 in Algorithm 2).

We partition the (dynamic) vertex set $V = V_i$ into $R = R_i$, $S = S_i$, and $O = O_i$: The sets of *risky*, *safe*, and *outlier* vertices, respectively. A matched vertex u is designated as risky only if $\deg'(u) > 2 \cdot \deg'(v)$, where $v = \text{mate}(u)$; otherwise it is designated as safe. Thus, at most one endpoint (the one of higher estimated degree) of any matched edge may be risky. A free

vertex can be either risky or an outlier. Upon any change in the estimated degree of a risky matched vertex, the algorithm designates that vertex as safe (even if $\deg'(u) > 2 \cdot \deg'(v)$, where $v = \text{mate}(u)$).

For a vertex u, the sets of its neighbors, risky neighbors, safe neighbors, and outlier neighbors are denoted by $N(u), R(u)$, $S(u)$, and $O(u)$, respectively. We further partition $R(u)$ into $R_{\leq}(u)$ and $R_{>}(u)$, which contain the risky neighbors of u with estimated degree $\leq \deg'(u)$ and $> \deg'(u)$, respectively. The algorithm maintains the following invariant.

▶ **Invariant 1.** *For each vertex v, we maintain a partition of $N(u)$ into $S(u), R_{\leq}(u), R_{>}(u)$, and $O(u)$. All vertices in $S(u)$ are matched and all free neighbors of u are in $R_{\leq}(u) \cup R_{>}(u) \cup O(u)$. (The set $O(u)$ is the set of outlier neighbors of u; all vertices in $O(u)$ are free.)*

The main intuition behind partitioning the neighborhood of each vertex into risky, safe, and outlier vertices is that for any safe vertex we have "credits" for scanning its entire neighborhood. This informally means that we do not need to maintain invariants for safe vertices, and the goal of the invariants is to provide a way to handle the risky vertices without scanning their neighborhoods.

The following invariant will allow us to argue that the matching maintained by update algorithm GENERAL-DYNAMICMM is in fact maximal.

▶ **Invariant 2.** *For each risky vertex, all its free neighbors (if any) belong to $R_{\leq}(u) \cup O(u)$.*

4.1 The update algorithm

Whenever there is an update, some vertices may switch their states from risky to safe and vice versa; our algorithm will check and perform these state updates. (See also procedure HANDLEUPDATE(u) in Algorithm 2.) To keep the presentation of our ideas clean, we ignore the details of these state changes.

Suppose first that u is safe. In this case we scan its entire neighborhood. If u has a free neighbor, we match u to its free neighbor of highest estimated degree, denoted w. If $w \in O(u)$, we move w from $O(u)$ to the subset among $R_{\leq}(u), R_{>}(u), S(u)$ to which it should belong, and then update the data structures of w's neighbors accordingly. (E.g., if $\deg'(w) \leq 2 \cdot \deg'(u)$, w becomes safe and moves to $S(u)$.) Similarly, we designate u as either risky or safe as appropriate, and update the data structures of u's neighbors accordingly. In the case that u does not have any free neighbor, we leave u free, and designate u as an outlier by moving it from $S(w)$ to $O(w)$ for each neighbor w of u. (Note that we only need to designate u as an outlier for its risky neighbors w such that $u \notin R_{\leq}(w)$, so as to maintain Invariant 2, but we might as well designate u as an outlier with respect to all its neighbors.)

If u is risky and $R_{\leq}(u) \cup O(u) = \emptyset$, we leave u free. In what follows we assume that u is risky and $R_{\leq}(u) \cup O(u) \neq \emptyset$. If $O(u) \neq \emptyset$, we match u with an arbitrary (free) vertex $w \in O(u)$, move w from $O(u)$ to the subset among $R_{\leq}(u), R_{>}(u), S(u)$ to which it should belong. Otherwise we match u with an arbitrary vertex $w \in R_{\leq}(u)$. Finally, if w was previously matched to w', we remove edge (w, w') from the matching, and handle w' recursively. Since w was a risky vertex, it implies that w' was previously designated as safe; thus we handle w' recursively *as a safe vertex*.

If the estimated degree of some vertex x changes in the current vertex update (line 15 in Algorithm 2), we designate is as safe and update the matching and invariants using the same procedure, HANDLEUPDATE(x).

This completes the description of the algorithm. We are ready to prove Theorem 2. We argue correctness of the algorithm in Lemma 17 and analyze its running time in Lemma 18.

We proceed to argue correctness of the algorithm.

