Simple Deterministic Algorithms for Fully Dynamic Maximal Matching [*]

Ofer Neiman [†]   Shay Solomon [‡]

Abstract

A maximal matching can be maintained in fully dynamic (supporting both addition and deletion of edges) n-vertex graphs using a trivial deterministic algorithm with a worst-case update time of $O(n)$. No deterministic algorithm that outperforms the naïve $O(n)$ one was reported up to this date. The only progress in this direction is due to Ivković and Lloyd [14], who in 1993 devised a deterministic algorithm with an *amortized* update time of $O((n + m)^{\sqrt{2}/2})$, where m is the number of edges.

In this paper we show the first deterministic fully dynamic algorithm that outperforms the trivial one. Specifically, we provide a deterministic *worst-case* update time of $O(\sqrt{m})$. Moreover, our algorithm maintains a matching which is in fact a 3/2-approximate maximum cardinality matching (MCM). We remark that no fully dynamic algorithm for maintaining $(2 - \epsilon)$-approximate MCM improving upon the naïve $O(n)$ was known prior to this work, even allowing amortized time bounds and *randomization*.

For low arboricity graphs (e.g., planar graphs and graphs excluding fixed minors), we devise another simple deterministic algorithm with *sub-logarithmic update time*. Specifically, it maintains a fully dynamic maximal matching with amortized update time of $O(\log n / \log \log n)$. This result addresses an open question of Onak and Rubinfeld [19].

We also show a deterministic algorithm with optimal space usage, that for arbitrary graphs maintains a maximal matching in amortized $O(\sqrt{m})$ time, and uses only $O(n + m)$ space.

1 Introduction

In this paper we study deterministic algorithms for maximal matching in a dynamically changing graph. While graphs have been traditionally studied as static objects, in some of the modern applications of graph theory (e.g. communication and social networks, graphics and AI) graphs are subject to discrete changes. In the last few decades there has been a growing interest in algorithms and data structures for such dynamically changing graphs. In particular, several classical combinatorial problems, such as connectivity, min-cut, minimum spanning tree [11, 12, 21, 15, 5, 16, 9, 10, 4], have been considered in such a dynamic setting.

Our goal is to design a data structure that maintains a maximal matching, or an approximate maximum cardinality matching, in a fully dynamic graph. This dynamic setting allows both insertions and deletions of edges, while the vertex set is fixed. A standard assumption is that in each step a single edge is added to the graph or removed from it; such a step is called an *edge update* (or shortly, an *update*). Throughout the paper let n denote the number of vertices in the graph, and m the (current) number of edges.

Observe that a simple greedy algorithm computes a maximal matching in $O(m)$ time, so recomputing a maximal matching from scratch would cost $O(m)$ per update. It is not hard to see that one can also dynamically maintain a maximal matching with a worst-case update time of $O(n)$ (see Section 1.2). To the best of our knowledge there was no deterministic algorithm known that beats this naïve $O(n)$ time, even allowing amortized time bounds. Ivković and Lloyd [14] showed an algorithm with an *amortized* update time of $O((n+m)^{\frac{\sqrt{2}}{2}})$, which improves upon the $O(n)$ time algorithm only when $m = o(n^{\sqrt{2}})$.

Our results. We show a simple deterministic algorithm that for any dynamic graph maintains explicitly a maximal matching in $O(\sqrt{m})$ *worst-case* update time.[1] That is, if the current graph has m edges, then the next add or delete operation will be handled in $O(\sqrt{m})$ time. This improves upon the amortized bound of [14] for all values of m. Moreover, our algorithm is arguably simpler than that of [14].

It is well known that a maximal matching is in particular a 2-approximation for maximum cardinality matching (MCM).[2] Our algorithm has the additional property that there are no augmenting paths of length 3 at all times, which implies that the matching we maintain is in fact a 3/2-approximation for MCM. Remarkably, no algorithm with update time better than $O(n)$ was known for maintaining $(2-\epsilon)$-approximate MCM (for any $\epsilon > 0$), even allowing amortized time bounds and *randomization*. Our deterministic data structure also maintains an approximate *vertex cover*, as it is well known that the set of vertices participating in a maximal matching is in fact a 2-approximate vertex cover.

We also obtain improved bounds for low arboricity graphs, which are uniformly sparse graphs (see Definition 4.1). We show a simple deterministic algorithm that maintains a maximal matching in amortized $O\left(\frac{\log n}{\log((\log n)/c)} + c\right)$ time per update, provided that the dynamic graph has arboricity at most $c = o(\log n)$ at all times. When $c = \Omega(\log n)$ we obtain amortized $O(c)$ time. It is well known that the arboricity of any graph with m edges is at most $\sqrt{m}$. So this algorithm for bounded arboricity graphs gives rise to an even simpler $O(\sqrt{m})$ amortized time data structure for dynamic maximal matching in *arbitrary* graphs (but not a 3/2-approximate MCM). Moreover, we show that this algorithm can be implemented using *optimal* space $O(n+m)$.

At the "Open Problems" section of [19] the following question was raised: "Is there a *deterministic* data structure that achieves a constant approximation factor with polylogarithmic update time?" Our result gives a 2-approximation with sub-logarithmic update time for low arboricity graphs.

---

[1]We ignore additive terms that depend at most logarithmically on n.

[2]We say that a matching M is a *ρ-approximation for MCM* or *ρ-approximate MCM*, for an arbitrary parameter $\rho \geq 1$, if the cardinality of M is smaller than the MCM by at most a factor of ρ.

1.1   Related Work

Maintaining the maximum cardinality matching dynamically seems to be a difficult task. The state-of-the-art static algorithm by Micali and Vazirani from 1980 [17] takes $O(\sqrt{n} \cdot m)$ time, which suggests that obtaining a dynamic algorithm with $o(\sqrt{n})$ amortized time, would be considered a breakthrough for the static problem. Sankowski [20] showed a randomized algorithm with $O(n^{1.495})$ time per update that dynamically maintains an MCM. In a certain randomized model, where the edge being added or deleted is chosen at random, Alberts and Henzinger [2] showed that MCM can be maintained in amortized $O(n)$ update time.

