Fully dynamic maximal matching in $O(\log n)$ update time

Surender Baswana
Department of CSE,
I.I.T. Kanpur, India
sbaswana@cse.iitk.ac.in

Manoj Gupta
Department of CSE,
I.I.T. Delhi, India
gmanoj@cse.iitd.ernet.in

Sandeep Sen Department of CSE,
I.I.T. Delhi, India
ssen@cse.iitd.ernet.in

April 17, 2012

Abstract

We present an algorithm for maintaining maximal matching in a graph under addition and deletion of edges. Our data structure is randomized that takes $O(\log n)$ expected amortized time for each edge update where n is the number of vertices in the graph. While there is a trivial $O(n)$ algorithm for edge update, the previous best known result for this problem was due to Ivković and Llyod[4]. For a graph with n vertices and m edges, they give an $O((n+m)^{0.7072})$ update time algorithm which is sublinear only for a sparse graph.

For the related problem of maximum matching, Onak and Rubinfeld [5] designed a randomized data structure that achieves $O(\log^2 n)$ expected amortized time for each update for maintaining a c-approximate maximum matching for some large constant c. In contrast, we can maintain a factor two approximate maximum matching in $O(\log n)$ expected amortized time per update as a direct corollary of the maximal matching scheme. This in turn also implies a two approximate vertex cover maintenance scheme that takes $O(\log n)$ expected amortized time per update.

1 Introduction

In the last decade, there has been considerable research in *Dynamic* Graph Algorithms where we want to maintain a data structure associated with some property (like connectivity, transitive closure or matching) under insertion and deletion of edges. Even for a simple property like *connectivity*, it took researchers considerable effort to design a polylog(n) update time algorithm [2, 3]. In this work, we address fully dynamic maintenance of maximal matching in a graph.

Let $G = (V, E)$ be a graph on n vertices and m edges. A matching in G is a set of edges $\mathcal{M} \subseteq E$ such that no two edges in $\mathcal{M}$ share any vertex. A maximum matching is a matching that contains the largest possible number of edges. A matching is said to be a maximal matching if it cannot be strictly contained in any other matching. It is well known that a maximal matching guarantees a 2-approximation of the maximum matching. Ivković and Llyod [4] designed the first fully dynamic algorithm for maximal matching with $O((n+m)^{0.7072})$ update time. In contrast, there exists a much larger body of work for maximum matching.

Sankowski [6] gave an algorithm for the maintaining maximum matching which processes each update in $O(n^{1.495})$ time. Alberts and Henzinger [1] gave an expected $O(n)$ update time algorithm for maintaining maximum matching with respect to a *restricted random model*. Therefore the goal of a $\mathbf{polylog}(n)$ update

time dynamic maximum matching algorithm appears to be too ambitious. In particular, even achieving a $o(\sqrt{n})$ bound on the update time would imply an improvement of the longstanding $O(m\sqrt{n})$ bound of the best static algorithm for maximum matching due to Micali and Vazirani [**?**]. So approximation appears to be inevitable if we wish to achieve really fast update time for maintaining matching. Recently, Onak and Rubinfeld [5] presented a randomized algorithm for maintaining a c-approximate (for some large constant c) matching in a dynamic graph that takes $O(\log^2 n)$ amortized time for each edge update. This matching is not necessarily maximal, as a maximal matching would imply a factor two approximate maximum matching. In particular, they pose the following question -

"Our approximation factors are large constants. How small can they be made with polylogarithmic update time ? Can they be made 2 ? Can the approximation constant be made smaller than two for maximum matching ?.."

We resolve one of their central questions by presenting a fully dynamic algorithm for maximal matching which achieves $O(\log n)$ expected amortized time per edge insertion or deletion. Our bound also implies a similar result for maintaining a two approximate vertex cover.

2 An overview

Let $\mathcal{M}$ denote the matching of the given graph at any moment. Every edge of $\mathcal{M}$ is called a *matched* edge and an edge in $E\backslash\mathcal{M}$ is called an *unmatched* edge. For an edge $(u, v) \in \mathcal{M}$, we define u to be the *mate* of v and v to be the *mate* of u. For a vertex x if there is an edge incident to it from $\mathcal{M}$, then x a *matched* vertex; otherwise it is *free* or *unmatched*.

In order to maintain a maximal matching, it suffices to ensure that there is no edge (u, v) in the graph such that both u and v are free with respect to the matching. From this observation, an obvious approach will be to maintain the information for each vertex whether it is matched or free at any stage. When an edge (u, v) is inserted, add (u, v) to the matching if u and v are free. For a case when an unmatched edge (u, v) is deleted, no action is required. Otherwise, for both u and v we search their neighborhood for any free vertex and update the matching accordingly. It follows that each update takes $O(1)$ computation time except when it involves deletion of a matched edge; in this case the computation time is of the order of the sum of the degrees of the two vertices. So this trivial algorithm is quite efficient for *small* degree vertices, but could be expensive for *large* degree vertices. An alternate approach to handling deletion of a matched edge is to use a simple randomized technique - a vertex u is matched with a randomly chosen neighbor v. Following the standard adversarial model, it can be observed that an expected $\deg(u)/2$ edges incident to u will be deleted before deleting the matched edge (u, v). So the expected amortized cost per edge deletion for u is roughly $O\Big(\frac{\deg(u)+\deg(v)}{\deg(u)/2}\Big)$. If $\deg(v) \gg \deg(u)$, then this update time can be as bad as the one obtained by the trivial algorithm mentioned above; but if $\deg(u)$ is high, the update time is better. We combine the idea of choosing a random mate and the trivial algorithm suitably as follows. We introduce the notion of *ownership* of edges wherein we assign an edge to that endpoint which has *higher* degree. We maintain a partition of the set of vertices into two levels : 0 and 1. Level 0 consists of vertices which own *few* edges and we handle the updates in level 0 using the trivial algorithm. The level 1 consists of vertices (and their mates) which own *large* number of edges and we use the idea of random mate to handle their updates. In particular, a vertex chooses a random mate from its set of owned edges which ensures that it selects a neighbor having a lower degree. This is the basis of our first fully dynamic algorithm which achieves expected amortized $O(\sqrt{n})$ time per update.

A careful analysis of the $O(\sqrt{n})$ update time algorithm suggests that a *finer* partition of vertices may help in achieving a better update time. This leads to our main algorithm which achieves expected amortized $O(\log n)$ time per update. More specifically, our algorithm maintains an invariant that can be informally summarized as follows.

Each vertex tries to rise to a level higher than its current level if upon reaching that level, there are sufficiently large number of edges incident on it from lower levels. Once a vertex reaches a new level, it selects a random edge from this set and makes it matched.

2.1 Related Work

Onak and Rubinfeld [5] also pursue an approach based on use of randomization to achieve efficient updates and maintain a partitioning of vertices into a hierarchy of $O(\log n)$ level that is along the lines of Parnas and Ron [8]. The algorithm of Onak and Rubinfeld [5] takes a global approach in building level i of this hierarchy as follows. For level i, they consider the subgraph consisting of vertices V_i and their neighbors and argue that a random subset of these edges form a matching of size $|V_i|/a$ with high probability for some constant $a > 1$. If the matching at level i falls below a predefined threshold, then a new matching is computed for the vertices at level i. The matching algorithm always tries to ensure that the matched edges at level i is always greater than $|V_i|/a$. As a consequence, a free vertex at level i may not get processed if the matching size at level i is above the threshold. This is the reason that the matching obtained is not maximal. The approximation factor a is an outcome of some probabilistic calculations using Chernoff bounds that is chosen to be a *sufficiently* large. Therefore, it is unlikely that any simple variation of this global approach can lead to a maximal matching.

We also maintain a hierarchical partitioning of vertices but it is distinctly different from the scheme of Onak and Rubinfeld [5]. As described earlier, the update algorithm in [5] may not process a vertex at level i if the matching at level i is above a certain threshold. This is a *global* approach of maintaining large matching at level i. On the other hand, we process a free vertex at level i as soon as it becomes *free*. Irrespective of the matching size at level i, we try to find a matched edge for this *free* vertex. This is a *vertex centric* approach for maintaining matching which ensures that the matching is maximal. Our algorithm achieves significantly better results than [5], i.e., a guaranteed factor 2 matching. The use of randomization is limited to choice of a random matching vertex and the $O(\log n)$ expected update time can be derived using $O(\log n)$ purely random bits.

2.2 Organization of the paper

For a gentle exposition of the ideas and techniques, we first describe a fully dynamic algorithm for maximal matching that has 2 levels and achieves expected amortized $O(\sqrt{n})$ time per update. This is followed by our final fully dynamic algorithm which has $\log n$ levels and achieves expected amortized $O(\log n)$ time per update (Theorem 4.1). All logarithms in this paper are with base 2 unless mentioned otherwise.

3 Fully dynamic algorithm with expected amortized $O(\sqrt{n})$ time per update

The algorithm maintains a partition of the set of vertices into two levels. We shall use $\text{LEVEL}(u)$ to denote the level of a vertex u. We define $\text{LEVEL}(u, v)$ for an edge (u, v) as $\max(\text{LEVEL}(u), \text{LEVEL}(v))$.