Algorithm 2 GENERAL-DYNAMICMM.

procedure HANDLEDELETION(u)
  let Q be an empty queue ▷ vertices to be processed by HANDLEUPDATE
  if mate(u) $\neq \emptyset$ **then**
    delete $(u, \text{mate}(u))$ from $\mathcal{M}$ and add mate(u) to Q
  UPDATEDEGREES($u, Q, -1$) ▷ decrease the degree of each neighbor by one
  for $x \in Q$ **do** HANDLEUPDATE(x)
procedure HANDLEINSERTION(u)
  $\deg'(u) \leftarrow |N(u)|$; $\deg(u) \leftarrow \deg'(u)$
  NOTIFY(u, safe)
  let Q be a list containing vertex u; UPDATEDEGREES($u, Q, +1$)
  for $x \in Q$ **do** HANDLEUPDATE(x)
procedure UPDATEDEGREES(u, Q, sgn)
  for $x \in N(u)$ **do**
    $\deg(x) \leftarrow \deg(x) + \text{sgn}$
    if $\deg(x) \leq \deg'(x)/2$ **or** $\deg(x) \geq 2 \cdot \deg'(x)$ **then**
      $\deg'(x) \leftarrow \deg(x)$
      NOTIFY(x, safe)
      if mate(x) $= \emptyset$ **then** add x to Q
procedure HANDLEUPDATE(u)
  if state(u) $=$ safe **then**
    choose $w \in N(u)$ maximizing $\deg'(w)$ among free neighbors of u
    if u has no free neighbors **then** NOTIFY(u, outlier)
    else MATCH(u, w)
  else if $O(u) \neq \emptyset$ **then** take an arbitrary $w \in O(u)$ and MATCH(u, w) ▷ u is risky
  else if $R_{\leq}(u) \neq \emptyset$ **then** take an arbitrary $w \in R_{\leq}(u)$ and MATCH(u, w)
procedure MATCH(u, w)
  add (u, w) to $\mathcal{M}$
  if $\deg'(u) \leq 2 \cdot \deg'(w)$ **then** NOTIFY(u, safe) **else** NOTIFY(u, risky)
  if $\deg'(w) \leq 2 \cdot \deg'(u)$ **then** NOTIFY(w, safe) **else** NOTIFY(w, risky)
  if w was matched to w' **then**
    remove (w, w') from $\mathcal{M}$
    HANDLEUPDATE(w')
procedure NOTIFY($u, State$) ▷ $State \in$ {safe, risky, outlier}
  if state(u) $= State$ **then exit** ▷ do nothing if state remains the same
  state(u) $\leftarrow B$
  for $x \in N(u)$ **do** move u to appropriate set among $S(x)$, $R_{\leq}(x)$, and $R_{>}(x)$

▶ **Lemma 17.** *Algorithm GENERAL-DYNAMICMM maintains an MM in the* `GENERAL-dynamic` *setting throughout the entire sequence of updates. Moreover, following any update, the algorithm removes at most a single edge from the maintained matching $\mathcal{M}$.*

Proof. We first argue that the maintained matching is maximal at all times. By the description of the algorithm, a safe vertex that becomes free is left free if and only if it does not have any free neighbors. Moreover, a risky vertex u may be left free if and only if $R_{\leq}(u) \cup O(u) = \emptyset$, while there cannot be any free vertex in $R_{>}(u) \cup S(u)$ by Invariants 1 and 2. It follows that any vertex that is left free throughout the execution of the algorithm does not lead to a violation of $\mathcal{M}$'s maximality.

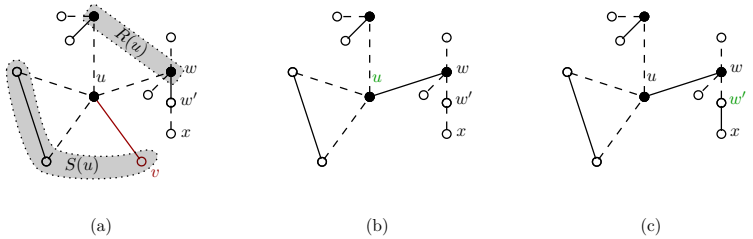

Figure 2 An illustration of an update step of algorithm GENERAL-DYNAMICMM. Solid edges are in the maintained matching $\mathcal{M}$ and solid vertices are risky. **(a)** Before deletion of vertex v, it is matched to a risky vertex u. Vertex u has two risky neighbors constituting $R(u)$ and three safe neighbors constituting $S(u)$. **(b)** Following deletion of v, vertex u becomes free and the algorithm tries to rematch it (cf. procedure HANDLEUPDATE(u) in Algorithm 2). Since $O(u)$ is empty, vertex w is chosen from $R_{\leq}(u)$ and (u, w) is added to $\mathcal{M}$. (This change might trigger u and/or w to become safe, which we ignored in this example for simplicity.) At this stage, (w, w') is removed from $\mathcal{M}$ and w' becomes free. Since w was risky while matched to (w, w'), this means that w' has to be safe. **(c)** The algorithm tries to rematch w' (recursively invoking HANDLEUPDATE(w')). Vertex x which is an outlier is chosen to be matched to w'.

