

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

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 [6] 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 $\text{polylog}(n)$ 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 [9] 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 $\text{polylog}(n)$ 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 [5]. So approximation appears to be inevitable if we wish to achieve really fast update time for maintaining matching. Recently, Onak and Rubinfeld [6] 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 graph at any moment. Every edge of \mathcal{M} is called a *matched* edge and an edge in $E \setminus \mathcal{M}$ 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, all that is required is 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

[†] Presently at IBM, India Research Lab, New Delhi on leave from IIT Delhi.

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 is 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 $O\left(\frac{\deg(u)+\deg(v)}{\deg(u)/2}\right)$. 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 final 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 [6] 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 [6] 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 $a > 1$. 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 [6]. Our algorithm takes a vertex centric approach as described above for maintaining matching at each level. Our algorithm achieves significantly better results than [6], 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 pairwise independent random numbers.

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 which 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 two invariants throughout.

- 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.

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 will firstly involve 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 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 $\text{LEVEL}(u) = 1$, 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).

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
   $x \leftarrow \text{RANDOM-SETTLE}(u)$ ;
  if  $x \neq \text{NULL}$  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  $\mathcal{O}_u$ ;
if  $v$  is matched then
   $x \leftarrow \text{MATE}(v)$ ;
   $\mathcal{M} \leftarrow \mathcal{M} \setminus \{(v, x)\}$ 
else
   $x \leftarrow \text{NULL}$ ;
 $\mathcal{M} \leftarrow \mathcal{M} \cup \{(u, v)\}$ ;
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 $\text{LEVEL}(u) = \text{LEVEL}(v) = 0$.

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 insertion of (u, v) also leads to increase $|\mathcal{O}_u|$ 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 $|\mathcal{O}_u| = \sqrt{n}$, the 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 . 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 both the invariants for v were satisfied before the current edge update, it follows that the second invariant for v still remains intact.

Handling deletion of an edge: Let (u, v) be an edge that is deleted. If $(u, v) \notin \mathcal{M}$, both invariants are still intact. 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).

Procedure HANDLING-DELETION (u, v)

```

foreach  $(u, w) \in \mathcal{O}_u$  and LEVEL( $w$ ) = 1 do
  move  $(u, w)$  from  $\mathcal{O}_u$  to  $\mathcal{O}_w$ ;
if  $|\mathcal{O}_u| \geq \sqrt{n}$  then
   $x \leftarrow \text{RANDOM-SETTLE}(u)$ ;
  if  $x \neq \text{NULL}$  then NAIVE-SETTLE( $x$ );
else
  LEVEL( $u$ )  $\leftarrow$  0;
  NAIVE-SETTLE( $u$ );
  foreach  $(u, w) \in \mathcal{O}_u$  do
    if  $|\mathcal{O}_w| \geq \sqrt{n}$  then
       $x \leftarrow \text{RANDOM-SETTLE}(w)$ ;
      if  $x \neq \text{NULL}$  then NAIVE-SETTLE( $x$ );

```

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

Firstly 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

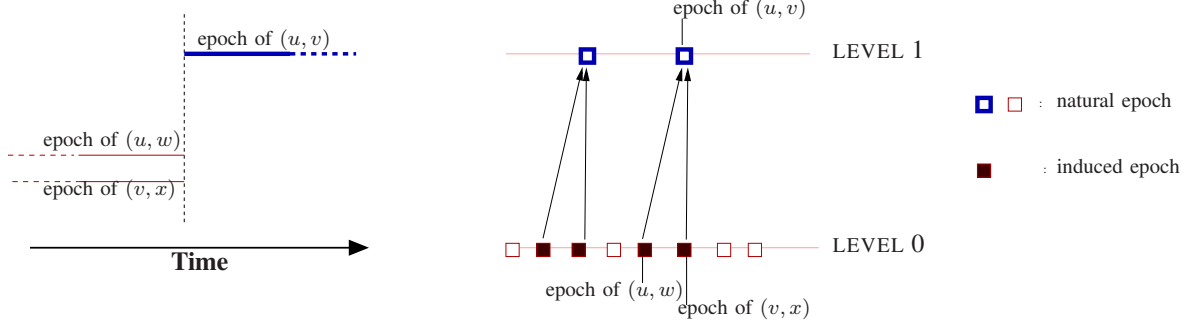


Figure 3. Epochs at level 0 and 1; the creation of an epoch at level 1 can destroy at most two epochs at level 0.

for each dirty neighbor w (that is, $|\mathcal{O}_w| \geq \sqrt{n}$), we execute $\text{RANDOM-SETTLE}(w)$ 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 creating 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 make transition to level 0.