We now introduce the concept of *ownership* of the edges. Each edge present in the graph will be owned by one or both of its end points as follows. If both the endpoints of an edge are at level 0, then it is owned by both of them. Otherwise it will be owned by exactly that endpoint which lies at higher level. If both the endpoints are at level 1, the tie will be broken suitably by the algorithm. As the algorithm proceeds, the vertices will make transition from one level to another and the ownership of edges will also change accordingly. Let $\mathcal{O}_u$ denote the set of edges owned by u at any moment of time. Each vertex $u \in V$ will keep the set $\mathcal{O}_u$ in a dynamic hash table [7] so that each search or deletion on $\mathcal{O}_u$ can be performed in worst case $O(1)$ time and each insertion operation can be performed in expected $O(1)$ time. This hash table is

also suitably augmented with a linked list storing $\mathcal{O}_u$ so that we can retrieve all edges of set $\mathcal{O}_u$ in $O(|\mathcal{O}_u|)$ time.

The algorithm maintains the following three invariants before the next update is processed.

1. Every vertex at level 1 is matched. Every free vertex at level 0 has all its neighbors matched.
2. Every vertex at level 0 owns less than $\sqrt{n}$ edges at any moment of time.
3. Both the endpoints of matched edges are at the same level.

The first invariant implies that the matching $\mathcal{M}$ maintained is maximal at each stage. A vertex u is said to be a *dirty* vertex at a moment if at least one of its invariants does not hold. In order to restore the invariants, each dirty vertex might make transition to some new level and do some processing. This processing involves owning or disowning some edges depending upon whether the level of the vertex has risen or fallen. Thereafter, the vertex will execute RANDOM-SETTLE or NAIVE-SETTLE to *settle down* at its new level. The pseudocode for insert and delete operation is given in Figure 1 and Figure 2.

Handling insertion of an edge

Let (u, v) be the edge being inserted. If either u or v are at level 1, there is no violation of any invariant. So the only processing that needs to be done is to assign (u, v) to $\mathcal{O}_u$ if LEVEL$(u) = 1$, and to $\mathcal{O}_v$ otherwise. This takes $O(1)$ time. However, if both u and v are at level 0, then we execute HANDLING-INSERTION procedure which does the following (see Figure 1).

If u and v are free, then insertion of (u, v) has violated the first invariant for u as well as v. We restore it by adding (u, v) to $\mathcal{M}$. Note that the insertion of (u, v) also leads to increase of $|\mathcal{O}_u|$ and $|\mathcal{O}_v|$ by one. We process that vertex out of u and v which owns larger number of edges; let u be that vertex. If $|\mathcal{O}_u| = \sqrt{n}$, then invariant 2 has got violated. We execute RANDOM-SETTLE(u); as a result, u moves to level 1 and gets matched to some vertex, say y, selected randomly uniformly from $\mathcal{O}_u$. Vertex y moves to level 1 to satisfy Invariant 3. If w and x were respectively the earlier mates of u and y at level 0, then the matching of u with y has rendered w and x free. So to restore invariant 1, we execute NAIVE-SETTLE(w) and NAIVE-SETTLE(x). This finishes the processing of insertion of (u, v). Note that when u rises to level 1, $|\mathcal{O}_v|$ remains unchanged. Since all the invariants for v were satisfied before the current edge update, it follows that the second invariant for v still remains valid.

Handling deletion of an edge

Let (u, v) be an edge that is deleted. If $(u, v) \notin \mathcal{M}$, all the invariants are still valid. So let us consider the nontrivial case when $(u, v) \in \mathcal{M}$. In this case, the deletion of (u, v) has made u and v free. Therefore, potentially the first invariant might have got violated for u and v, making them dirty. We do the following processing in this case.

If edge (u, v) was at level 0, then following the deletion of (u, v), vertex u executes NAIVE-SETTLE(u), and then vertex (v) executes NAIVE-SETTLE(v). This restores the first invariant and the vertices u and v are *clean* again. If edge (u, v) was at level 1, then u is processed using the procedure shown in Figure 2 which does the following (v is processed similarly).

First, u disowns all its edges whose other endpoint is at level 1. If $|\mathcal{O}_u|$ is still greater than or equal to $\sqrt{n}$, then u stays at level 1 and executes RANDOM-SETTLE(u). If $|\mathcal{O}_u|$ is less than $\sqrt{n}$, u moves to level 0 and executes NAIVE-SETTLE(u). Note that the transition of u from level 1 to 0 leads to an increase in the number of edges owned by each of its neighbors at level 0. The second invariant for each such neighbor, say w, may get violated if $|\mathcal{O}_w| = \sqrt{n}$, making w dirty. So we scan each neighbor of u sequentially and for each

Procedure HANDLING-INSERTION(u, v)

if u *and* v *are* FREE **then** $\mathcal{M} \leftarrow \mathcal{M} \cup \{(u, v)\}$;
if $|\mathcal{O}_v| > |\mathcal{O}_u|$ **then** swap(u, v);
if $|\mathcal{O}_u| = \sqrt{n}$ **then**
    foreach $(u, w) \in \mathcal{O}_u$ **do**
        delete (u, w) from $\mathcal{O}_w$;
    $x \leftarrow$ RANDOM-SETTLE(u);
    if $x \neq$ NULL **then** NAIVE-SETTLE(x);
    if w *was previous mate of* u **then** NAIVE-SETTLE(w);

Procedure RANDOM-SETTLE(u): Finds a random edge (u, v) from the owned edges of u and returns the previous mate of v

Let (u, v) be a uniformly randomly selected edge from $\mathcal{O}_u$;
foreach $(v, w) \in \mathcal{O}_u$ **do**
    delete (v, w) from $\mathcal{O}_w$;
if v *is matched* **then**
    $x \leftarrow$ MATE(v);
    $\mathcal{M} \leftarrow \mathcal{M} \backslash \{(v, x)\}$
else
    $x \leftarrow$ NULL;
$\mathcal{M} \leftarrow \mathcal{M} \cup \{(u, v)\}$;
LEVEL$(u) \leftarrow 1$; LEVEL$(v) \leftarrow 1$;
return x;

Procedure NAIVE-SETTLE(u) : Finds a free vertex adjacent to u deterministically

for *each* $(u, x) \in \mathcal{O}_u$ **do**
    if x *is free* **then**
        $\mathcal{M} \leftarrow \mathcal{M} \cup \{(u, x)\}$;
        Break;

Figure 1: Procedure for handling insertion of an edge (u, v) where LEVEL$(u) =$ LEVEL$(v) = 0$.

```
Procedure HANDLING-DELETION(u,v)
1  foreach (u, w) ∈ O_u and LEVEL(w) = 1 do
2      move (u, w) from O_u to O_w;
3  if |O_u| ≥ √n then
4      x ← RANDOM-SETTLE(u);
5      if x ≠ NULL then NAIVE-SETTLE(x);
   else
6      LEVEL(u) ← 0;
7      NAIVE-SETTLE(u);
8      foreach (u, w) ∈ O_u do
9          if |O_w| ≥ √n then
10             x ← RANDOM-SETTLE(w);
11             if x ≠ NULL then NAIVE-SETTLE(x);
```

Figure 2: Procedure for processing u when $(u, v) \in \mathcal{M}$ is deleted and LEVEL(u)=LEVEL(v)=1.

dirty neighbor w (that is, $|\mathcal{O}_w| \geq \sqrt{n}$), we execute RANDOM-SETTLE(w) to restore the second invariant. This finishes the processing of deletion of (u, v).

It can be observed that, unlike insertion of an edge, the deletion of an edge may lead to creation of a large number of dirty vertices. This may happen if the deleted edge is a matched edge at level 1 and at least one of its endpoints move to level 0.

3.1 Analysis of the algorithm

We analyze the algorithm using the concept of *epochs*, which we explain as follows. While processing the sequence of insertions and deletions of edges, some matched edges become unmatched and some unmatched edges become matched.

Definition 3.1 *Consider any edge (u, v), and let it be a matched edge at time t. Then the* **epoch** *of (u, v) is the maximal continuous time period containing t for which it remains in $\mathcal{M}$.*

The entire life span of an edge (u, v) consists of a sequence of epochs of (u, v) separated by the continuous periods when (u, v) is not matched.

The third invariant implies that epochs can be partition into epochs at level 0 and level 1. In order to bound the computation time per update, we calculate the computation involved in an epoch at level 0 and 1. We will see that the major computations are done either at time of the start of an epoch (when the edge is included in the matching) or at the end of an epoch(when the edge removed from the matching). An update steps may involve starting and ending of many epoch. Let the total time taken by our algorithm at update step t is C_t. For the analysis we will redistribute this computational time among the various epochs at step t. Precisely, if an epoch of (u, v) starts at time t, let the computational cost associated with the start of this epoch be $S_t^{(u,v)}$. Similarly, let the computational cost associated with the end of epoch of (w, y) be $Q_t^{(w,y)}$. We redistribute C_t in such a way that the sum of all the S_t's and Q_t's is C_t at update step t. Now, we look at all the procedures of our algorithm and try to distribute their computation cost among various epochs.

1. NAIVE-SETTLE(u)
   In the procedure NAIVE-SETTLE(u), either u finds a mate v or it becomes free. In both the cases the

time required is $O(\sqrt{n})$ as u searches atmost $\sqrt{n}$ owned edges at level 0. Let us deal with these cases separately.

(a) *u manages to find a new mate v*
In this case, an epoch of (u, v) starts. The $O(\sqrt{n})$ time is associated with the start of the epoch of (u, v).