To see that the algorithm is a 1-revocable algorithm, note that the only case in which the algorithm removes an edge from the matching is when a *risky* vertex u becomes free due to the adversary. More specifically, the only case in which the algorithm removes an edge (w, w') from the matching is when u is risky, $O(u) = \emptyset$ and $R_{\leq}(u) \neq \emptyset$, and then an arbitrary vertex w from the set $R_{\leq}(u)$ is chosen as a mate for u. Then the algorithm proceeds recursively to handling w', the old mate of w, which must be safe, since w was risky. Consequently, the algorithm will not remove any edge from the matching as part of this recursive call.

It is easy to verify that the algorithm never violates the validity of Invariant 1.

Finally, we argue that Invariant 2 remains valid throughout the execution of the algorithm. Suppose first that u is safe. If u has a free neighbor, it is matched to its free neighbor of highest estimated degree, denoted w. If $\deg'(w) > \deg'(u)$, u remains safe and there cannot be any violation to the invariant with respect to u. As for w, even if it becomes risky, it does not have any free neighbors by the maximality of the matching, thus the invariant holds for it vacuously. Otherwise ($\deg'(w) \leq \deg'(u)$), it is possible that u becomes risky, but there is no violation to the invariant since all free neighbors z of u satisfy $\deg'(z) \leq \deg'(w) \leq \deg'(u)$, hence they all belong to $R_{\leq}(u) \cup O(u)$. There is no violation with respect to w either, as it becomes safe. If u does not have any free neighbor, it becomes free and is designated as an outlier, which cannot lead to any violation to the invariant.

We henceforth assume that u is risky. If $R_{\leq}(u) \cup O(u) = \emptyset$, u is left free, and it is immediate that there is no violation to Invariant 2 for all neighbors of u. As for u itself, there cannot be any free vertex in $R_{>}(u) \cup S(u)$ by Invariant 2. Otherwise, u is matched with an arbitrary (free) vertex $w \in O(u)$, and if none exists it is matched to an arbitrary vertex $w \in R_{\leq}(u)$. In this case, it is immediate that there is no violation to Invariant 2 with respect to u. As for w, even if it becomes risky, either it belonged to $O(u)$, and then it cannot have any free neighbors by the maximality of the matching, or it belonged to $R_{\leq}(u)$ (and was risky), and then all its free neighbors must belong to $R_{\leq}(w) \cup O(w)$ by Invariant 2.

If w was matched to w', w' becomes free, which could potentially violate the validity of the matching's maximality and the invariants. However, the algorithm does not stop at this stage, but rather proceeds to handling w' recursively, hence the validity follows by induction.

Finally, upon a change in the estimated degree of any vertex z, the vertex is designated as safe, which cannot lead to any violation to Invariant 2. ◀

▶ **Lemma 18.** *Algorithm GENERAL-DYNAMICMM has a total runtime of $O(K)$.*

Proof. Before going into the details of our running time analysis, we briefly describe the intuition. For simplicity, we will first explain the intuition without taking into account the cost of $O(\deg'(w))$ incurred in line 24; later on (in Claims 19 and 20), we address this issue. Our focus is on the most difficult case where a vertex v is deleted from the graph by the adversary. We could afford to spend $O(\deg'(v))$ time, since we can charge this time to edges incident to v; each edge will be charged only $O(1)$ time. If the mate of v, say u, is also a safe vertex, ignoring the running time of the recursive call, HANDLEUPDATE(u) takes $O(\deg'(u))$ time, which is also $O(\deg'(v))$ since $\deg'(u) \leq 2\deg'(v)$ by the definition of safe vertices. Thus, the running time is within our budget. However, if u is risky, we could not afford to scan all neighbors of u. Instead, we can show that the running time of HANDLEUPDATE(u), ignoring the recursive calls, is $O(1)$ if $R_{\leq}(u) = \emptyset$ – in this case, the running time is within our budget again – or is $O(\deg'(w))$ where w is the vertex chosen to match u in line 25. However, $\deg'(w)$ could be much larger than $\deg'(v)$ and hence, we could not charge the running time of $O(\deg'(w))$ to the deletion of v. Our key idea to resolve this case is to charge this running time to the newly created matching edge (u, w). The cost associated with each matching edge will be paid when the matching edge is deleted either by the adversary or removed by the algorithm. Note that we can assume that eventually every vertex of the graph is deleted. This assumption can be enforced by appending to the update sequence the deletions of the remaining vertices of the graph; the cost of the algorithm is increased by at most a factor of 2.