Remark 3.1: After every update, the two endpoints of each matched edge will be present at the same level. In other words, edges having endpoints at distinct levels will never be part of the maximal matching \mathcal{M} maintained by our algorithm.

3.1. Analysis of the algorithm

We analyze the algorithm using the concept of *matched 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. 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 duration of the epoch associated with (u, v) will be the number of updates which u or v undergo during the epoch. The entire life span of an edge (u, v) would consist of a sequence of (matched) epochs of (u, v) separated by the continuous periods when (u, v) is not matched.

In order to bound the expected computation time per update, first we calculate the computation involved in an epoch at level 0 and 1.

Consider an edge $(u, v) \in \mathcal{M}$ at any moment. If $\text{LEVEL}(u, v) = 0$, then the creation of the epoch associated with (u, v) involves scanning \mathcal{O}_u (or \mathcal{O}_v) to find the free vertex v . An equivalent amount of work is required at the termination of this epoch. Being at level 0, u as well as v own less than \sqrt{n} edges each. So the total computation involving an epoch at level 0 is $O(\sqrt{n})$. Let us consider the case when $\text{LEVEL}(u, v) = 1$. Suppose the epoch got created by vertex u .

If the epoch was created when u rises from level 0 to level 1, then $|\mathcal{O}_u|$ is exactly equal to \sqrt{n} , and $|\mathcal{O}_v|$ is at most \sqrt{n} . Creation of this epoch would require transfer of u (as well as v) from level 0 to level 1, eliminating (u, x) from \mathcal{O}_x for each $(u, x) \in \mathcal{O}_u$ followed by eliminating (v, x) from \mathcal{O}_x for each $(v, x) \in \mathcal{O}_v$. So the total computation performed during creation of the epoch is $O(\sqrt{n})$.

Another way for the creation of an epoch at level 1 is when the previous epoch of u (say epoch of (u, w)) is destroyed at level 1. As a result, u disowns edges at level 1 and if it still has at least \sqrt{n} edges it selects a random vertex v and starts (new) epoch of (u, v) . As we can see that there are two tasks involved in this process. The first task involves disowning the edges of u ; the computational cost of this task is attributed to the termination of the old epoch of (u, w) . The other task involves transfer of v from level 0 to level 1, and eliminating (v, x) from \mathcal{O}_x for each $(v, x) \in \mathcal{O}_v$. The computational cost of this task is attributed to the creation of the epoch of (u, v) . Since \mathcal{O}_v is at most \sqrt{n} , the computational cost performed during the creation of the epoch of (u, v) in this case is again $O(\sqrt{n})$.

At the termination of the epoch, the computation involves disowning edges incident on u and v from vertices at level 1. This computation is of the order of $|\mathcal{O}_u| + |\mathcal{O}_v|$ which can be bounded by $O(n)$. Excluding the two updates that cause creation and termination 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 performed at the time of creation and termination of an epoch. From our analysis, it follows that the amount of computation involved in an epoch at level 1 and level 0 are $O(n)$ and $O(\sqrt{n})$ respectively.

An epoch corresponding to some edge, say (u, v) , gets terminated 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 is terminated due to cause (i); otherwise it will be called an *induced* epoch. *Induced* epoch can be seen as a premature termination of an

epoch since, unlike natural epoch, the matched edge is not actually deleted from the graph when an *induced* epoch is terminated.

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) which destroys them. So the overall computation *charged* to an epoch of (u, v) at level 1 is $O(n + |\mathcal{O}_x| + |\mathcal{O}_w|)$. This cost is indeed $O(n)$ since each of $|\mathcal{O}_x|$ and $|\mathcal{O}_w|$ is less than \sqrt{n} .

Lemma 3.2: 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 get destroyed during the updates and the epochs which are alive at the end of all the updates.