(b) *u becomes free*
Even in this case the time required is $O(\sqrt{n})$. We have to associate this cost to some epoch. Let us now look at the procedure NAIVE-SETTLE(u) closely. NAIVE-SETTLE(u) can be called from two places: HANDLING-INSERTION (line 7 and line 8) and HANDLING-DELETION(line 5 and line 11). In these case NAIVE-SETTLE(u) is executed because of the following reason: the previous mate of u(say w) has moved to level 1. w moves to level 1 either to execute RANDOM-SETTLE or due to the fact that some vertex at level 1 executed RANDOM-SETTLE and chose w as its random mate. So the computation cost of $O(\sqrt{n})$ is associated with the end of epoch of (u, w).

2. RANDOM-SETTLE(u)
In RANDOM-SETTLE(u), u finds a random mate v from level 0. It then pulls v to level 1 to satisfy the third invariant. In this process, v becomes the sole owner of all the edges that have other endpoint at level 0(line 2,3). Since v was the owner of atmost $\sqrt{n}$ edges, the total computation time involved in performing this step is $O(\sqrt{n})$. Other steps in RANDOM-SETTLE can be executed in $O(1)$ time. So the total computation time is $O(\sqrt{n})$. This computation time is associated with the start of the epoch (u, v).

3. HANDLING-INSERTION(u, v)
We have already taken care of line 6,7 and 8 of INSERTION-HANDLE as it calls either RANDOM-SETTLE or NAIVE-SETTLE on those lines. The only non-trivial step in this procedure left is line 4 and 5. Here, vertex u is about to execute RANDOM-SETTLE(u). In a preparation for this, it becomes the sole owner of its edges at level 0. Since u was the owner of atmost $\sqrt{n}$ edges, the total computation time involved in performing this step is $O(\sqrt{n})$. This computation time is associated with the start of the epoch (u, v).

4. HANDLING-DELETION(u, v) Again note that we have already accounted for the major part of this procedure. Only line 1 and line 2 are the non-trivial section of this procedure that are not accounted. Here a matched edge of u was deleted and u starts this procedure by disowning all its owned edges whose other endpoint is at level 1. The number of such edges can be $O(n)$. Therefore, the computation time is $O(n)$ and it is associated to the end of the previous epoch of u.

Excluding the updates that cause the start and end of an epoch of (u, v), every other edge update on u and v during the epoch is handled in just $O(1)$ time. Therefore, we shall focus only on the amount of computation associated with the start and end of an epoch. Let us now analyze the actual computation associated with the epoch at level 0 and level 1.

- Epoch at level 0
The only non-trivial computation associated with the start of the epoch is in NAIVE-SETTLE. We have already seen that the computational cost of NAIVE-SETTLE is $O(\sqrt{n})$ and can be associated with the start or end of some epoch.

- Epoch at level 1
Consider the epoch of (u, v) at level 1. There are two ways in which epoch at level 1 starts:

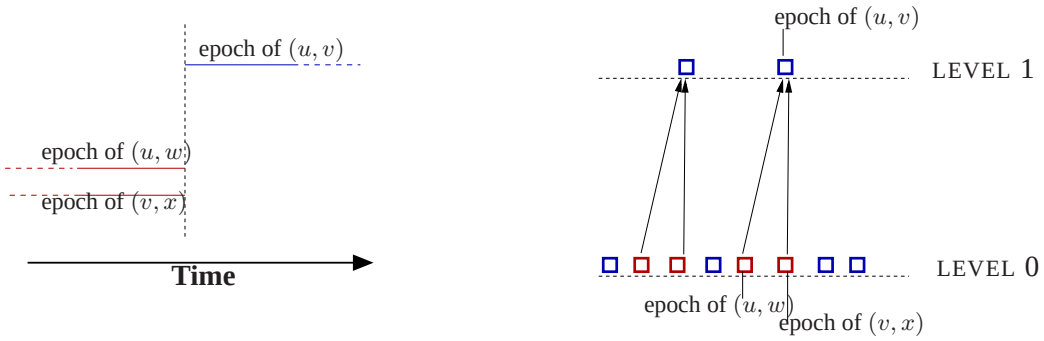

Figure 3: Epochs at level 0 and 1; the creation of an epoch at level 1 can destroy at most two epochs at level 0.

- In HANDLING-INSERTION
  The computation cost associated with the start of the epoch of (u, v) is atmost the computation cost required in executing the procedure HANDLING-INSERTION. We have already calculated that line 4,5 of HANDLING-INSERTION takes $O(\sqrt{n})$ time. All this computation cost is associated to epoch of (u, v).
- In HANDLING-DELETION
  In HANDLING-DELETION, a epoch of (u, v) may start due to calling RANDOM-SETTLE at line 4 or at line 10. In both the cases the cost associated is equal to the cost of executing RANDOM-SETTLE which is $O(\sqrt{n})$.

Now let us calculate the computation cost associated with epoch of (u, v) when it ends. The only place where computation cost is associated with the end of the epoch is at line 1-2 of HANDLING-DELETION. This cost is atmost $O(n)$.

From our analysis, it follows that the amount of computation associated with an epoch at level 0 is $O(\sqrt{n})$ and to level 1 is $O(n + \sqrt{n})$ which is $O(n)$.

An epoch corresponding to some edge, say (u, v), ends because of exactly one of the following causes.

(i) if (u, v) is deleted from the graph.

(ii) u (or v) get matched to some other vertex leaving its current mate free.

An epoch will be called a *natural* epoch if it ends due to cause (i); otherwise it will be called an *induced* epoch. *Induced* epoch can end prematurely since, unlike natural epoch, the matched edge is not actually deleted from the graph when an *induced* epoch ends.

It follows from the algorithm described above that every epoch at level 1 is a natural epoch whereas an epoch at level 0 can be natural or induced depending on the cause of its termination. Furthermore, each induced epoch at level 0 can be associated with a natural epoch at level 1 whose creation led to the termination of the former. In fact, there can be at most two induced epochs at level 0 which can be associated with an epoch at level 1. It can be explained as follows (see Figure 3).

Consider an epoch at level 1 associated with an edge, say (u, v). Suppose it was created by vertex u. If u was already matched at level 0, let $w \neq v$ be its mate. Similarly, if v was also matched already, let $x \neq u$ be its current mate at level 0. So matching u to v terminates the epoch of (u, w) as well as the epoch of edge (v, x) at level 0. We *charge* the overall cost of these two epochs to the epoch of (u, v). We have seen that the computational cost associated with an epoch at level 0 is $O(\sqrt{n})$. So the overall computation *charged* to an epoch of (u, v) at level 1 is $O(n + 2\sqrt{n})$ which is $O(n)$.

Lemma 3.1 *The computation charged to a natural epoch at level 1 is $O(n)$ and the computation charged to a natural epoch at level 0 is $O(\sqrt{n})$.*

In order to analyze our algorithm, we just need to get a bound on the computation *charged* to all natural epochs at level 0 and level 1 during a sequence of updates. In particular, we need to bound the computation *charged* to all the natural epochs which either end during the updates or remain *alive* at the end of all the updates. By *alive*, we mean that matched edges corresponding to these epoch remain in the matching after all the update ends.

3.1.1 Bounding the computation charged to the natural epochs destroyed

Let t be the total number of updates. Each natural epoch at level 0 which is destroyed can be assigned uniquely to the deletion of its matched edge. Hence it follows from Lemma 3.1 that the computation *charged* to all natural epochs destroyed at level 0 during t updates is $O(t\sqrt{n})$.

Now we shall analyze the number of epochs destroyed at level 1. Consider an epoch at level 1 *initiated* by some vertex, u by selecting a random edge (u, v) from $\mathcal{O}_u^{init}$. From the invariant, $\mathcal{O}_u^{init}$ has at least $\sqrt{n}$ edges. Consider the sequence of edge deletions following the creation of the epoch initiated by u. Let t_u denote the subsequence of edges from $\mathcal{O}_u^{init}$, and observe that the this epoch is completely determined by t_u and the epoch ends when (u, v) is deleted. Since the vertex u selected a matched edge out of $\mathcal{O}_u^{init}$ uniformly at random, the matched edge of u may appear anywhere in this subsequence with equal probability [1]. The *duration* of the epoch initiated by u is the number of edges in $\mathcal{O}_u^{init}$ deleted during the epoch (before (u, v)) and is a random variable uniformly distributed in the range $[1, |\mathcal{O}_u^{init}|]$. See Figure 4.

Henceforth, t_u will denote the truncated subsequence and we denote the *duration* of this epoch by $|t_u|$. Since an edge e can be deleted and inserted multiple times in an update sequence, let e^i denote the i-th occurrence of the edge e - e^{i+1} can appear only after e^i is deleted. Let τ_u denote the *labelled* version of t_u where the edges carry the additional information of their occurrence index. Since an edge can be owned exactly by one *initiator* at any time, it implies the following observation.

Lemma 3.2 *τ_w and τ_u are disjoint for $w \neq u$.*

In our subsequent discussion, we will not distinguish between t_u and τ_u and assume that t_u's are disjoint.