Recently some very successful randomized algorithms were developed for maintaining an approximate MCM dynamically. Onak and Rubinfeld [19] showed a randomized $O(1)$-approximate MCM whose amortized time per update is $O(\log^2 n)$ with high probability. This was improved by Baswana, Gupta and Sen [6] to a randomized algorithm that maintains a maximal matching (and in particular 2-approximate MCM) in $O(\log n)$ expected amortized update time.

Concurrently and independently of our work, Anand [3] obtained similar results for deterministic fully dynamic approximate MCM. Specifically, Anand obtained a 3/2-approximate MCM in amortized $O(\sqrt{m})$ update time. (Recall that our bound $O(\sqrt{m})$ is worst-case rather than amortized.)

1.2   Overview of the Algorithm

We will only show how to get a somewhat weaker bound of $O(\sqrt{m+n})$ on the worst-case update time. The improvement to the bound $O(\sqrt{m})$ is a simple yet technical issue that we prefer to disregard for clarity purposes. Besides, it is quite natural to assume that $m = \Omega(n)$.

Let us recall how the naïve $O(n)$ time per update algorithm works. For every update, if an edge is added to the graph, check if it can be added to the matching. If a matched edge $\{u, v\}$ is deleted from the graph, examine all the neighbors of u and v to see if some edge $\{u, w\}$ or $\{v, w'\}$ (or both) can be added to the matching. It is not hard to verify that the resulting matching remains maximal. Now the question is, can we do anything better than scanning all the neighbors of a free vertex in order to find a new match for it? We believe that, in general, the answer is no.

The way our algorithm overcomes this obstacle is by ensuring that high degree vertices are *never* free. In particular, we maintain the following invariant: vertices of degree larger than $\sqrt{2(m+n)}$ are matched at all times. Then scanning all neighbors of a free vertex is not so expensive. Next we briefly explain how to maintain this invariant. When a high degree vertex u becomes free and cannot be matched (because all its neighbors are matched), we find a *surrogate* for it, that is, a vertex v' that is matched to a neighbor v of u, such that the degree of v' is at most $\sqrt{2m}$. Then we can match u to v, and the low degree vertex v' becomes free instead of u. We prove that such a vertex v' must exist, and show how to find one in $O(\sqrt{m+n})$ time.

One has to be careful when defining the invariant with respect to the number of edges, as this number changes with time. It is even possible that at some point many vertices violate the invariant simultaneously. The first attempt is to find a low degree surrogate for each of these vertices. Finding a surrogate, however, takes $O(\sqrt{m+n})$ time, and so we cannot handle many vertices at once. Instead, at each edge update we handle $O(1)$ "problematic" vertices, those that are getting close to violating the invariant. By handling the problematic vertices in decreasing order of degree (one at each edge update), we demonstrate that each problematic vertex will be handled long before it can violate the invariant.

In order to obtain a 3/2-approximate MCM, this approach does not suffice. When a vertex u becomes free but has no free neighbors, we may be forced to search every one of its $\sqrt{2m}$ neighbors v, who are matched to v', for a free neighbor of v'. To this end we use another idea: instead of searching for a free neighbor, we maintain a certain data structure for every vertex that holds information about all its free neighbors. This data structure will enable us to determine the existence of a free neighbor in $O(1)$ time, update a single neighbor (a free neighbor that becomes matched, and vice versa) in $O(1)$ time, and find a free neighbor in $O(\sqrt{n})$ time. Observe that whenever a vertex changes its status (free or matched) it

must inform all of its neighbors in order to update their free neighbors data structure. However, since we guarantee that high degree vertices never change their status (they are always matched), updating this data structure will only cost $O(\sqrt{m+n})$ time per update.

Low arboricity graphs. For graphs with bounded arboricity we use a result of Brodal and Fagerberg [7], who devised a fully dynamic algorithm for maintaining a bounded *edge orientation*. That is, assign a direction for each edge in the graph such that the out-degree of every vertex is bounded. Although the naïve $O(n)$ time algorithm mentioned above is in fact very efficient for bounded degree graphs (it runs in $O(d)$ time if d is the maximum degree), in bounded arboricity graphs the in-degree can be arbitrarily high. Our algorithm does the following: vertices have only *partial information* about their free neighbors. In particular, each vertex will send information about its status only along the out-going edges of the orientation. This greedy approach guarantees that each vertex will hold authentic information about all its (possibly many) in-coming neighbors at all times. Information about the few out-going neighbors will not be authentic, but can be verified on demand by scanning all of them.

2 Preliminaries

Let $G = (V, E)$ be an arbitrary graph. Any set $M \subseteq E$ of vertex-disjoint edges is called a *matching*. A matching of maximum cardinality in G is called a *maximum (cardinality) matching* (or shortly, *MCM*), and a matching that is maximal under inclusion is called a *maximal matching*. For any parameter $t \geq 1$, a matching that contains at least $1/t$ fraction of the edges in an MCM is called a *t-approximate MCM*. It is easy to see that any maximal matching is a 2-approximate MCM.

A vertex is called *matched* if it is incident on some edge of M. Otherwise it is *free*. For any edge $\{u, v\} \in M$, we say that u (respectively, v) is the *mate* of v (resp., u). An *alternating path* is a path whose edges alternate between M and $E \setminus M$. An *augmenting path* is an alternating path that starts and ends at different free vertices. It is well known [13] that any matching without augmenting paths of length at most $2k - 3$ is a $(k/(k-1))$-approximate MCM. In particular, if there are no augmenting paths of length at most 3, we get 3/2-approximate MCM.

3 General Graphs

In this section we present a fully dynamic algorithm for maintaining a maximal matching that is also a 3/2-approximate MCM. Our data structure is deterministic and requires a worst-case update time of $O(\sqrt{n+m})$, for general n-vertex graphs with m edges. Denote the (static) vertex set of the graph by $V = \{1, 2 \ldots, n\}$, and assume for simplicity that $\sqrt{n}$ is an integer. Let $\mathcal{G} = (G_0, G_1, \ldots)$ be the given sequence of graphs, we assume that the initial graph G_0 is empty, and each graph G_i is obtained from the previous graph G_{i-1} by either adding or deleting a single edge. For each time step i, write $G_i = (V, E_i)$, $m_i = |E_i|$. We maintain the number of edges in the current graph $G = (V, E)$ in the variable m.