We now present the details of our argument. As mentioned above, each matching edge will be charged some cost arising from handling the updates of vertices. We control the cost associated with each matched edge by maintaining the following invariant.

▶ **Invariant 3.** *Each edge $(u, v) \in \mathcal{M}$ is charged a cost of at most $c \cdot \min(\deg'(u), \deg'(v))$ for a sufficiently large constant c.*

We now focus on bounding the running time of HANDLEUPDATE(u). We consider two cases: u is safe and u is risky. The former case will be handled in Claim 19 and the latter case will be handled in Claim 20.

▷ Claim 19. If u is safe, then the cost of HANDLEUPDATE(u) is $c_0 \deg'(u)$, where c_0 is a constant independent of c, plus the cost charged to the new matching edge (u, w) (if any) that satisfies Invariant 3 when $c \geq c_0$.

Proof. Observe that the algorithm spends only $O(\deg'(u))$ time if a free neighbor $w \in R_{>}(u)$ exists (lines 20–23). In case that no free neighbor has been found, u becomes an outlier and we assign a credit of $O(\deg'(u))$ to it. (We can allow to assign this credit since u is a safe vertex.) Otherwise, if there is a free vertex w in $O(u)$, we add (u, w) to the matching and spend $O(\deg'(w))$ time to notify the neighbors of w about it new status. This cost is charged to the credit assigned to w at the time it switched from being a safe vertex to an outlier, as described above. The credit is spent only once – at the point when the outlier becomes a safe/risky vertex. If no match has been found, then u gets matched to $w \in R_{\leq}(u)$ (if $R_{\leq}(u) \neq \emptyset$, meaning that:

$$\deg'(u) \geq \deg'(w). \tag{1}$$

Ignoring the recursive call on the neighbor w' of w or the possible cost charged to the outlier, the total running time is $c_0 \deg'(u)$.

We now handle the recursive call HandleUpdate(w') (line 32). Since w is risky, w' must be safe. It follows that $\deg'(w) > 2\deg'(w')$. Thus, the recursive invocation of HandleUpdate(w') is performed on a safe vertex with a degree lower by at least a factor of two. The cost of HandleUpdate(w') is $c_0 \deg'(w')$. Thus, we only need to charge (i) the cost $c_0 \deg'(w')$ due to HandleUpdate(w') and (ii) the cost associated with the matching edge (w, w') (since it is removed from $\mathcal{M}$) to the new matching edge (u, w). By Invariant 3, and Equation (1), the total cost charged to (u, w) is at most $c_0 \deg'(w') + c \deg'(w') \leq c \min(\deg'(u), \deg'(w))$, when $c \geq c_0$. Invariant 3 now follows. ◁

▷ Claim 20. If u is risky, then the cost of HandleUpdate(u) is $O(1)$ plus the cost charged to the new matching edge (u, w) (if any) that satisfies Invariant 3 when $c \geq 3c_0$. Here c_0 is a sufficiently large constant independent of c.

Proof. If there a free vertex w in $O(u)$ (line 24), then the algorithm spends $O(\deg'(w))$ time notifying neighbors of w. This cost is charged to the credit assigned to w at the time it became an outlier. The credit is spent only once – at the point when the outlier becomes a safe/risky vertex. Otherwise, if $R_{\leq}(u)$ is not empty, a vertex $w \in R_{\leq}(u)$ is chosen to be matched to u (line 25). The cost of notifying w's neighbors of its changed status is $O(\deg'(w)) \leq c_0 \deg'(w)$. We charge this cost to the new matching edge (u, w).

If w is matched to w', we need to charge additional costs to (u, w). In particular, we charge to (u, w) the cost associated with the old matching edge (w, w'), which is at most $c \deg'(w')$ by Invariant 3, and the cost due to HandleUpdate(w'). Since w' is safe, the cost of HandleUpdate(w') in line 32 is $c_0 \deg'(w')$ plus the cost charged to the new matching edge (if any) incident to w' by Claim 19. We do not need to worry about the cost charged to the new matching edge incident to w' since Invariant 3 is satisfied, and hence we only charge $c_0 \deg'(w')$ to edge (u, w). It follows that (u, w) is charged a total cost of at most $c_0 \deg'(w) + c_0 \deg'(w') + c \deg'(w') \leq c \min(\deg'(u), \deg'(w))$, when $c \geq 3c_0$. In the last inequality, we use the fact that w is risky and hence $\deg'(w) \geq 2\deg'(w')$. Invariant 3 now follows. ◁

We'll now complete the proof of Lemma 18 by considering each type of update separately.