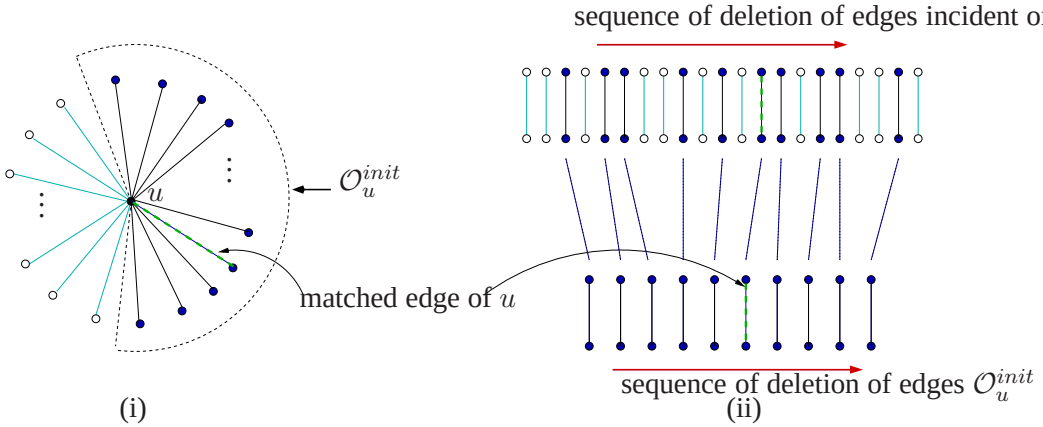

Figure 4: (i) Edges incident on u and the set $\mathcal{O}_u^{init}$ at any moment of time; (ii) The matched edge can appear any where in the subsequence of deletions of $\mathcal{O}_u^{init}$.

For a vertex u, let t'_u denote the concatenation of successive t_u for all the epochs initiated by u. For a deletion sequence t'_u, we will find the expected number of epochs which created this deletion sequence. We

[1] assuming that all the edges appear and only the first deletion of any edge is relevant

will prove that on expectation, the number of epochs for a deletion sequence t'_u is $O(|t'_u|/\sqrt{n})$. Since the t'_u are disjoint (Lemma 3.2), the expected total number of epochs is $O(t/\sqrt{n})$, where t is the total number of edges deleted at level 1. From Lemma 3.1, the expected total cost charged to these epochs is $O(t\sqrt{n})$.

Let X_t be the random variable denoting the number of epochs at level 1 destroyed during a sequence of t updates. As described above $X_t = \sum_u X_{|t'_u|}$, where t'_u is the deletion sequence for u as described above. Consider the *first* epoch of u to be the first epoch to terminate during a sequence of t'_u updates such that $|t'_u| = k$. At the moment of its creation, u owned $\mathcal{O}_u^{init}$ edges and the duration of this epoch is a random variable Z distributed uniformly in the range $[1...\mathcal{O}_u^{init}]$. Using the law of conditional expectation, that is, $\mathbf{E}[X_k = \mathbf{E}[\mathbf{E}[X_k|Z]]$, and setting $y_k = \mathbf{E}[X_k]$ and $M = |\mathcal{O}_u^{init}|$, we obtain the following recurrence.

$$y_k = \begin{cases} \frac{1}{M}\sum_{i=1}^{M}(y_{k-i}+1) & \text{for } k \geq M \\ \\ \frac{1}{M}\sum_{i=1}^{k-1}(y_i+1) & \text{for } 2 < k < M \end{cases} \tag{1}$$

where $y_1 = \Pr[Z = 1] = 1/M$. For $k < M$, by subtracting the recurrence of y_{k-1} from y_k, we obtain $y_k - y_{k-1} = \frac{1}{M}(y_{k-1}+1)$. Solving the recurrence, we obtain

$$\begin{aligned}
y_k &= \left(1+\frac{1}{M}\right)y_{k-1} + \frac{1}{M} \\
&= \left(1+\frac{1}{M}\right)^{k-1} y_1 + \frac{1}{M}\sum_{i=0}^{k-2}\left(1+\frac{1}{M}\right)^i = \left(1+\frac{1}{M}\right)^{k-1}\frac{1}{M} + \left(1+\frac{1}{M}\right)^{k-1} - 1 \\
&= \left(1+\frac{1}{M}\right)^{k} - 1 = \sum_{i=1}^{k}\binom{k}{i}\frac{1}{M^i} < \sum_{i=1}^{k}\left(\frac{ek}{i}\right)^i\frac{1}{M^i} \\
&= \frac{ek}{M} + \frac{ek^2}{4M^2} + \sum_{i=3}^{\infty}\left(\frac{ek}{iM}\right)^i < \frac{ek}{M} + \frac{ek^2}{4M^2} + \frac{ek}{3M}\sum_{i=2}^{\infty}\left(\frac{ek}{3M}\right)^i \\
&\leq \frac{ek}{M} + \frac{ek}{4M} + \frac{ek}{3M}\sum_{i=2}^{\infty}\left(\frac{e}{3}\right)^i \qquad \text{(since } k/M < 1\text{)} \\
&\leq \frac{ek}{M} + \frac{ek}{4M} + 9\frac{ek}{3M} \qquad \text{(since } \textstyle\sum_{i=2}^{\infty}\left(\frac{e}{3}\right)^i < 9\text{)} \\
&\leq \frac{5ek}{M}
\end{aligned}$$

We will prove that for for all k, $y_k \leq 5ek/M$. We prove this by using induction on k. The base cases is true : $\forall 0 \leq i < M$, $y_i \leq 5ei/M$. By induction hypothesis, $\forall i \leq k-1$, $y_i \leq 5ei/M$. To show that the claim holds for y_k, we substitute the values of $y_{k-1}, y_{k-2}, \ldots, y_{k-M}$ in Equation 1. We obtain

$$\begin{aligned}
y_k &\leq \tfrac{1}{M}\left(\textstyle\sum_{i=1}^{M}\frac{5e(k-i)}{M} + 1\right) \\
&= 1 + 5ek/M - \left(\tfrac{5e}{M^2}\textstyle\sum_{i=1}^{M} i\right) \\
&= 1 + 5ek/M - \left(\tfrac{5e}{M^2}\tfrac{M(M+1)}{2}\right) \\
&\leq 1 + 5ek/M - 1 = 5ek/M
\end{aligned}$$

So $E[X_t] = \sum_u E[X_{|t'_u|}] \leq \sum_u 5e|t'_u|/\sqrt{n} = 5et/\sqrt{n}$.
Thus we conclude the following lemma:

Lemma 3.3 *For any $t > 0$, $\mathbf{E}[X_t] = O(t/\sqrt{n})$.*

Notice that this proof does not rely on the independence of the random numbers since it is obtained by summing up expectation of random variables representing number of epochs destroyed over subsequences of updates. For the base case of a sequence of length 1, we use a random number in the range $[1..M]$, where $M = |\mathcal{O}_u^{init}| \leq n$. To choose a random number R_M in the range $[1..M]$, we first generate a random number R in the range $[1..n]$ and if $R \leq n - (n \mod M)$ then set $R_M = (R \mod M) + 1$ else generate a new

R and repeat the procedure. The reason for choosing R in this way is to ensure uniform distribution in the range $[1..M]$. For $n \geq 2M$, the expected number of repetitions is less than 2.

If we allow total independence, then we can obtain this bound with high probability as well. Since the matched edge of an epoch is selected independent of other epochs, it follows that duration of each natural epoch at level 1 is independent of other natural epochs. Hence we can state the following lemma.

Lemma 3.4 *The probability that a given epoch at level 1 has duration at most i is bounded by $\frac{i}{\sqrt{n}}$*

Lemma 3.5 *For any given $q \geq 1$,*

$$\mathbf{Pr}[X_t = q] \leq \left(\frac{4et}{q\sqrt{n}}\right)^{q/2}$$

Proof: If there are q epochs destroyed during t updates, at least half of them have duration $\leq 2t/q$. Hence, $\mathbf{Pr}[X_t = q]$ is bounded by the probability that there are at least $q/2$ epochs of duration at most $\frac{2t}{q}$. So, using Lemma 3.4 and exploiting the independence among the epochs,

$$\mathbf{Pr}[X_t = q] \leq \binom{q}{q/2}\left(\frac{2t}{q\sqrt{n}}\right)^{q/2} \leq \left(\frac{4et}{q\sqrt{n}}\right)^{q/2}$$

For the last inequality we used $\binom{q}{q/2} \leq \left(\frac{eq}{q/2}\right)^i = (2e)^i$. □

We use Lemma 3.5 to analyze $\mathbf{E}[X_t]$ and deviation of X_t.

Lemma 3.6 *For any $t > 0$ X_t is $O(t/\sqrt{n} + \log n)$ with very high probability.*

Proof:

We choose $q_0 = 4\ (\log n + 4et/\sqrt{n})$. It follows from Lemma 3.5 that for any $q \geq q_0$, $\mathbf{Pr}[X_t = q]$ is of the form b^q where base $b < 1/2$. Hence $\mathbf{Pr}[X_t \geq q_0]$ is bounded by a geometric series with the first term $< 2^{-q_0}$ and the common ratio less than $1/2$. Furthermore $q_0 > 4 \log n$, hence $\mathbf{Pr}[X_t \geq q_0]$ is bounded by $2/n^4$. Hence X_t is bounded by $O(t/\sqrt{n} + \log n)$ with high probability. □

Notice that the proofs of 3.6 rely heavily on the total independence of random numbers used for selecting random mates. Using standard techniques based on generalized Chebyshev's inequality, we can obtain the same bound using $O(\log n)$ way independence (**[?]**).

Now, recall from Lemma 3.1 that each natural epoch destroyed at level 1 has $O(n)$ computation *charged* to it. So Lemmas 3.3 and 3.6, when combined together, imply the following result.