3.1 Data Structures

The algorithm will maintain the following data structures.

- The current matching M is stored in an AVL tree. It supports `insert` and `delete` in $O(\log n)$ time. Every vertex $v \in V$ holds a `mate(v)` that returns its current mate in the matching (or $\perp$ if v is free).

- For each vertex $v \in V$ an AVL tree $N(v)$ that stores its current neighbors, and a variable $\deg(v)$ for its degree. It supports `insert` and `delete` in $O(\log n)$ time, and extracting arbitrary r neighbors in $O(r)$ time (by traversing the tree).

- For each vertex $v \in V$ a data structure $F(v)$ that holds its free neighbors, and supports the following operations: `insert` and `delete` in $O(1)$ time, `has-free`(v) that returns TRUE if v has a free neighbor in $O(1)$ time, and `get-free`(v) that returns an arbitrary free neighbor of v in $O(\sqrt{n})$ time.

  In order to implement $F(v)$ for each vertex $v \in V$, we use a boolean array of size n indicating the current free neighbors, a counter array of size $\sqrt{n}$ that has in position j the number of free neighbors in the range $[\sqrt{n} \cdot j + 1, \sqrt{n}(j+1)]$, and a variable for the total number of free neighbors. Now `insert`, `delete` and `has-free`(v) are clearly $O(1)$ operations, and in order to implement `get-free`(v), we can scan in $\sqrt{n}$ time the counter array for a positive entry, and check the appropriate range in the boolean array.

- A maximum heap F_{max} of all free vertices indexed by their degree. It supports `insert`, `delete`, `update-key`, and `find-max` in $O(\log n)$ time.

`handle-addition`$(\{u, v\})$:

1. Update $N(u)$, $N(v)$, $\deg(u)$, $\deg(v)$ and F_{max};

2. If both u, v are free, `match`(u, v);

3. If only u is free:

   (a) Set $v' = $ `mate`(v);

   (b) Remove u from $F(v')$;

   (c) If `has-free`(v'):

      i. Call `match`(u, v); `match`$(v', $`get-free`$(v'))$;

      ii. Remove $\{v, v'\}$ from M;

   (d) Else, for all $w \in N(u)$, add u to $F(w)$;

4. If only v is free, return to 3 with roles of u, v switched;

Figure 1: Handle an edge addition.

`match`(u, v):

1. Add $\{u, v\}$ to M;

2. Remove u, v from F_{max};

3. For $w \in \{u, v\}$:

   (a) If `mate`$(w) = \perp$, remove w from $F(x)$
       for all $x \in N(w)$;

4. Set `mate`$(u) = v$; `mate`$(v) = u$;

Figure 2: Add an edge to the matching.

```
aug-path(u):

    1. For all w ∈ N(u):

        (a) Let w′ = mate(w);
        (b) If has-free(w′):

                i. Let x = get-free(w′);
               ii. break;

    2. If a free x was found:

        (a) Call match(u, w); match(w′, x);
        (b) Remove {w, w′} from M;

    3. Else, if no augmenting path was found:

        (a) For all w ∈ N(u) add u to F(w);
        (b) Add u to F_max; Set mate(u) = ⊥;
```

Figure 3: Finding a length 3 augmenting path starting at u and adding it to the matching. It is assumed that $\deg(u) \leq \sqrt{2n + 2m}$ and that u has no free neighbor.

```
surrogate(u):

    1. For all w ∈ N(u):

        (a) Let w′ = mate(w);
        (b) If deg(w′) ≤ √2m, break;

    2. Remove {w, w′} from M;

    3. Call match(u, w);

    4. Return w′;
```

Figure 4: Finding a surrogate low degree vertex for the vertex u. It is assumed that $\deg(u) > \sqrt{2m}$ and that u has no free neighbor.

3.2 Algorithm

At the outset the graph is empty, and we perform an initialization phase for our data structures. Next, the algorithm is carried out in rounds: In each round $i = 1, 2, \ldots$, a single edge e_i is either added to the graph or deleted from it, and the algorithm will update the data structure in $O(\sqrt{n + m})$ time to preserve the following invariants at the end of each step i.

Invariant 1 *All free vertices have degree at most $\sqrt{2n + 2m}$.*

Invariant 2 *All vertices that became free in round i have degree at most $\sqrt{2m}$.*

Invariant 3 *The matching M maintained by the algorithm is maximal. Moreover, there are no augmenting paths of length 3 (with respect to M).*

The invariants clearly hold before the first round starts and the edge e_1 is handled. Fix a time step i. We will now describe a single round of the algorithm, which handles an edge e_i that is added to the graph or deleted from it.

5

```
handle-deletion({u, v}):

   1. Update N(u), N(v), deg(u), deg(v) and F_max;

   2. If {u, v} ∉ M:

      (a) If u is free, remove it from F(v); If v is free, remove it from F(u);

   3. Else, if {u, v} ∈ M:

      (a) Remove {u, v} from M;
      (b) For z ∈ {u, v}:
            i. If has-free(z), call match(get-free(z), z);
           ii. Else, if there is no free neighbor for z:
                A. If deg(z) > √2m, let z = surrogate(z); Return to 3(b)i.
                B. Else, call aug-path(z);
```

Figure 5: Handle an edge deletion.

3.2.1 Edge Addition

We start with the case where the edge $e_i = \{u, v\}$ is *added* to the graph, see Figure 1. First update the relevant data structures: $N(u), N(v), \deg(u), \deg(v)$ and the keys of u, v in F_{max} (if needed), which takes $O(\log n)$ time. Next, we distinguish between four cases.

Case 1: Both u and v are matched. In this case there is nothing to do.