First, we consider the case when u is inserted into the graph. The cost of notifying its neighbors in line 9 is $O(\deg(u))$ and the cost of update its neighbor's degrees in line 10, *excluding the cost to handle status changes of u's neighbors in lines 16–18,* is $O(\deg(u))$. (By the end of this proof, we will discuss the cost of handling lines 16–18.) Thus, we can charge the insertion cost to edges incident to u, each is charged $O(1)$ cost.

We analyze the case when u is deleted from the graph. If u is free, we only update the degree of neighbors of u, and the cost is $O(\deg(u))$ *excluding the cost to handle status changes of u's neighbors in lines 16–18.* Thus, we can charge this cost to the deletion of u; each edge incident to u is charged $O(1)$ cost. If u is not free, we denote by w the mate of u. The algorithm will call HandleUpdate(w) as w is added to Q in line 4. We first charge the costs associated with matching edge (u, w), which is $O(\deg'(u)) = O(\deg(u))$ by Invariant 3, to edges incident to u, each is charged $O(1)$ cost. Next, if w is safe, the cost of handling w is $O(\deg'(w)) = O(\deg(w)) = O(\deg(u))$ by Claim 19. Note that the cost charged to the matching edge incident to w already satisfies Invariant 3 by Claim 19. Thus, we only need to charge the cost of handling w to incident edges of u, each is charged a cost of $O(1)$. If w is risky, by the same argument and Claim 20, we only need to charge the cost of $O(1)$ due to HandleUpdate(w) to incident edges of u. Summarizing, the total cost charged to each edge incident to u is $O(1)$.

It remains to bound the update time in lines 16–18. We observe that the total cost is $O(\deg'(x))$. This cost consists of the notification cost in line 17 and the cost of HandleUpdate(x) (as it is added to the queue Q in line 18) which are both $O(\deg'(x))$ by Claim 19, since x is a safe vertex. (Here we do not account for the cost charged to the new matching edge incident to x in HandleUpdate(x).) Thus, we can charge the cost of the update in lines 16–18 to the edges incident to x being added/inserted which lead to changes in the estimated degree of x; each edge is charged $O(1)$ cost as the number of such edges is $\Omega(\deg'(x))$. ◀

5 An $(\frac{e}{e-1})$-approximation in the BASIC-dynamic setting

In this section, we switch our attention back to the BASIC-dynamic setting, for which we provide an $(\frac{e}{e-1})$-approximate algorithm for the MCM problem. This approximation is achieved by building on the celebrated ranking algorithm by Karp, Vazirani, and Vazirani [25]. Specifically, we demonstrate how to maintain a matching corresponding to the output of the ranking algorithm in a total runtime of $O(K)$, assuming an oblivious adversary.

We briefly recall the ranking algorithm. The algorithm works in a setting akin to the BASIC-incremental setting: A set of n servers S is fixed and the clients arrive to C one after another, starting from $C = \emptyset$ and ending with $|C| = |S| = n$. At the outset, the algorithm samples a random permutation σ over all n servers, which assigns a unique *rank* $\sigma(s)$ to any server $s \in S$. Upon the arrival of a new client, the algorithm scans its neighborhood and *irrevocably* matches it with its free neighboring server of lowest rank, if any; if none exists, the client is left (forever) free. This completes the description of the ranking algorithm in the online setting, as given by [25]. For the analysis, we use the following theorem by [25]; several different proofs of this theorem have been proposed (see, e.g., [15, 10]).