Lemma 3.7 *The computation cost* charged *to all the natural epochs which get destroyed during any sequence of t updates is $O(t\sqrt{n})$ in expectation and $O(t\sqrt{n} + n \log n)$ with high probability.*

Let us now analyze the cost *charged* to all those epochs which are alive at the end of t updates. Consider an epoch (say epoch of (u, v)) such that (u, v) is still in the matching after t updates. Note that u(as well as v) is involved in at most one live epoch. Since epoch of (u, v) doesn't end, the cost *charged* to this epoch is

1. The computation cost of starting the epoch is equal to the edges owned by u and v at the start of epoch (u, v). Since each vertex is involved in at most one live epoch, this cost is at most $\sum_{\substack{v:\text{epoch of } v \text{ is} \\ \text{live after } t \text{ updates}}} |\mathcal{O}_v^{init}|$,

   where $|\mathcal{O}_v^{init}|$ is the number of edges owned by v when the live epoch of v started. Note that the number of edges owned by v is less than the number of edges added to v during t updates. If t_v is the the total number of edges added to v, then $\sum_v \mathcal{O}_v \leq \sum_v t_v$. This implies that the cost associated to all live epochs $\leq \sum_v t_v = 2t$

2. The computation cost associated with the *induced* epoch which may have ended due to epoch of (u, v). The cost associated with these *induced* epoch is charged to *natural* epoch of (u, v). We know that can be atmost two such epoch(say epoch of (u, w) and (v, y)). By Lemma 3.1, this cost is $O(\sqrt{n})$. Since the total number of updates at t, the total live epoch at after t updates can atmost be t. So this computation charged to live epochs due to these *induced* epoch is at most $O(t\sqrt{n})$.

It thus follows that the computation cost *charged* to all the live epochs at after t updates is $O(t\sqrt{n})$. During any sequence of t updates, the total number of epochs created is equal to the number of epochs destroyed and the number of epochs that are alive at the end of t updates. Hence using Lemma 3.7 we can state the following theorem.

Theorem 3.1 *Starting with a graph on n vertices and no edges, we can maintain maximal matching for any sequence of t updates in $O(t\sqrt{n})$ time in expectation and $O(t\sqrt{n} + n \log n)$ with high probability.*

3.2 On improving the update time beyond $O(\sqrt{n})$

In order to extend our 2-LEVEL algorithm for getting a better update time, it is worth exploring the reason underlying $O(\sqrt{n})$ update time guaranteed by our 2-LEVEL algorithm. For this purpose, let us examine the second invariant more carefully. Let $\alpha(n)$ be the threshold for the maximum number of edges that a vertex at level 0 can own. Consider an epoch at level 1 associated with some edge, say (u, v). The computation associated with this epoch is of the order of the number of edges u and v own which can be $\Theta(n)$ in the worst case. However, the expected duration of the epoch is of the order of the minimum number of edges u can own at the time of its creation, i.e., $\Theta(\alpha(n))$. Therefore, the expected amortized computation per edge deletion for an epoch at level 1 is $O(n/\alpha(n))$. Balancing this with the $\alpha(n)$ update time at level 0, yields $\alpha(n) = \sqrt{n}$.

In order to improve the running time of our algorithm, we need to decrease the ratio between the maximum and the minimum number of edges a vertex can own during an epoch at any level. It is this ratio that actually bounds the expected amortized time of an epoch. This insight motivates us for having a finer partition of vertices : the number of levels should be increased to $O(\log n)$ instead of just 2. When a vertex creates an epoch at level i, it will own at least 2^i edges, and during the epoch it will be allowed to own at most $2^{i+1} - 1$ edges. As soon as it starts owning 2^{i+1} edges, it should migrate to higher level. Notice that the ratio of maximum to minimum edges owned by a vertex during an epoch gets reduced from $\sqrt{n}$ to a constant.

We pursue the approach sketched above combined with some additional techniques in the following section. This leads to a fully dynamic algorithm for maximal matching which achieves expected amortized $O(\log n)$ update time per edge insertion or deletion.

4 Fully dynamic algorithm with expected amortized $O(\log n)$ time per update

Like the 2-LEVEL algorithm, we maintain a partition of vertices among various levels. We iterate the difference in the partition vis-a-vis 2-LEVEL algorithm.

1. The fully dynamic algorithm maintains a partition of vertices among $\lfloor \log n \rfloor + 2$ levels. The levels are numbered from -1 to $\mathrm{L}_0 = \lfloor \log n \rfloor$. We will see that the algorithm moves a vertex to level i if the vertex is the owner of atleast 2^i edges at that moment. So a vantage point is needed for a vertex that does not own any edge. As a result, we introduce a level -1 that contains all the vertices that do not own any edge.

2. We use the notion of ownership of edges which is slightly different from the one used in 2-LEVEL algorithm. In the 2-LEVEL algorithm, at level 0, both the endpoints of the edge are the owner of the edge. Here, at every level, each edge is owned by exactly one of its endpoints. In particular, the endpoint at the higher level always owns the edge. If the two endpoints are at the same level, then the tie is broken appropriately by the algorithm.

Like the 2-LEVEL algorithm, each vertex u will maintain a dynamic hash table storing the edges $\mathcal{O}_u$ owned by it. In addition, the generalized fully dynamic algorithm will maintain the following data structure for each vertex u. For each $i \geq \text{LEVEL}(u)$, let $\mathcal{E}_u^i$ be the set of all those edges incident on u from vertices at level i and are not owned by u. For each vertex u and level $i \geq \text{LEVEL}(u)$, the set $\mathcal{E}_u^i$ will be maintained in a dynamic hash table. However the onus of maintaining $\mathcal{E}_u^i$ will not be on u. For any edge $(u, v) \in \mathcal{E}_u^i$, v will be responsible for the maintenance of (u, v) in $\mathcal{E}_u^i$ since $(u, v) \in \mathcal{O}_v$.

4.1 Invariants and a basic subroutine used by the algorithm

As can be seen from the 2-level algorithm, it is more cost-effective for each vertex u to get settled at a higher level once it owns a *large* number of edges. Pushing this idea still further, our fully dynamic algorithm will allow a vertex to rise to a higher level if it can own *sufficiently large* number of edges after moving there. In order to formally define this approach, we introduce an important notation here.

For a vertex v with $\text{LEVEL}(v) = i$,

$$\phi_v(j) = \begin{cases} |\mathcal{O}_v| + \sum_{i \leq k < j} |\mathcal{E}_v^k| & \text{if } j > i \\ 0 & \text{otherwise} \end{cases}$$

In other words, for any vertex v at level i and any $j > i$, $\phi_v(j)$ denote the number of edges which v can own if v rises to level j. Our algorithm will be based on the following strategy. If a vertex v has $\phi_v(j) \geq 2^j$, then v would rise to the level j. In case, there are multiple levels to which v can rise, v will rise to the highest such level. With this key idea, we now describe the three invariants which our algorithm will maintain.

1. Every vertex at level ≥ 0 is matched and every vertex at level -1 is free.
2. For each vertex v and for all $j > \text{LEVEL}(v)$, $\phi_v(j) < 2^j$ holds true.
3. Both the endpoints of the matched edge are at the same level.

By definition, vertices at level -1 cannot be owner of any edge. We will see that our algorithm moves a vertex from level -1 to a higher level as soon as it becomes the owner of some edges. In fact, the second invariant bounds us to act in such a manner. If a vertex v still remains at level -1 after owning some edges then $\phi_v(0) \geq 1 \geq 2^0$, violating the second invariant. This, together with the fact that an edge is owned by one of its endpoints, implies that a vertex at level -1 cannot be a neighbor of another vertex at that level. This fact together with the first invariant imply that the matching maintained by the algorithm will indeed be a maximal matching. Figure 5 depicts a snapshot of the algorithm. The second invariant captures the key idea described above - after processing every update there is no vertex which fulfills the criteria of rising. An edge update may lead to violation of the invariants mentioned above and the algorithm basically restores these invariants. This may involve rise or fall of vertices between levels. Notice that the second invariant of a vertex is influenced by the rise and fall of its neighbors. We now state and prove two lemmas which quantify this influence more precisely.

Lemma 4.1 *Consider a vertex u for which both invariants hold. The rise of any neighbor, say v, cannot violate the second invariant for u.*

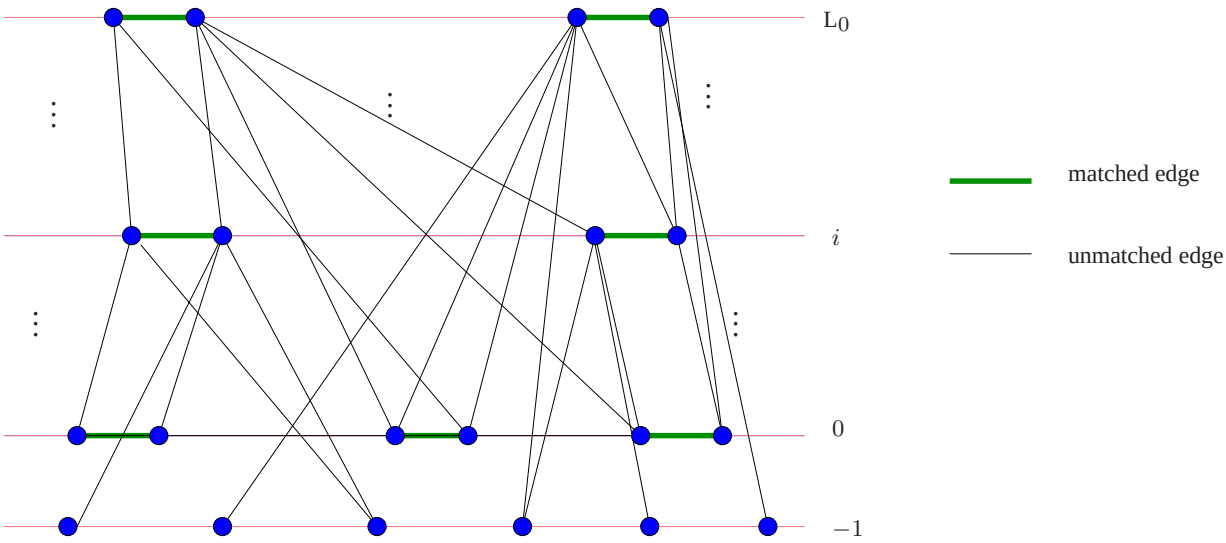

Figure 5: A snapshot of the algorithm on K_9: all vertices are matched(thick edges) except vertex x at level -1. Vertex v is the owner of just the edge (v, x). $\phi_v(2) = 2 < 2^2$ and $\phi_v(3) = 4 < 2^3$, so v cannot rise to a higher level.