Case 2: Both u and v are free. In this case we match u with v, see Figure 2. This involves updating the data structures M, F_{max} and removing u, v from the free neighbor data structures $F(x)$ for all neighbors x of u, v. By Invariant 1 both $\deg(u)$ and $\deg(v)$ are at most $\sqrt{2n + 2m} + 1$, so this takes $O(\sqrt{n + m})$ time. Observe that adding the edge $\{u, v\}$ to M does not create any new augmenting paths of length at most 3, because by maximality of M, both u, v could not have had a free neighbor, and so Invariant 3 is preserved.

Case 3: u is free and v is matched. In this case, adding the edge $\{u, v\}$ to the graph may give rise to new augmenting paths of length 3 that include $\{u, v\}$. Specifically, such a path may exist iff $v' = \text{mate}(v)$ has a free neighbor $w \neq u$. We determine if v' has a free neighbor $w \neq u$ or not in the following way: First, we remove u from $F(v')$ (we will "undo" this before the round is over). Next, we check if $v' = \text{mate}(v)$ has a free neighbor w (note that $w \neq u$). If we can find one, then we add $\{u, v\}$ and $\{v', w\}$ to M, and remove $\{v, v'\}$ from M. Observe that when adding the edges we update $F(x)$ *only* for the vertices x that are neighbors of u and w (v and v' were already matched), by Invariant 1 both $\deg(u)$ and $\deg(w)$ are at most $\sqrt{2n + 2m} + 1$, so this takes $O(\sqrt{n + m})$ time. If we cannot find such a free neighbor w, then u will remain free, and is added to the free neighbor data structures of all its neighbors (in particular to $F(v)$ and if needed to $F(v')$ as well).

Case 4: u is matched and v is free. This case is symmetric to case 3.

It is easy to see that handling the addition of $\{u, v\}$ in each of the four cases above: (1) requires an overall update time of $O(\sqrt{n + m})$, and (2) preserves both Invariants 2 and 3. Invariant 1 will be handled separately.

3.2.2 Edge Deletion

We proceed to the case where the edge $e_i = \{u, v\}$ is *deleted* from the graph, see Figure 5. First update the relevant data structures: $N(u), N(v), \deg(u), \deg(v)$ and F_{max} (if needed), which takes $O(\log n)$ time. There are two cases to consider.

Case 1: $\{u, v\} \notin M$. In this case, the only remaining thing to do is to remove u from $F(v)$ if u is free, and remove v from $F(u)$ if v is free.

Case 2: $\{u, v\} \in M$. Here we delete the edge $\{u, v\}$ from M (we do not update the status of u and v from matched to free yet for technical reasons that will become clear soon, but we will make sure to update u and v with their correct status before the end of this round). Deleting $\{u, v\}$ from M may give rise to new augmenting paths of length at most 3 that start at one of the endpoints of this edge. Next, we show how to handle u. The other endpoint v should be handled in the same way.

If u has a free neighbor w, we add $\{u, w\}$ to M by calling $\mathtt{match}(u, w)$. Observe that we *do not* update $F(x)$ for the neighbors $x \in N(u)$ (as $\mathtt{mate}(u) = v$ before $\mathtt{match}$ is called). Since w was free, Invariant 1 suggests that $\deg(w) \leq \sqrt{2n + 2(m + 1)}$, so this will take only $O(\sqrt{n + m})$ time. We henceforth assume that u has no free neighbor, and consider two cases.

Case 2.a: $\deg(u) \leq \sqrt{2m}$. In this case we can allow u to become free, but still must search for an augmenting path of length 3 starting at u by calling $\mathtt{aug\text{-}path}(u)$ (see Figure 3). An augmenting path exists iff some neighbor w of u is matched to w' and w' has a free neighbor $x \neq u$. For each such neighbor $w \in N(u)$ we can in $O(1)$ time detect if its mate w' (recall that w must be matched) has a free neighbor. Only if we find such a w' we do the $O(\sqrt{n})$ operation of actually extracting this free neighbor x, and then stop the search (we are guaranteed that $x \neq u$, because we have not changed the status of u to free just yet). If such an x was found, we change M by adding $\{u, w\}$ and $\{w', x\}$ instead of $\{w, w'\}$. Observe that we update $F(y)$ only for neighbors of x (as u, w, w' are recorded as matched), which takes $O(\sqrt{n + m})$ time by Invariant 1. If no augmenting path was found, we declare u as a new free vertex (which complies with Invariant 2), and update $F(w)$ for all its neighbors w. A delicate matter that needs attention is the following: if u is the first among $\{u, v\}$ that is handled, v is still recorded as matched, so we will not be able to find an augmenting path of length 3 that starts at u and ends in v. However, this path can be detected once we are done with u, set its status to free, and handle v.

Case 2.b: $\deg(u) > \sqrt{2m}$. Note that u cannot become free because its degree is too high, alas it has no free neighbor. In order to keep u matched, we run $\mathtt{surrogate}(u)$ to find a surrogate s_u for u that may become free instead of u (see Figure 4). Even though $\deg(u)$ is high, we claim that after inspecting $\sqrt{2m}$ of the neighbors $w \in N(u)$, we must have found one with degree at most $\sqrt{2m}$, and then stop the scan. Indeed, otherwise the sum of degrees in the graph would be more than $\sqrt{2m} \cdot \sqrt{2m} = 2m$ (note that $\mathtt{mate}(w)$ are distinct for different w), which is impossible. Since the surrogate s_u has degree at most $\sqrt{2m}$, changing its status to free (if needed) would not violate Invariant 2. Next, handle s_u as u is handled above just before *Case 2.a* (that is, find a free neighbor of s_u or an augmenting path of length 3). Note that handling s_u cannot bring us to *case 2.b*, so there is no risk of an infinite loop. We claim that no augmenting path of length 3 can remain, because any augmenting path emanating from s_u is detected, and the edge $\{u, w\}$ that is added to M in $\mathtt{surrogate}(u)$ cannot be a part of an augmenting path because u has no free neighbors[3].

It is easy to see that handling the deletion of $\{u, v\}$ in each of the two cases above: (1) requires an overall update time of $O(\sqrt{n + m})$, and (2) preserves both Invariants 2 and 3.