▶ **Theorem 21** ([25]). *If the maximum matching size of the final bipartite graph is η, then the* expected *size of the matching provided by the ranking algorithm is $\eta(1 - 1/e)$.*

In the BASIC-dynamic setting, the vertices in C may also leave the graph and the algorithm is not necessarily irrevocable, i.e., the algorithm is allowed to remove some previously matched edges and add others in their place. We shall use $G_i = (S \sqcup C_i, E_i)$ to denote the graph in this setting right after the ith update step. Following the ranking algorithm, before the first update, we sample a random permutation σ over all n servers, which defines the *ranks* of servers. Our algorithm works against an oblivious adversary, which in particular means that the adversary is not aware of the random bits used for sampling σ. For each server $x \in S$, we keep a list of neighbors, $L(x)$, to which *end* the algorithm appends new neighboring clients. Whenever a new client u arrives, it is appended to the end of the list $L(x)$, for each of its neighboring servers $x \in N(u)$. To ensure that the entry of u in $L(x)$ can be deleted in constant time (if needed), we implement $L(x)$ as a doubly linked list, and keep with u a pointer to its entry in $L(x)$ (see lines 2 and 17 of Algorithm 3). Once it updates the lists $L(\cdot)$, the algorithm scans $N(u)$ to see if there is any free vertex $v \in N(u)$ to be matched with u. Among the free neighbors of u, if any, the one of lowest rank is chosen. This completes the description of the procedure for handling vertex insertion, which closely follows the ranking algorithm; the pseudocode of this procedure, HandleInsertion(u), is given in Algorithm 3.

Whenever a client u is deleted, it gets removed from the lists $L(x)$, for each of its neighbors $x \in N(u)$. If u was previously matched, say with server v, the algorithm removes edge (u, v) from the maintained matching $\mathcal{M}$ and tries to rematch vertex v. This is done by sequentially scanning the list $L(v)$ until an *appropriate* mate w is found; vertex w is appropriate if it

is free or it is matched with a vertex of higher rank than that of v. If w is free, we simply add (v, w) to the matching $\mathcal{M}$. Otherwise, we remove $(w, \text{mate}(w))$ from $\mathcal{M}$, we add (v, w) to $\mathcal{M}$, and handle the server $\text{mate}(w)$ recursively (in the same way as we handled v). The pseudocode of this procedure, HandleDeletion(u), is given in Algorithm 3.

Algorithm 3 BasicApproxMCM.

```
1:  procedure HandleDeletion(u)
2:      for x ∈ N(u) do remove u from L(x)
3:      if mate(u) ≠ ∅ then                                  ▷ let v = mate(u)
4:          delete (u, v) from M
5:          RematchServer(v)
6:  procedure RematchServer(v)
7:      while L(v) ≠ ∅ do
8:          let w be the first vertex in L(v)                ▷ also remove w from L(v)
9:          if w is free then
10:             add (v, w) to M
11:             break                                        ▷ exit the loop
12:         else if σ(v) < σ(mate(w)) then
13:             remove (w, mate(w)) from M; add (v, w) to M
14:             RematchServer(mate(w))
15:             break
16: procedure HandleInsertion(u)
17:     for w ∈ N(u) do add u to L(w)
18:     look for a free vertex v ∈ N(u) of lowest rank; if v is found, add (u, v) to M
```

We next argue, via Lemmas 22 and 23, that algorithm BasicApproxMCM proves Theorem 3: Lemma 22 implies the algorithm's correctness and Lemma 23 bounds its runtime.

▶ **Lemma 22.** *Algorithm BasicApproxMCM maintains an $(\frac{e}{e-1})$-approximation (in expectation) to the MCM.*

Proof. Recall that we used a random permutation σ for the ranks of vertices in S. Following the ith update step, we let π_i be the permutation of C_i representing the arrival order of the first i arriving clients; that is, $\pi_i(u) < \pi_i(v)$ for $u \neq v, u, v \in C_i$ iff u arrives before v. For a given permutation σ, arrival order π_i, and graph $G_i = (S \sqcup C_i, E_i)$, we let $\text{Ranking}(G_i, \sigma, \pi_i)$ be the matching returned by the ranking algorithm on G_i using ranks defined by σ and the arrival order π_i. We next assert that for each $i \geq 0$, following the ith update, the maintained matching $\mathcal{M}_i$ of G_i is equal to $\text{Ranking}(G_i, \sigma, \pi_i)$. By Theorem 21 and using the oblivious adversary assumption, if the maximum matching size of $G_i = (S \sqcup C_i, E_i)$ is μ, then the expected size of the matching provided by our dynamic algorithm is $\mu(1 - 1/e)$. That would complete the proof of Lemma 22.

The proof of this assertion proceeds in two stages. In the first stage we analyze a variant of BasicApproxMCM, which works in the same way as the original algorithm, except that no element is deleted from the lists $L(\cdot)$ as part of the execution of procedure RematchServer (cf. line 8 of Algorithm 3). In other words, procedure RematchServer(v) scans all elements of $L(v)$, namely all the neighboring clients of v in the current graph, in their arrival order. Denote this procedure by NaiveApproxMCM and the matching it maintains at the ith step by $\mathcal{M}'_i$. One can prove by induction on the update step that $\mathcal{M}'_i = \text{Ranking}(G_i, \sigma, \pi_i)$; the proof is rather straightforward and is omitted due to space constraints.