Proof: Let LEVEL$(u) = k$. Since the invariants hold true for u before rise of v, so $\phi_u(i) < 2^i$ for all $i > k$. It suffices if we can show that $\phi_u(i)$ does not increase for any i due to the rise of v. We show this as follows.

Let vertex v rises from level j to ℓ. If $\ell \leq k$, the edge (u, v) continues to remain in $\mathcal{O}_u$, and so there is no change in $\phi_u(i)$ for any i. Let us consider the case when $\ell > k$. The rise of v from j to ℓ causes removal of (u, v) from $\mathcal{O}_u$ (or $\mathcal{E}_u^j$ if $j \geq k$) and insertion to $\mathcal{E}_u^\ell$. As a result $\phi_u(i)$ decreases by one for each i in $[\max(j, k) + 1, \ell]$, and remains unchanged for all other values of i. □

Lemma 4.2 *Consider a vertex u for which both invariants hold. Then any fall of one of its neighbors, say v, from level j to $j - 1$ increases $\phi_u(j)$ by at most one.*

Proof: Let LEVEL$(u) = k$. In case $k \geq j$, there is no change in $\phi_u(i)$ for any i due to fall of v. So let us consider the case $j > k$. In this case, the fall of v from level j to $j - 1$ leads to the insertion of (u, v) in $\mathcal{E}_u^{j-1}$ and deletion from $\mathcal{E}_u^j$. Consequently, $\phi_u(i)$ increases by one only for $i = j$ and remains unchanged for all other values of i. □

In order to detect any violation of the second invariant for a vertex v due to rise or fall of its neighbors, we shall maintain $\{\phi_v(i) | i \leq L_0\}$ in an array $\phi_v[]$ of size $L_0 + 2$. The updates on this data structure during the algorithm will involve the following two types of operations.

- DECREMENT-$\phi(v, I)$: As per Lemma 4.1, when a neighbor of v rises from level j to l, $\phi_v(i)$ decreases by one for all i in interval $[\max(j, \text{LEVEL}(v)) + 1, \ell]$. This operation decrements $\phi_v(i)$ by one for all i in interval $I = [\max(j, \text{LEVEL}(v)) + 1, \ell]$.

- INCREMENT-$\phi(v, i)$: this operation increases $\phi_v(i)$ by one. This operation will be executed when some neighbor of v falls from i to $i - 1$.

It can be seen that a single DECREMENT-$\phi(v, I)$ operation takes $O(|I|)$ time which is $O(\log n)$ in the worst case. On the other hand any single INCREMENT-$\phi(v, i)$ operation takes $O(1)$ time. However, since $\phi_v(i)$ is 0 initially and is non-negative always, we can conclude the following.

Lemma 4.3 *The computation cost of all* DECREMENT-$\phi()$ *operations over all vertices is upper-bounded by the computation cost of all* INCREMENT-$\phi()$ *operations over all vertices during the algorithm.*

Observation 4.1 *It follows from Lemma 4.3 that we just need to analyze the computation involving all* INCREMENT-$\phi()$ *operations since the computation involved in* DECREMENT-$\phi()$ *operations is subsumed by the former.*

Procedure GENERIC-RANDOM-SETTLE(u, i)

1 **if** LEVEL$(u) < i$ **then** //u owns edges till it reaches level i
2 **for** *each* $j =$ LEVEL(u) *to* $i - 1$ **do**
3 **for** *each* $(u, w) \in \mathcal{E}_u^j$ **do**
4 transfer (u, w) from $\mathcal{E}_u^j$ to $\mathcal{E}_w^i$;
5 transfer (u, w) from $\mathcal{O}_w$ to $\mathcal{O}_u$;
6 DECREMENT-$\phi(w, [j + 1, i])$;
7 Let (u, v) be a uniformly randomly selected edge from $\mathcal{O}_u$;
8 **if** v *is matched* **then**
9 $x \leftarrow$ MATE(v);
10 $\mathcal{M} \leftarrow \mathcal{M} \backslash \{(v, x)\}$
else
11 $x \leftarrow$ NULL
12 **for** *each* $j =$ LEVEL(v) *to* $i - 1$ **do** //v rises to level i and thus owns edges incident from vertices at levels LEVEL(v) to $i-1$
13 **for** *each* $(v, w) \in \mathcal{E}_v^j$ **do**
14 transfer (v, w) from $\mathcal{E}_v^j$ to $\mathcal{E}_w^i$;
15 transfer (v, w) from $\mathcal{O}_w$ to $\mathcal{O}_v$;
16 DECREMENT-$\phi(w, [j + 1, i])$;
17 $\mathcal{M} \leftarrow \mathcal{M} \cup \{(u, v)\}$;
18 LEVEL$(u) \leftarrow i$; LEVEL$(v) \leftarrow i$;
19 return x;

Figure 6: Procedure used by a free vertex u to *settle* at level i.

If any invariant of a vertex, say u, gets violated, it might rise or fall, though in some cases, it may still remain at the same level. However, in all these cases, eventually the vertex u will execute the procedure, GENERIC-RANDOM-SETTLE, shown in Figure 6. This procedure is essentially a generalized version of RANDOM-SETTLE(u) which we used in the 2-level algorithm. GENERIC-RANDOM-SETTLE(u, i) starts with moving u from its current level (LEVEL(u)) to level i. If level i is higher than the previous level of u, u acquires the ownership of all the edges whose endpoints lie at the level $\in$ [LEVEL$(u), i - 1$]. For each such edge (u, v) that is now owned by u, we perform DECREMENT-$\phi(v, [$LEVEL$(v) + 1, i])$ to reflect that the edge is now owned by vertex u which has moved to level i. Henceforth, the procedure then resembles RANDOM-SETTLE. It finds a random edge (u, v) from O_v and moves v to level i. The procedure returns the previous mate of v, if v was matched.

Lemma 4.4 *Consider a vertex u that executes* GENERIC-RANDOM-SETTLE(u, i) *and ends up selecting a mate v. Excluding the time spent in* DECREMENT$-\phi$ *operations, the computation time of this procedure is*

of the order of $|\mathcal{O}_u| + |\mathcal{O}_v|$ where $\mathcal{O}_u$ and $\mathcal{O}_v$ is the set of edges owned by u and v just at the end of the procedure.

4.2 Handling edge updates by the fully dynamic algorithm

Our fully dynamic algorithm will employ a generic procedure called PROCESS-FREE-VERTICES(). The input to this procedure is a sequence S consisting of ordered pairs of the form (x, k) where x is a free vertex at level $k \geq 0$. Observe that the presence of free vertices at level ≥ 0 implies that matching $\mathcal{M}$ is not necessarily maximal. In order to preserve maximality of matching, the procedure PROCESS-FREE-VERTICES restores the invariants of each such free vertex till S is emptied. We now describe our fully dynamic algorithm.

Handling deletion of an edge

Consider deletion of an edge, say (u, v). For each $j > \max(\text{LEVEL}(u), \text{LEVEL}(v))$, we decrement $\phi_u(j)$ and $\phi_v(j)$ by one. If (u, v) is an unmatched edge, no invariant gets violated. So we only delete the edge (u, v) from the data structures of u and v. Otherwise, let $k = \text{LEVEL}(u) = \text{LEVEL}(v)$. We execute the Procedure PROCESS-FREE-VERTICES($\langle\langle(u, k), (v, k)\rangle\rangle$).

Handling insertion of an edge

Consider the insertion of an edge, say (u, v). We check if the second invariant has got violated for either of u or v. The invariant may get violated for u (likewise for v) if there is any integer $i > \max(\text{LEVEL}(u), \text{LEVEL}(v))$, such that $\phi_u(i)$ was $2^i - 1$ just before the insertion of edge (u, v). In case there are multiple such integers, let $i_{\max}$ be the largest such integer. We increment $\phi_u(\ell)$ and $\phi_v(\ell)$ by one for each $\ell > i_{\max}$. To restore the invariant, u leaves its current mate, say w, and rises to level $i_{\max}$. We execute GENERIC-RANDOM-SETTLE$(u, i_{\max})$, and let x be the vertex returned. Let j and k be respectively the levels of w and x. Note that x and w are two free vertices now. We execute PROCESS-FREE-VERTICES($\langle\langle(x, k), (w, j)\rangle\rangle$).