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 3.2 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. Let us define the *duration* of the epoch as the number of edges incident on u which are deleted during the epoch. Consider an epoch at level 1 created by some vertex, say u . At the time of its creation u must be owning a set of at least \sqrt{n} edges, and u selected a matched edge out of its owned edges uniformly randomly and independent of other epochs. This implies the following lemma.

Lemma 3.3: The probability that a given epoch at level 1 has duration at most i is bounded by $\frac{i}{\sqrt{n}}$.

Let X_t be the random variable denoting the number of epochs at level 1 destroyed during a sequence of t updates.

Lemma 3.4: For any given $q \geq 1$,

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

Proof: If there are q epochs destroyed during t updates, at least half of them have duration $\leq 2t/q$. Hence, $\Pr[X_t = q]$ is bounded by the probability that there are at least $q/2$ epochs of duration at most $\frac{2t}{q}$. So, using Lemma 3.3 and exploiting the independence among the epochs,

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

For the last inequality we used $\binom{q}{q/2} \leq \left(\frac{eq}{q/2}\right)^i = (2e)^i$. ■ We use Lemma 3.4 to analyze $\mathbf{E}[X_t]$ and deviation of X_t .

Lemma 3.5: For any $t > 0$, $\mathbf{E}[X_t] = O(t/\sqrt{n})$.

Proof: Let us set $q_0 = 5et/\sqrt{n}$. For any $q > q_0$, it follows from Lemma 3.4 that $\Pr[X_t = q] \leq (4/5)^{q/2}$. Hence,

$$\Pr[X_t \geq q] = \sum_{q' \geq q} \Pr[X_t = q'] < 10 \left(\frac{4}{5}\right)^{q/2} \quad (1)$$

Now we use the following well known equality which holds since X_t takes nonnegative integral values only.

$$\mathbf{E}[X_t] = \sum_{q \geq 1} \Pr[X_t \geq q] \leq q_0 + \sum_{q > q_0} \Pr[X_t \geq q] \quad (2)$$

Using Equations 1 and 2, it follows that $\mathbf{E}[X_t] = 5et/\sqrt{n} + O(1) = O(t/\sqrt{n})$. ■

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 3.4 that for any $q \geq q_0$, $\Pr[X_t = q]$ is of the form b^q where base $b < 1/2$. Hence $\Pr[X_t \geq q_0]$ 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 $\Pr[X_t \geq q_0]$ 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 Lemmas 3.5 and 3.6 rely heavily on the total independence of random numbers used for selecting random mates. However, similar bound on $\mathbf{E}[X_t]$ can be derived even if we assume only pairwise independence between the random numbers used (see Appendix for an alternate proof of Lemma 3.5).

Now, recall from Lemma 3.2 that each natural epoch destroyed at level 1 has $O(n)$ computation *charged* to it. So Lemmas 3.5 and 3.6, when combined together, imply the following lemma.

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. Note that each vertex is involved in at most one alive epoch. It thus follows that the computation cost *charged* to all the alive epochs at any instance is $O(t)$. 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 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. By following these guidelines, notice that the ratio of maximum to minimum edges owned by a vertex during an epoch gets reduced from \sqrt{n} to a constant, which is what we aimed for.

We pursue the approach sketched above and some new ideas 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

The fully dynamic algorithm maintains a partition of vertices among $\lfloor \log n \rfloor + 2$ levels. The levels are numbered

from -1 to $L_0 = \lfloor \log n \rfloor$. Note that the level starts from -1 and not 0 . We again use the notion of ownership of edges which is slightly different from the one used in 2-LEVEL algorithm. 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 suitably 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. Note that the onus of maintaining \mathcal{E}_u^i will not be on u . In fact, for any edge $(u, v) \in \mathcal{E}_u^i$, it will be v which 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 pays 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.

<p>For a vertex v with $\text{LEVEL}(v) = i$,</p> $\phi_v(j) = \begin{cases} \mathcal{O}_v + \sum_{i \leq k < j} \mathcal{E}_v^k & \text{if } j > i \\ 0 & \text{otherwise} \end{cases}$
--

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 two 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.