3.2.3 Bounding the Degree of Free Vertices

Here we show how to preserve Invariant 1, that free vertices have bounded degrees, which is a key property in our algorithm. The general idea is to identify some *problematic* vertices at the end of each round, and to *correct* them. We say that a vertex is *problematic* if (i) it is free, and (ii) its degree exceeds $\sqrt{2m}$. Such a problematic vertex x is corrected by applying *case 2.b* above on x. That is, find a surrogate s_x that may become free instead of x, with $\deg(s_x) \leq \sqrt{2m}$. In order to preserve Invariant 3, we then find an augmenting path of length at most 3 emanating from s_x if one exists.

Since each correction takes $O(\sqrt{n + m})$ time, we can only afford to correct $O(1)$ problematic vertices at the end of each round. It turns out that correcting the following three vertices (if they are problematic)

---
[3]Again we exclude augmenting paths ending at v, those will be detected later.

suffices: first the two endpoints u and v of the handled edge e_i, and afterwards a free vertex x with maximal degree (such a vertex can be extracted from the heap F_{max} in $O(\log n)$ time).

The next lemma implies that Invariant 1 is preserved.

Lemma 3.1 *At the end of each round i, for any free vertex x, $\deg(x) \leq \sqrt{2n + 2m}$.*

Proof: Recall that $G_i = (V, E_i)$ denotes the i-th graph in the graph sequence $\mathcal{G}$, and $m_i = |E_i|$ stands for the number of edges in it. First observe that the degree of a problematic vertex x cannot change as long as it is problematic. This is because any change to the degree would mean that we added or deleted an edge touching x, and so must have corrected it.

Seeking contradiction, assume that at round t the vertex x is free and $\deg(x) > \sqrt{2n + 2m_t}$. Let $k < t$ be such that at round k, $\deg(x) \leq \sqrt{2m_k}$ and in all rounds $k < j \leq t$ we have that $\deg(x) > \sqrt{2m_j}$. Since x is problematic in all the rounds from $k + 1$ to t, its degree does not change, and it follows that $\sqrt{2m_k} > \sqrt{2n + 2m_t}$, or $m_k - m_t > n$. Let $k \leq q < t$ be the minimal round such that the number of edges in every round $q + 1, \ldots, t$ is less than m_k, observe that $n < t - q$ because there must have been more that n deletions of edges.

We claim that x must be corrected in one of these n rounds. To prove this, it suffices to show that every vertex that becomes problematic after round q will have smaller degree than $\deg(x)$. Once it becomes problematic its degree cannot change, so in F_{max} the vertex x will be handled before all the "new" problematic vertices. Since we handle one vertex from F_{max} at each round, and there are more than n rounds from q to t, it must be that x is handled in one of them. Suppose vertex w becomes problematic at the conclusion of round $q < r \leq t$. There could be three reasons for this: First, if the degree of w changed in round r, then actually it must have been corrected (recall that we correct both endpoints of the new edge). Second, if w became a new free vertex, then by Invariant 2 $\deg(w) \leq \sqrt{2m_r}$, and as x is problematic in round r, $\deg(x) > \sqrt{2m_r}$. The last case is that w was already free, and the number of edges decreased. But then in round $r - 1$, x is problematic and w is not, thus $\deg(w) \leq \sqrt{2m_{r-1}} < \deg(x)$. We conclude that x is indeed corrected before round t ends, a contradiction. $\qquad\square$

We have shown the following.

Theorem 3.2 *Starting with the empty graph on n vertices, a maximal matching in the graph which is also a 3/2-approximate MCM can be maintained in time $O(\sqrt{n + m})$ per edge update, where m is the (current) number of edges.*

4 Low Arboricity Graphs

In this section we consider graphs with arboricity bounded by c.

Definition 4.1 *A graph $G = (V, E)$ has arboricity c if*

$$c = \max_{U \subseteq V} \left\lceil \frac{|E(U)|}{|U| - 1} \right\rceil,$$

where $E(U) = \{\{u, v\} \in E \mid u, v \in U\}$.

The family of graphs with bounded arboricity is the family of uniformly *sparse* graphs. In particular, it contains planar and bounded genus graphs, bounded tree-width graphs and in general all graphs excluding fixed minors.

A Δ-orientation of an undirected graph $G = (V, E)$ is a directed graph $H = (V, A)$ where A contains the same edges as in E (each edge is given a direction), so that the out-degree of every vertex in H is at most Δ. A well-known theorem of Nash and Williams [18] asserts that a graph has arboricity at most c iff E can be partitioned to $E_1, \ldots, E_c$ such that (V, E_i) is a forest for all $1 \leq i \leq c$. This suggests that

one can select an arbitrary root for all trees in the forests, and direct all edges towards the root. The out-degree of any vertex in each forest (V, E_i) is at most 1, so G has a c-orientation.

Consider a sequence of graphs $\mathcal{G} = (G_0, G_1, \ldots, G_k)$ on the vertex set V with $|V| = n$. We say that $\mathcal{G}$ has arboricity c if: G_0 is the empty graph and, for all $1 \leq i \leq k$, G_i is obtained from G_{i-1} by adding or deleting an edge, and all graphs G_i have arboricity at most c. We say that an algorithm maintains a Δ-orientation for $\mathcal{G}$ with amortized time T if it provides a Δ-orientation H_i for every G_i, and the total number of edge re-orientations is $k \cdot T$. Brodal and Fagerberg [7] proved the following theorem.

Theorem 4.2 (Brodal and Fagerberg [7]) *For any sequence of graphs $\mathcal{G}$ with arboricity c, and any $\Delta \geq 2\delta > 2c$ there is an algorithm that maintains a Δ-orientation for $\mathcal{G}$ with amortized time $T = O(\Delta + \frac{\Delta+1}{\Delta+1-2\delta} \cdot \log_{\delta/c} n)$.*

4.1   Reduction from Matchings to Orientations

Now we explain how to use an algorithm $\mathcal{A}$ that maintains a Δ-orientation in amortized time T, in order to obtain an algorithm that maintains a maximal matching in amortized time $O(\Delta + T)$. The idea behind the algorithm is the following: every vertex is responsible to notify about its state - free or matched - all the vertices it is pointing to (there are at most Δ such vertices). In other words, each vertex knows exactly who is free among the (possibly many) vertices pointing towards it, but knows nothing of the (at most Δ) vertices it is pointing to. This partial information enables vertices to pay only $O(\Delta)$ time in order to retrieve all information about their neighbors, and also they can perform the necessary status updates in $O(\Delta)$ time.