Suppose that the insertion of edge (u, v) violates the second invariant for both u and v. Wlog assume that u is at a higher level than v. We add (u, v) to $\mathcal{O}_u$ and $E_v^{\text{LEVEL}(u)}$. The highest level to which u can rise after this insertion is j. This implies $\mathcal{O}_u + \sum_{\text{LEVEL}(u) \leq k < j} |\mathcal{E}_u^k| = 2^j$ just after this edge is inserted. Similarly, v may rise atmost to level l and $\mathcal{O}_v + \sum_{\text{LEVEL}(v) \leq k < l} |\mathcal{E}_v^k| = 2^l$. If $j > l$, we select u which can rise to the higher level and restore its second invariant. After moving u to level j, edge (u, v) becomes an element of $\mathcal{E}_v^j$. So $\sum_{\text{LEVEL}(v) \leq k < l} |\mathcal{E}_v^k|$ decreases by 1. So, $\mathcal{O}_v + \sum_{\text{LEVEL}(v) \leq k < l} |\mathcal{E}_u^k|$ decreases by one and is now strictly less than 2^l, thus the second invariant for v is also restored. If $i > j$, then we first process v and move it to a higher level and the invariant of u is restored automatically. So we only process the vertex which can move to a higher level in this case.

4.2.1 Description of Procedure PROCESS-FREE-VERTICES

The procedure receives a sequence S of ordered pairs (x, i) such that x is a free vertex at level i. It processes the free vertices in the decreasing order of their levels starting from L_0. We give an overview of this processing at level i. For a free vertex at level i, if it owns *sufficiently* large number of edges, then it settles at level i and gets matched by selecting a random edge from the edges owned by it. Otherwise the vertex falls down by one level. Notice that the fall of a vertex from level i to $i - 1$ may lead to rise of some of its neighbors lying at level $< i$. However, as follows from Lemma 4.2, for each such vertex v, only $\phi_v(i)$ increases by one and $\phi_v()$ value for all other level remains same. So the second invariant may get violated only for $\phi_v(i)$. This implies that v will rise only to level i. After these rising vertices move to level i(by

Procedure PROCESS-FREE-VERTICES(S)

```
1  for each (x, i) ∈ S do ENQUEUE(Q[i], x);
2  for i = L_0 to 0 do
3      while Q[i] ≠ ∅ do
4          v ← DEQUEUE(Q[i]);
5          if FALLING(v) then                                   //v falls to i − 1
6              LEVEL(v) ← i − 1;
7              ENQUEUE(Q[i − 1], v);
8              for each u ∈ O_v do
10                 transfer (u, v) from E_u^i to E_u^{i−1};
11                 INCREMENT-φ(u, i);
12                 INCREMENT-φ(v, i);
13                 if φ_u(i) ≥ 2^i then                          //u rises to i
14                     x ← GENERIC-RANDOM-SETTLE(u, i);
15                     if x ≠ NULL then
16                         ℓ ← LEVEL(x);
17                         ENQUEUE(Q[ℓ], x);
18
           else                                                  //v settles at level i
19             x ← GENERIC-RANDOM-SETTLE(v, i);
20             if x ≠ NULL then
21                 ℓ ← LEVEL(x);
22                 ENQUEUE(Q[ℓ], x);
```

Function FALLING(v)

```
1  i ← LEVEL(v);
2  for each (u, v) ∈ O_v such that LEVEL(u) = i do        //v disowns all edges at level i
3      transfer (u, v) from O_v to O_u;
4      transfer (u, v) from E_u^i to E_v^i;
5  if |O_v| < 2^i then return TRUE else return FALSE;
```

Figure 7: Procedure for processing free vertices given as a sequence S of ordered pairs (x, i) where x is a free vertex at LEVELi.

executing GENERIC-RANDOM-SETTLE), we move onto level $i-1$ and proceed similarly. Overall, the entire process can be seen as a wave of free vertices falling level by level. Eventually this wave of free vertices reaches level -1 and fades away ensuring maximal matching. With this overview, we now describe the procedure in more details and its complete pseudocode is given in Figure 7.

The procedure uses an array Q of size $\mathrm{L}_0 + 2$, where $Q[i]$ is a pointer to a queue (initially empty) corresponding to level i. For each ordered pair $(x, k) \in S$, it inserts x into queue $Q[k]$. The procedure executes a for loop from L_0 down to 0 where the ith iteration extracts and processes the vertices of queue $Q[i]$ one by one as follows. Let v be a vertex extracted from $Q[i]$. First we execute the function FALLING(v) which does the following. v disowns all its edges whose other endpoint lies at level i. If v owns less than 2^i edges then it is decided that v has to fall, otherwise v will continue to stay at level i.

1. *v has to stay at level i*
   v executes GENERIC-RANDOM-SETTLE and selects a random mate, say w, from level $j < i$ (if w is present in $Q[j]$ then it is removed from it and is raised to level i). If x was the previous mate of w, then x is a falling vertex. Vertex x gets added to $Q[j]$. This finishes the processing of v.

2. *v owns less than 2^i edges and has to fall*
   In this case, v falls to level $i-1$ and is inserted to $Q[i-1]$. This fall leads to increase $\phi_u(i)$ by one for each neighbor u of v lying at level lower than i (see Lemma 4.2). In case $\phi_u(i)$ has become 2^i, u has to rise to level i and is processed as follows. u executes GENERIC-RANDOM-SETTLE and selects a random mate, say w from level $j < i$. If w was in $Q[j]$ then it is removed from it. If x was the previous mate of w, then x is a falling vertex, and so it gets added to queue $Q[j]$.

In case 1, v remains at the same level and w moves to the level of v. This renders x free and x is added to the $Q[j]$. We want to see if the invariant of any other vertex is violated in processing v and w. Since x is free, the first invariant of x is violated. So x is added to the queue at its level. The processing of v does not change ϕ_u for any neighbor u of v. Furthermore, the rise of w does not lead to the violation of any invariant due to Lemma 4.1. In case 2, v falls to level $i-1$ and due to this some vertex rise to level i. All such rising vertex execute GENERIC-RANDOM-SETTLE. The second invariant is not violated for other vertices except these vertices. As in case 1, we see that processing these rising vertices may create some free vertices which are duly added to the queue at their level. However, their processing does not break second invariant to any other vertex.
Thus we conclude the following lemma.

Lemma 4.5 *After ith iteration of the for loop of* PROCESS-FREE-VERTICES*, the free vertices are present only in the queues at level $< i$, and for all vertices not belonging to these queues the three invariants holds.*

Lemma 4.5 establishes that after termination of procedure PROCESS-FREE-VERTICES, there are no free vertices at level ≥ 0 and all the invariants get restored globally. We want to mention this specially for the second invariant. The second invariant ensures that if a vertex can rise to a higher level, it should rise. We have seen that many vertex may rise from one level to a higher level. The algorithm processes these vertices in any arbitrary order. Since every vertex acts in a local way, when it is processed, it restores its second invariant if it is indeed violated at that moment. Lemma 4.5 ensures that there is a stable state for our algorithm, when all the vertices have their invariant restored.

4.3 Analysis of the algorithm

Processing the deletion or insertion of an edge (u, v) begins with decrementing or incrementing $\phi_u(i)$ and $\phi_v(i)$ for all levels $i \geq \max(\text{LEVEL}(u), \text{LEVEL}(v))$. The computation associated with this task over a

sequence of t updates will be $O(t \log n)$. This task may be followed by executing the procedure PROCESS-FREE-VERTICES. We would like to mention an important point here. Along with other processing, the execution of this procedure involves INCREMENT-$\phi()$ and DECREMENT-$\phi()$ operations. However, as implied by Observation 4.1, the computation involving DECREMENT-$\phi()$ is subsumed by INCREMENT-$\phi()$ operations.

Our analysis of the entire computation performed while processing a sequence of t updates is along similar lines to the 2-LEVEL algorithm. We visualize the entire algorithm as a sequence of creation and termination of various matched epochs. All we need to do is to analyze the number of epochs created and terminated during the algorithm and computation associated to each epoch.

Let us analyze an epoch of a matched edge (u, v). Suppose this epoch got created by vertex v at level j. So v would have executed GENERIC-RANDOM-SETTLE and selected u as a random mate from level $< j$. Note that v must be owning less than 2^{j+1} edges and u would be owning at most 2^j edges at that moment. This observation and Lemma 4.4 imply that the computation involved in creation of the epoch is $O(2^j)$. Once the epoch is created, any update pertaining to u or v will be performed in just $O(1)$ time until the epoch gets terminated. Let us analyze the computation performed when the epoch gets terminated. At this moment either one or both u and v become free vertices. If v becomes free, v executes the following task (see procedure PROCESS-FREE-VERTICES in Figure 7) v scans all edges owned by it, which is less than 2^{j+1}, and disowns those edges incident from vertices of level j. Thereafter, if v still owns at least 2^j edges, it settles at level j and becomes part of a new epoch at level j. Otherwise, v keeps falling one level at a time. For a single fall of v from level i to $i-1$, the computation performed involves the following tasks: scanning the edges owned by v, disowning those incident from vertices at level i, incrementing ϕ_w values for each neighbor w of v lying at level less than i, and incrementing $\phi_v(i)$ by one. All this computation is of the order of the number of edges v owns at level i which is less than 2^{i+1}. Eventually either v settles at some level $k \geq 0$ and becomes part of a new epoch or it reaches level -1. The total computation performed by v is, therefore, of the order of $\sum_{i=k}^{j} 2^{i+1} = O(2^j)$. This entire computation involving v (and u) in this process is associated with the the epoch of (u, v). Hence we can state the following Lemma.