The second invariant implies that a vertex at level -1 will have no neighbor at level -1 . This fact together with the first invariant imply that the matching maintained by the algorithm will indeed be a maximal matching. In fact, similar to the 2-LEVEL algorithm, the endpoints of each matched edge will lie at the same level. Figure 4 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

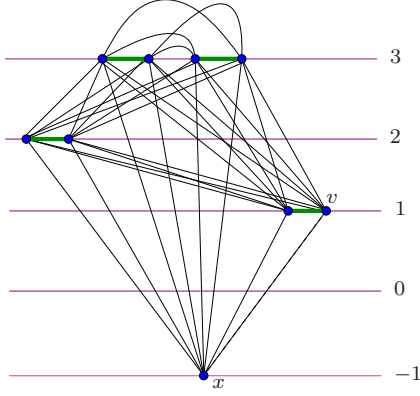


Figure 4. A snapshot of the algorithm on K_9 : all vertices are matched (thick edges) except vertex x at level -1. Vertex v is the owner of just the edge (v, x) . $\phi_v(2) = 2 < 2^2$ and $\phi_v(3) = 4 < 2^3$, so v cannot rise to a higher level.

and the algorithm basically restores these invariants. This may involve rise or fall of vertices. 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 capture this influence in precise words.

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

Proof: 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^ℓ . 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 at level k 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: 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- $\phi(v, I)$: this operation decrements $\phi_v(i)$ by one for all i in interval I . This operation will be executed when some neighbor of v rises.

- 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 is upper-bounded by the computation cost of all INCREMENT- $\phi()$ operations 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)

if LEVEL(u) < i **then** // u owns edges till it reaches level i

for each $j = \text{LEVEL}(u)$ to $i - 1$ **do**

for each $(u, w) \in \mathcal{E}_u^j$ **do**

 transfer (u, w) from \mathcal{E}_u^j to \mathcal{E}_w^i ;

 transfer (u, w) from \mathcal{O}_w to \mathcal{O}_u ;

 DECREMENT- $\phi(w, [j + 1, i])$;

Let (u, v) be a uniformly randomly selected edge from \mathcal{O}_u ;

if v is matched **then**

$x \leftarrow \text{MATE}(v)$;

$\mathcal{M} \leftarrow \mathcal{M} \setminus \{(v, x)\}$

else

$x \leftarrow \text{NULL}$

for each $j = \text{LEVEL}(v)$ to $i - 1$ **do** // v rises to level i and thus owns edges incident from vertices at levels LEVEL(v) to $i - 1$

for each $(v, w) \in \mathcal{E}_v^j$ **do**

 transfer (v, w) from \mathcal{E}_v^j to \mathcal{E}_w^i ;

 transfer (v, w) from \mathcal{O}_w to \mathcal{O}_v ;

 DECREMENT- $\phi(w, [j + 1, i])$;

$\mathcal{M} \leftarrow \mathcal{M} \cup \{(u, v)\}$;

LEVEL(u) $\leftarrow i$; LEVEL(v) $\leftarrow i$;

return x ;

Figure 5. 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 5. 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 $\in [\text{LEVEL}(u), i - 1]$. For each such edge (u, v) that is now owned by u , we perform $\text{DECREMENT-}\phi(v, [\text{LEVEL}(v) + 1, i])$ 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 \mathcal{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 executing $\text{GENERIC-RANDOM-SETTLE}(u, i)$ and selecting a mate v . Excluding the time spent in $\text{DECREMENT-}\phi$ 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 $\text{PROCESS-FREE-VERTICES}$. 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 $\text{PROCESS-FREE-VERTICES}$ restores the invariants of each such free vertex. 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. Hence nothing needs to be done except deleting the edge (u, v) from the data structures of u and v . Otherwise, let $k = \text{LEVEL}(u) = \text{LEVEL}(v)$. We execute Procedure $\text{PROCESS-FREE-VERTICES}(\langle\langle(u, k), (v, k)\rangle\rangle)$.

Handling insertion of an edge: Consider 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 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 $\text{GENERIC-RANDOM-SETTLE}(u, i_{\max})$, 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 $\text{PROCESS-FREE-VERTICES}(\langle\langle(x, k), (w, j)\rangle\rangle)$.

Remark 4.5: If the insertion of edge (u, v) violates the second invariant for both u and v , we select that vertex which can rise to the higher level to restore its invariant and process that vertex.