In the second stage we prove that $\mathcal{M}'_i = \mathcal{M}_i$. To this end we observe that the following invariant is maintained by algorithm BasicApproxMCM at all times.

▶ **Invariant 4.** *For any client c, if c is matched to a server v at update step $i \geq 0$, then at any later update step j $(j > i)$, client c is matched to a server of rank no higher than $\sigma(v)$.*

To see that Invariant 4 holds, note that the only place where the algorithm may rematch a client is in line 13 of procedure RematchServer. If that happens, the client gets matched to a server of rank lower than that of its previous mate.

Invariant 4 implies that any element that gets discarded from the list $L(v)$ in procedure RematchServer(v) without getting matched to v is always matched to a server of rank lower than v (until it possibly gets deleted from the graph), and thus the algorithm should never match v to any of the discarded elements. Hence, the executions of the two algorithms BasicApproxMCM and NaiveApproxMCM yield the same outcome, i.e., $\mathcal{M}'_i = \mathcal{M}_i$.

Summarizing, it follows that $\mathcal{M}_i = \mathcal{M}'_i = \text{Ranking}(G_i, \sigma, \pi_i)$, and we are done. ◀

▶ **Lemma 23.** *Algorithm BasicApproxMCM has a total runtime of $O(K)$.*

Proof. Procedure HandleInsertion(u) takes $O(\deg(u))$ time. Procedure HandleDeletion(u) also takes $O(\deg(u))$ time, except for the call to the recursive procedure RematchServer. Next, we analyze the runtime of procedure RematchServer. Note that the parameter v of RematchServer(v) is a server. We argue that the total time spent by procedure RematchServer(v) for a fixed vertex $v \in S$ is linear in the total number of inserted edges incident on v. Indeed, each call to procedure RematchServer involves a while loop (line 7); each iteration of the while loop takes constant time to examine a single neighbor w of v and do the required updates, and it also removes w from $L(v)$, so that w will never be examined again. It follows that the total time spent by RematchServer, over all servers, is upper bounded by $O(K)$. ◀

References

1. Ittai Abraham, David Durfee, Ioannis Koutis, Sebastian Krinninger, and Richard Peng. On fully dynamic graph sparsifiers. In Irit Dinur, editor, *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pages 335–344. IEEE Computer Society, 2016.
2. S. Baswana, M. Gupta, and S. Sen. Fully dynamic maximal matching in $O(\log n)$ update time. In *Proceedings of the 52nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, October 23-25, 2011*, pages 383–392, 2011. See also the *SICOMP'15* version, and the subsequent erratum.
3. Soheil Behnezhad, Jakub Lacki, and Vahab S. Mirrokni. Fully dynamic matching: Beating 2-approximation in Δ^{ε} update time. In Shuchi Chawla, editor, *Proc. 51th SODA*, pages 2492–2508, 2020.
4. Aaron Bernstein, Jacob Holm, and Eva Rotenberg. Online bipartite matching with amortized replacements. In *SODA*, pages 947–959. SIAM, 2018.
5. Aaron Bernstein, Tsvi Kopelowitz, Seth Pettie, Ely Porat, and Clifford Stein. Simultaneously load balancing for every p-norm, with reassignments. In *Proc. 8th ITCS*, pages 51:1–51:14, 2017.
6. Aaron Bernstein and Cliff Stein. Fully dynamic matching in bipartite graphs. In *Proc. 42nd ICALP*, pages 167–179, 2015.
7. Aaron Bernstein and Cliff Stein. Faster fully dynamic matchings with small approximation ratios. In *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 692–711, 2016.

8 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.

9 Sayan Bhattacharya and Peter Kiss. Deterministic rounding of dynamic fractional matchings. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *Proc. 48th ICALP*, volume 198, pages 27:1–27:14, 2021.

10 Benjamin E. Birnbaum and Claire Mathieu. On-line bipartite matching made simple. *SIGACT News*, 39(1):80–87, 2008.

11 Bartlomiej Bosek, Dariusz Leniowski, Piotr Sankowski, and Anna Zych. Online bipartite matching in offline time. In *Proc. 55th FOCS*, pages 384–393, 2014.

12 Bartlomiej Bosek, Dariusz Leniowski, Piotr Sankowski, and Anna Zych. Shortest augmenting paths for online matchings on trees. In *Proc. of 13th WAOA*, pages 59–71, 2015.