Lemma 4.6 *For any $i \geq 0$, the computation associated with an epoch at level i is $O(2^i)$.*

Let us now analyze the number of epochs terminated during any sequence of t updates. An epoch corresponding to edge (u, v) at level i could be terminated if the matched edge (u, v) gets deleted. However, it could be terminated by any of the following reasons also.

- u or v get selected as a random mate by one of their neighbors present at LEVEL $> i$.
- u or its mate starts owning 2^{i+1} or more edges.

Each of the above factors render the epoch to be an induced epoch. We shall assign the cost of each induced epoch to the epoch which led to the destruction of the former. To this objective, we now introduce the notion of computation *charged* to an epoch at any level i. Note that no epoch is created at level -1 as the vertices at level -1 are always free. If $i = 0$, the computation *charged* to the epoch is the actual computation performed during the epoch which is $O(1)$. For any level $i > 0$, the creation of an epoch causes destruction of at most two epochs at levels $< i$. It can be explained as follows: Consider an epoch at level i associated with an edge, say (u, v). Suppose it was created by vertex u. If u was already matched at level $j (j < i)$, let $w \neq v$ be its mate. Similarly, if v was also matched already, let $x \neq u$ be its current mate at level k. So matching u to v terminates the epoch of (u, w) and (v, x) at level j and k respectively. We *charge* the overall cost of these two epochs to the epoch of (u, v).

The computation charged to an epoch at level $i > 0$ is defined recursively as the actual computation cost of the epoch and the computation *charged* to at most two epochs destroyed by it at level $< i$. Let C_i be the computation charged to an epoch at level i. The epoch terminated by this epoch can be at level $i - 1$ in the

worst case. Also the computational cost associated with an epoch at level i is $c.2^i$ where c is a constant. So we get the following recurrence: $C_i \leq 2C_{i-1} + c.2^i$. This implies $C_i = O(i2^i)$.

Lemma 4.7 *The computation charged to a natural epoch at level i is $O(i2^i)$.*

Henceforth we just proceed along the lines of the analysis of our 2-LEVEL algorithm analogous to the proof of Lemma 3.3. Let $X_t(i)$ be the random variable denoting the number of *natural* epochs at level i terminated during a sequence of t_i updates. Recall that $X_{t_i} = \sum_u X_{|t_u(i)|}$, where $t_u(i)$ is the deletion sequence for u By substituting $M = 2^i$, in the the earlier analysis for two levels, using identical arguments, it follows that $E[X_t(i)] = \sum_u E[X_{|t_u(i)|}] = \sum_u O(t_u(i)/2^i) = O(t_i/2^i)$.

Note that Lemma 3.5 shows that $Pr[X_t = q] \leq \left(\frac{4et}{qM}\right)^{q/2}$, where $M = \sqrt{n}$. Similarly by using $M = 2^i$, we can show that if t_i is the deletion sequence at level i and X_{t_i} is the random variable denoting the total number of epoch ending at level i, then $P[X_{t_i} = q] \leq \left(\frac{4et_i}{q2^i}\right)^{q/2}$. Similar to Lemma 3.6, we choose $q_0 = 4\ (\log n + 4et_i/2^i)$. For any $q \geq q_0$, $\mathbf{Pr}[X_{t_i} = q]$ is of the form b^q where base $b < 1/2$. Hence $\mathbf{Pr}[X_{t_i} \geq q_0]$ is bounded by a geometric series with the first term $< 2^{-q_0}$ and the common ratio less than $1/2$. Furthermore $q_0 > 4 \log n$, hence $\mathbf{Pr}[X_{t_i} \geq q_0]$ is bounded by $2/n^4$. Hence X_{t_i} is bounded by $O(t/2^i + \log n)$ with high probability.

Lemma 4.8 *The expected number of natural epochs terminated at level i is $O(t_i/2^i)$ and $O(t_i/2^i + \log n)$ with high probability.*

It thus follows from Lemma 4.7 and Lemma 4.8 that the computation *charged* to all natural epochs terminated at level i is $O(it_i)$ in expectation. Summing up for all levels, the expected total number of natural epoch = $\sum_{i=1}^{\log n} it_i \leq \log n \sum_{i=1}^{\log n} t_i = O(t \log n)$. Similarly the total number of epochs at level i is $O(it_i + i2^i \log n)$ with high probability. Summing up for all the levels, and using the union bound, the total number of epochs over all the $\log n$ levels = $\sum_{i=1}^{\log n}(it_i + i2^i \log n) \leq t \log n + \log^2 n \sum_{i=1}^{\log n} 2^i \leq O(t \log n + n \log^2 n)$ with high probability. We can summarize as follows.

Lemma 4.9 *For any sequence of t updates, the computation* charged *to all the natural epochs which get terminated is $O(t \log n)$ in expectation and $O(t \log n + n \log^2 n)$ with high probability.*

Let us now analyze the cost *charged* to all those epochs which are alive at the end of t updates. Consider an epoch (say epoch of (u, v) at level i) such that (u, v) is still in the matching after t updates.

By Lemma 4.7, computation charged to epoch of (u, v) is $O(i2^i)$. If $\mathcal{O}_v^{init}$ was the edges owned by v at the start of the epoch, then since $|\mathcal{O}_v^{init}| \geq 2^i$ at the start of the epoch. We can say that the computation charged is $O(i|\mathcal{O}_v^{init}|) \leq O((\log n)t_v)$ where t_v is the total updates on v. Since a vertex can be part of only one live epoch, the total computation cost charged to all the live epochs is $\sum_v O(t_v \log n) = O(t \log n)$.

Hence we can conclude the following result.

Theorem 4.1 *Starting from an empty graph on n vertices, a maximal matching in the graph can be maintained over any sequence of t insertion and deletion of edges in $O(t \log n)$ time in expectation and $O(t \log n + n \log^2 n)$ time with high probability.*

5 Conclusion

We presented a fully dynamic algorithm for maximal matching which achieves expected amortized $O(\log n)$ time per edge insertion or deletion. Maximal matching is also 2-approximation of maximum matching.

Our experiments show that for most of the inputs, the matching maintained is very close to the maximum matching. But it is not hard to come up with update sequence such that at the end of the sequence, the matching obtained is strictly half the size of maximum matching. We present one such example. Let $G(V \cup U, E)$ be a graph such that $V = \{v_1, v_2, \ldots, v_n\}$ and $U = \{u_1, u_2, \ldots, u_n\}$. Consider the following update sequence. In the first phase, add edges between two points (v_i, v_j) if there is no edge between them. Eventually this process ends when there is a complete graph on vertices of V. In the second phase, add edge (v_i, u_i) for all i. Note that the degree of each u_i is one at the end of the updates. Let us now find the matching which our algorithm maintains. After the first phase of update ends, we have a complete graph on V. At that moment, we claim that the size of matching obtained by our algorithm is n. Indeed, size of any maximal matching on a complete graph of size n is n. Let (v_i, v_j) be an edge in the matching after phase 1. Note that both these endpoints are at a level greater than -1. A vertex in U is at level -1 as it does not have any adjacent edges after phase 1. When an edge (u_i, v_i) is added, since v_i is at a higher level than u_i, v_i becomes the owner of this edge. In the worst case, the second invariant of v_i is not violated after this edge deletion and nothing happens at this update step and u_i still remains at level -1. Using same reasoning, we can show that u_j also remains at level -1 after the addition of edge (v_j, u_j). Since both u_i and u_j are free after these edge updates, there is a 3-augmenting path passing through edge (v_i, v_j). In general, there will be a vertex disjoint 3-augmenting passing through all the matched vertices. This implies that the maximum matching is $2n$ after phase 2. But size of matching of our algorithm is n at that point.

It would be a challenging problem to see if c-approximate maximum matching for $c < 2$ can also be maintained in $O(\log n)$ update time. In particular, for maintaining a 3/2-approximate matching, we have to take care of all the 3-augmenting path. It is not clear how to extend our algorithm to handle 3-augmenting paths.

6 Acknowledgment

We thank Pankaj K. Agarwal for his valuable feedback on the presentation of the paper.

References

[1] David Alberts and Monika Rauch Henzinger. Average case analysis of dynamic graph algorithms. In *SODA*, pages 312–321, 1995.

[2] Monika Rauch Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, 1999.

[3] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. In *STOC*, pages 79–89, 1998.

[4] Zoran Ivkovic and Errol L. Lloyd. Fully dynamic maintenance of vertex cover. In *WG '93: Proceedings of the 19th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 99–111, London, UK, 1994. Springer-Verlag.

[5] Krzysztof Onak and Ronitt Rubinfeld. Maintaining a large matching and a small vertex cover. In *STOC*, pages 457–464, 2010.

[6] Piotr Sankowski. Faster dynamic matchings and vertex connectivity. In *SODA*, pages 118–126, 2007.

[7] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51:122–144, 2004.

[8] Michal Parnas and Dana Ron. Approximating the minimum vertex cover in sublinear time and a connection to distributed algorithms In *Theor. Comput. Sci.*, volume 381, number 1-3, pages 183-196, 2007.