Consider a sequence of graphs $\mathcal{G}$ with arboricity c. For every graph $G_i \in \mathcal{G}$ we have an orientation H_i given by algorithm $\mathcal{A}$. For a vertex $u \in V$ denote by $N_i(u)$ the set of neighbors of u in G_i, and let $D_i(u) \subseteq N_i(u)$ be the set of vertices such that the edge (u, v) is directed out of u in the current orientation induced by H_i. Observe that $|D_i(u)| \leq \Delta$ for all $u \in V$ and $0 \leq i \leq k$. We will maintain the following data structures:

- A data structure $D(u)$ holding all the vertices of $D_i(u)$, for all $u \in V$.

- A data structure $F(u)$ holding all the *free* vertices $v \in N_i(u) \setminus D_i(u)$, for all $u \in V$.

- A data structure M containing all the matched edges, and a value for every vertex indicating whether it is free or matched.

Each of these data structures will be implemented using an array of size n augmented with a (single) linked list. For $D(u)$ and $F(u)$ the array will be boolean, while for M entry i will include the mate of node i in the matching, or $\perp$ if node i is free. Insertions can easily be done in $O(1)$ time, however deletions are done only in the array - the linked lists may contain extra elements that are in fact deleted. Whenever an extraction is needed, we go over the linked list starting at its head, and verify every element we encounter against the array. If an extraction took r verifications until a valid element was found, then we have deleted $r - 1$ elements from the list and the cost is divided among these $r - 1$ delete operations. We conclude that all these data structures support insertion, deletion and extraction in amortized $O(1)$ time. Denote by D (respectively F) as the union of the data structures $D(u)$ (resp. $F(u)$) for all $u \in V$.

We now give an overview of the algorithm that maintains a maximal matching in $\mathcal{G}$.

- **Orientation:** At every step we run algorithm $\mathcal{A}$ that preserves a Δ-orientation for the current graph. We then update the data structures F and D as described in Figure 6 so that they will be consistent with the current orientation. If there are t edge re-orientations then this takes $O(t)$ time.

- **Insertion:** (see Figure 7). Upon edge $\{u, v\}$ insertion, we check if we can add the edge to the matching M, and if so remove u and v from the free neighbors data structure $F(w)$ for every point w in $D(u)$ and $D(v)$. As all out-degrees are at most Δ, this takes $O(\Delta)$ time.

- **Deletion:** (see Figure 8). Upon deletion of a non-matched edge $\{u, v\}$, nothing more needs to be done. The interesting case is deleting a matched edge $\{u, v\}$: here we try to find new match for u (respectively v) by going over $F(u)$ and $D(u)$ (respectively $F(v)$ and $D(v)$). Observe that $D(u)$ may contain both free and matched vertices, so we must go over all of its elements, which takes $O(\Delta)$ time. Although $F(u)$ may be very large, we are guaranteed that all vertices in it are free, so we just need to extract one of them (or tell that $F(u)$ is empty) which takes amortized $O(1)$ time.

We conclude that the following result holds.

Theorem 4.3 *For every sequence of graphs $\mathcal{G}$ on n vertices with arboricity c, and for any $\Delta \geq 6c$, there is an algorithm that maintains a maximal matching in amortized time $O(\Delta + \log_{\Delta/c} n)$.*

Proof: By Theorem 4.2 in the particular case $\Delta = 3\delta \geq 6c$, we can maintain an orientation with out-degree at most Δ in amortized time $O(\Delta + \log_{\Delta/c} n)$. Our algorithm in addition performs at most $O(\Delta + t_i)$ operations, where t_i is the number of edge re-orientations at step i. We conclude that the amortized time is $O(\Delta + \log_{\Delta/c} n)$. It can be easily verified that the data structures F and D are consistent throughout the execution of the algorithm, and that M is indeed a maximal matching. $\qquad \square$

Corollary 4.4 *If for some $k \geq 2$ we have $c = \frac{\log n}{k}$, then choosing $\Delta = 6c + \frac{\log n}{\log k}$ will give amortized time $O\left(\Delta + \frac{\log n}{\log(\Delta/c)}\right) = O(\Delta)$; in particular, we will get amortized time $O(\log n / \log \log n)$ when $c \leq \log^{1-\epsilon} n$ (for any constant $\epsilon > 0$). For any $c = \Omega(\log n)$, choosing $\Delta = 6c$ will give amortized time $O(c)$.*

---

update-orientation($\{u, v\}$):

1. Run algorithm $\mathcal{A}$:

2. If edge (u, v) was inserted and directed from u to v:

   (a) Add v to $D(u)$; If u is free, add u to $F(v)$;

3. Otherwise, if edge (u, v) was deleted:

   (a) Remove v from $D(u)$; If u is free, remove u from $F(v)$;

4. For every edge (x, y) that was re-oriented to (y, x):

   (a) Delete y from $D(x)$, and add x to $D(y)$;

   (b) If x is free, delete x from $F(y)$; If y is free, add y to $F(x)$;

Figure 6: Updating the orientation for bounded arboricity graphs

5   Maximal Matching Using Optimal Space

In this section we show that the algorithm of Section 4 for bounded arboricity graphs can be implemented using only $O(n + m)$ space. Using the fact that any graph on m edges has arboricity at most $\sqrt{m}$ (see [8]), we conclude that the amortized update time is $O(\sqrt{m})$ even for *arbitrary* graphs.[4]