13 Bartłomiej Bosek, Dariusz Leniowski, Piotr Sankowski, and Anna Zych-Pawlewicz. A tight bound for shortest augmenting paths on trees. In *Proc. 13th LATIN*, pages 201–216, 2018.

14 Kamalika Chaudhuri, Constantinos Daskalakis, Robert D. Kleinberg, and Henry Lin. Online bipartite perfect matching with augmentations. In *Proc. of 28th INFOCOM*, pages 1044–1052, 2009.

15 Gagan Goel and Aranyak Mehta. Online budgeted matching in random input models with applications to adwords. In *SODA*, pages 982–991. SIAM, 2008.

16 Fabrizio Grandoni, Stefano Leonardi, Piotr Sankowski, Chris Schwiegelshohn, and Shay Solomon. $(1 + \epsilon)$-approximate incremental matching in constant deterministic amortized time. In Timothy M. Chan, editor, *Proc. 50th SODA*, pages 1886–1898, 2019.

17 Fabrizio Grandoni, Chris Schwiegelshohn, Shay Solomon, and Amitai Uzrad. Maintaining an EDCS in general graphs: Simpler, density-sensitive and with worst-case time bounds. *CoRR*, abs/2108.08825, 2021.

18 Edward F. Grove, Ming-Yang Kao, P. Krishnan, and Jeffrey Scott Vitter. Online perfect matching and mobile computing. In *Proc. of 45th Wads*, pages 194–205, 1995.

19 M. Gupta and R. Peng. Fully dynamic $(1 + \epsilon)$-approximate matchings. In *Proceedings of the 54th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2013, Berkeley, CA, USA, October 26-29, 2013*, pages 548–557, 2013.

20 Meng He, Ganggui Tang, and Norbert Zeh. Orienting dynamic graphs, with applications to maximal matchings and adjacency queries. In *ISAAC*, volume 8889 of *Lecture Notes in Computer Science*, pages 128–140. Springer, 2014.

21 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 47th Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 21–30, 2015.

22 John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973. doi:10.1137/0202019.

23 Zhiyi Huang, Ning Kang, Zhihao Gavin Tang, Xiaowei Wu, Yuhao Zhang, and Xue Zhu. How to match when all vertices arrive online. In *STOC*, pages 17–29. ACM, 2018.

24 Z. Ivković and E. L. Lloyd. Fully dynamic maintenance of vertex cover. In *Proceedings of the 19th International Workshop on Graph-Theoretic Concepts in Computer Science, WG 1993, Utrecht, The Netherlands, June 16-18, 1993*, pages 99–111, 1993.

25 Richard M. Karp, Umesh V. Vazirani, and Vijay V. Vazirani. An optimal algorithm for on-line bipartite matching. In *Proc. 22nd STOC*, pages 352–358, 1990.

26 Lazar Milenkovic and Shay Solomon. A unified sparsification approach for matching problems in graphs of bounded neighborhood independence. In Christian Scheideler and Michael Spear, editors, *Proc. of 32nd SPAA*, pages 395–406. ACM, 2020.

27 Ofer Neiman and Shay Solomon. Simple deterministic algorithms for fully dynamic maximal matching. In *Proceedings of the 45th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2013, Palo Alto, CA, USA, June 1-4, 2013*, pages 745–754, 2013.

28 D. Peleg and S. Solomon. Dynamic $(1 + \epsilon)$-approximate matchings: A density-sensitive approach. In *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, 2016.

29 Yongho Shin, Kangsan Kim, Seungmin Lee, and Hyung-Chan An. Online graph matching problems with a worst-case reassignment budget. *CoRR*, abs/2003.05175, 2020.

30 Noam Solomon and Shay Solomon. A generalized matching reconfiguration problem. In *ITCS*, volume 185 of *LIPIcs*, pages 57:1–57:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

31 S. Solomon. Fully dynamic maximal matching in constant update time. In *Proceedings of the 57th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2016, New Brunswick, NJ, USA, October 9-11, 2016*, pages 325–334, 2016.

32 Jan van den Brand, Danupon Nanongkai, and Thatchaphol Saranurak. Dynamic matrix inverse: Improved algorithms and matching conditional lower bounds. In *FOCS*, pages 456–480. IEEE Computer Society, 2019.

33 David Wajc. Rounding dynamic matchings against an adaptive adversary. In Konstantin Makarychev, Yury Makarychev, Madhur Tulsiani, Gautam Kamath, and Julia Chuzhoy, editors, *Proc. 52nd STOC*, pages 194–207, 2020.