---

[4] Again we ignore additive terms that depend at most logarithmically on n.

```
insert {u,v}:

   1. Run update-orientation({u,v});

   2. If both u,v are free:

      (a) Add {u,v} to M;
      (b) For all w ∈ D(u) ∪ D(v), remove u and v from F(w);
```

Figure 7: Edge insertion for bounded arboricity graphs

```
delete {u,v}:

   1. Run update-orientation({u,v});

   2. If {u,v} ∈ M:

      (a) remove {u,v} from M;
      (b) For all w ∈ D(u), add u to F(w); For all w ∈ D(v), add v to F(w);
      (c) For w ∈ {u,v}:
             i. If F(w) ≠ ∅, find some x ∈ F(w) and add {x,w} to M;
            ii. Otherwise, for all x ∈ D(w):
                   A. If x is free, then add {x,w} to M and break;
           iii. If {x,w} was added to M, then for all y ∈ D(w)∪D(x), remove w and x from F(y);
```

Figure 8: Edge deletion for bounded arboricity graphs

We remark that using dynamic hash tables it would have been easy to obtain space $O(n + m)$, but we desire a fully deterministic algorithm. Recall that c is the maximum arboricity of the graph, and let $\Delta = 5c$ be the maximum allowed out-degree in the graph.

Data Structures: A data structure M containing all the matched edges (and a value for each vertex indicating whether it is free or matched) will be maintained just as in Section 4. In addition, we will maintain for each vertex u its current neighbors $N(u)$, its outgoing neighbors $D(u)$ and its free incoming neighbors $F(u)$ as linked lists (and maintain their sizes as well). We will also maintain the degree $\deg(u)$ and a variable indicating whether u is free or matched. The lists $N(u)$ and $F(u)$ will not be *authentic* at all times. A list is not authentic if it may contain redundant elements (e.g. the list $N(u)$ is not authentic if it may contain vertices which are not neighbors of u). In contrast, the $D(u)$ lists will always be authentic. In fact, the Brodal-Fagerberg algorithm [7] that we use maintains these lists explicitly in $O(n + m)$ space, so we may assume that $D(u)$ is always up to date. Also, $\deg(u)$ will contain the current degree of u, and thus it may be smaller than $|N(u)|$ at some stages throughout the execution of the algorithm. In order to control the total space used, we will guarantee that $|N(u)|$ (respectively, $|F(u)|$) never exceeds $\deg(u)$ by more than a factor of 2 (resp., 3). We will also use a "smart" boolean array of size n (an array that allows to reset all the elements to 0 in $O(1)$ time, see, e.g., [1]) for authentication of the lists. By authentication we mean removing the redundant elements that may be present in the list. To meet the desired space requirement, the same smart array will be re-used for all lists.

Handling the $N(u)$ lists: First note that these lists are never used by the algorithm, their sole purpose is to assist in controlling the size of the $F(u)$ lists. Whenever an edge $\{u,v\}$ is added to the graph, we simply add u to $N(v)$ and v to $N(u)$ in $O(1)$ time. However, upon deletion of the edge $\{u,v\}$, we put 4Δ tokens: Δ on each of the lists $N(u)$ and $N(v)$ and Δ tokens on each list $F(u), F(v)$. Each token will be used to pay for operations that take $O(1)$ time. As there is at most one edge deletion per round, we can

afford to spend so many tokens. For every change to $N(u)$, we check that $|N(u)| < 2\deg(u)$. If it is not, we authenticate it in the following manner. Iterate over the list $N(u)$, and for every element $w \in N(u)$:

- Search $D(u)$ for w and search $D(w)$ for u, if none of them was found, remove w from $N(u)$.

Claim 5.1 *The time cost of all authentications of the list $N(u)$, can be paid for using its tokens.*

Proof: If at some point in time we need to authenticate $N(u)$, then it is the case that $|N(u)| = 2\deg(u)$. The authentication process searches for the $2\deg(u)$ elements $w \in N(u)$, and for each perform a search in $D(u)$ and $D(w)$, to check if indeed w is still a neighbor of u. As $|D(u)|, |D(w)| \le \Delta$, the cost of this search over all $w \in N(u)$ is at most $2\Delta \cdot |N(u)| = 4\Delta\deg(u)$. Finally, note that $N(u)$ will be of size exactly $\deg(u)$ after the authentication, so $\deg(u)$ edges touching u were removed from the graph since the last authentication on $N(u)$. Thus we can use the Δ tokens put on $N(u)$ from each of these deleted edges, to cover the total cost of $4\Delta\deg(u)$.

Handling the $F(u)$ lists: For $F(u)$, we will keep adding elements in $O(1)$ time according to the algorithm, but deletions are postponed. In order to be useful for the algorithm, we have to show how to extract an authentic element (a free neighbor of u) from $F(u)$ in amortized $O(\Delta)$ time. Observe that the algorithm requires at most two extractions of authentic elements at every round (when edge $\{u, v\}$ is deleted, extractions are required from $F(u)$ and $F(v)$), and thus we can afford to spend $O(\Delta)$ time for each. We must also guarantee that the size of $F(u)$ is always $O(\deg(u))$, in order to comply with the space requirement.

Whenever a delete operation is performed on $F(u)$ in the algorithm, we will place 2 tokens on it, rather than performing the deletion. Spending these tokens for each delete operation on $F(u)$ will increase the total amortized time by at most a constant factor. One of the tokens will be used in extracting an authenticate element of $F(u)$ in Claim 5.2, while the other will be spent for controlling the size of $F(u)$ in Claim 5.3. Similarly to $N(u)$, $F(u)$ may contain non-authentic elements (e.g. a matched neighbor, or even a vertex which is no longer a neighbor of u).

Claim 5.2 *The cost of extracting a free neighbor of u from $F(u)$ can be paid for using tokens plus $O(\Delta)$.*

Proof: Whenever an element of $F(u)$ is required in the algorithm, we do the following. Reset the smart array, and update it to contain the elements of $D(u)$ in $O(\Delta)$ time. Note that the elements of $D(u)$ are always authentic. Then for each $w \in F(u)$:

- If w is matched, remove w from $F(u)$, using one of the tokens created by the deletion of w from $F(u)$ to pay for this operation.

- Otherwise, if w is free: if $w \in D(u)$, remove w from $F(u)$. Checking if $w \in D(u)$ takes $O(1)$ time using the array, so we can pay for it with one of the tokens created by the deletion of w from $F(u)$.

- Otherwise, if w is free and $w \notin D(u)$, we search for u in $D(w)$:
  - If u is found, then w is an authentic free incoming neighbor of u. We spent Δ time searching for u in $D(w)$, but can terminate the search.
  - If u was not found, then the edge $\{u, w\}$ must have been deleted. Thus we remove w from $F(u)$. We spent Δ time, which can be paid for by Δ of the tokens placed on $F(u)$ by the deleted edge $\{u, w\}$.

Each of the non-authentic elements we encountered had enough tokens for executing its removal, thus the actual cost of the search is only Δ.

Claim 5.3 *At all times, $|F(u)| \le 3\deg(u)$.*

Proof: Whenever an element is added or removed from $F(u)$, we check if $|F(u)| \geq 3 \deg(u)$, and if so start a (partial) authentication process that will use tokens in order to reduce its size down to at most $2 \deg(u)$. We reset the smart array, and initialize it with $N(u)$. As $|N(u)| < 2 \deg(u)$, this will take at most $O(\deg(u))$ time. Recall that it could be that $N(u)$ is not authentic. Now iterate over $F(u)$, and for each element $w \in F(u)$:

- If w is matched, remove w from $F(u)$.

- Otherwise, if w is free but $w \notin N(u)$, remove w from $F(u)$. (Checking this takes $O(1)$ time using the array).

This (partial) authentication process may make one-sided mistakes: elements that should have been deleted from $F(u)$ may still remain, but no element will be deleted from $F(u)$ un-necessarily, since $N(u)$ contains all the current neighbors of u. Observe that since $|F(u)| \geq 3 \deg(u)$ but $|N(u)| \leq 2 \deg(u)$, at least $\deg(u)$ elements must have been removed from $F(u)$ in the process. Each removed element has a token placed for its removal from $F(u)$, so we have the $\deg(u)$ tokens to pay for this authentication process.

To conclude, we have shown that the algorithm from Section 4 can be implemented in optimal space $O(n + m)$, without increasing the amortized update time by more than a constant factor. We can also obtain a result for arbitrary graphs, captured by the following theorem.

Theorem 5.4 *Starting with the empty graph on n vertices, a maximal matching can be maintained in amortized time $O(\sqrt{m})$ using space $O(n + m)$, where m is the (current) number of edges.*

Proof: Recall that the arboricity of any graph with m edges is bounded by $\sqrt{m}$. We will use the orientation algorithm, and at the beginning of every stage, we set a bound on the maximum arboricity to be $c = \sqrt{2m}$. Whenever m changes by a factor of 2 (until that moment the arboricity of the graph is bounded by the parameter c that was fixed at the beginning of the stage), we end the current stage, reset the value of c, and recompute the orientation (this can be done in $O(m)$ time–which increases the amortized time by only a constant, and within $O(n + m)$ space). Observe that $\Delta = 5c \leq 10\sqrt{m}$, and so the amortized time per update is indeed $O(\sqrt{m})$.

Acknowledgements. We thank Tsvi Kopelowitz and Itay Gonshorovitz for helpful discussions.

References

[1] A. Aho, J. Hopcroft, , and J. Ullman. *The design and analysis of computer algorithms.* Addison-Wesley, 1974.

[2] D. Alberts and M. R. Henzinger. Average case analysis of dynamic graph algorithms. In *Proc. of 6th SODA*, pages 312–321, 1995.

[3] A. Anand, 2012. Personal communication.

[4] A. Andersson and M. Thorup. Dynamic ordered sets with exponential search trees. *J. ACM*, 54(3):13, 2007.

[5] B. Awerbuch and T. Leighton. Improved approximation algorithms for the multi-commodity flow problem and local competitive routing in dynamic networks. In *STOC*, pages 487–496, 1994.

[6] S. Baswana, M. Gupta, and S. Sen. Fully dynamic maximal matching in $O(\log n)$ update time. In *Proc. of 52nd FOCS*, pages 383–392, 2011.

[7] G. S. Brodal and R. Fagerberg. Dynamic representation of sparse graphs. In *Proc. of 6th WADS*, pages 342–351, 1999.

[8] A. M. Dean, J. P. Hutchinson, and E. R. Scheinerman. On the thickness and arboricity of a graph. *J. Comb. Theory, Ser. B*, 52(1):147–151, 1991.

[9] C. Demetrescu and G. F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992, 2004.

[10] C. Demetrescu and G. F. Italiano. Trade-offs for fully dynamic transitive closure on dags: breaking through the $O(n^2)$ barrier. *J. ACM*, 52(2):147–156, 2005.

[11] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification - a technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, 1997.

[12] M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, 1999.

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

[14] Z. Ivković and E. L. Lloyd. Fully dynamic maintenance of vertex cover. In *Proc. of 19th WG*, pages 99–111, 1993.

[15] M. T. J. Holm, K. de. Lichtenberg. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001.

[16] V. King and G. Sagert. A fully dynamic algorithm for maintaining the transitive closure. *J. Comput. Syst. Sci.*, 65(1):150–167, 2002.

[17] S. Micali and V. V. Vazirani. An $O(\sqrt{|V|}|E|)$ algorithm for finding maximum matching in general graphs. In *Proc. of 21st FOCS*, pages 17–27, 1980.

[18] C. Nash-Williams. Decomposition of finite graphs into forests. *Journal of the London Mathematical Society*, 39(1):12, 1964.

[19] K. Onak and R. Rubinfeld. Maintaining a large matching and a small vertex cover. In *Proc. of 42nd STOC*, pages 457–464, 2010.

[20] P. Sankowski. Faster dynamic matchings and vertex connectivity. In *Proc. of 18th SODA*, pages 118–126, 2007.

[21] M. Thorup. Fully-dynamic min-cut. *Combinatorica*, 27(1):91–127, 2007.