4.2.1. Description of Procedure $\text{PROCESS-FREE-VERTICES}$: The procedure receives a sequence S of ordered pairs (x, i) such that x is a free vertex at level

Procedure $\text{PROCESS-FREE-VERTICES}(S)$

```

for each  $(x, i) \in S$  do  $\text{ENQUEUE}(Q[i], x)$ ;
for  $i = L_0$  to  $0$  do
  while  $Q[i] \neq \emptyset$  do
     $v \leftarrow \text{DEQUEUE}(Q[i])$ ;
    if  $\text{FALLING}(v)$  then //  $v$  falls to  $i - 1$ 
       $\text{LEVEL}(v) \leftarrow i - 1$ ;
       $\text{ENQUEUE}(Q[i - 1], v)$ ;
       $\phi_v(i) \leftarrow |\mathcal{O}_v|$ ;
      for each  $u \in \mathcal{O}_v$  do
        transfer  $(u, v)$  from  $\mathcal{E}_u^i$  to  $\mathcal{E}_u^{i-1}$ ;
         $\text{INCREMENT-}\phi(u, i)$ ;
        if  $\phi_u(i) \geq 2^i$  then //  $u$  rises to  $i$ 
           $x \leftarrow \text{GENERIC-RANDOM-SETTLE}(u, i)$ ;
          if  $x \neq \text{NULL}$  then
             $\ell \leftarrow \text{LEVEL}(x)$ ;
             $\text{ENQUEUE}(Q[\ell], x)$ ;
      else //  $v$  settles at level  $i$ 
         $x \leftarrow \text{GENERIC-RANDOM-SETTLE}(v, i)$ ;
        if  $x \neq \text{NULL}$  then
           $\ell \leftarrow \text{LEVEL}(x)$ ;
           $\text{ENQUEUE}(Q[\ell], x)$ ;

```

Function $\text{FALLING}(v)$

```

 $i \leftarrow \text{LEVEL}(v)$ ;
for each  $(u, v) \in \mathcal{O}_v$  such that  $\text{LEVEL}(u) = i$  do
  //  $v$  disowns all edges at level  $i$ 
  transfer  $(u, v)$  from  $\mathcal{O}_v$  to  $\mathcal{O}_u$ ;
  transfer  $(u, v)$  from  $\mathcal{E}_u^i$  to  $\mathcal{E}_v^i$ ;
if  $|\mathcal{O}_v| < 2^i$  then return TRUE else return FALSE;

```

Figure 6. Procedure for processing free vertices given as a sequence S of ordered pairs (x, i) where x is a free vertex at $\text{LEVEL}i$.

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, this rise will be only to level i . After these rising vertices settle at level i , 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. Its complete pseudocode is given in Figure 6.

The procedure uses an array Q of size $L_0 + 2$, where $Q[i]$

is a pointer to a queue (initially empty). 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 i th 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 . In case 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 . Note that this 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. Let us discuss the more interesting case when 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 $Q[j]$. Based on the description of the procedure `PROCESS-FREE-VERTICES`, and using Lemmas 4.1 and 4.2, we can conclude the following.

Lemma 4.6: After i th 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 two invariants holds.

Lemma 4.6 establishes that after termination of procedure `PROCESS-FREE-VERTICES`, there are no free vertices at level ≥ 0 and the two invariants get restored globally.

4.3. Analysis of the algorithm

Processing 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, in our analysis we can safely ignore the computation involving `DECREMENT- $\phi()$` 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 involved with each epoch.

Let us analyze an epoch of a matched edge (u, v) . Let 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 one or both of u and v become free vertices. Let v becomes free. v executes the following task (see procedure `PROCESS-FREE-VERTICES` in Figure 6). 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 three tasks: scanning the edges owned by v , disowning those incident from vertices at level i , and incrementing ϕ_w values for each neighbor w of v lying at level less than i . 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 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 the computation associated with the epoch of (u, v) . Hence we can state the following Lemma.

Lemma 4.7: For any $i \geq 0$, the computation associated with an epoch at level i is of the order of 2^i .

Remark 4.8: As shown in Procedure `PROCESS-FREE-VERTICES`, when vertex v falls from level i to $i - 1$, the instruction “ $\phi_v(i) \leftarrow |\mathcal{O}_v|$ ” is executed. For the sake of analysis, this instruction can be viewed as a total of $|\mathcal{O}_v|$ increment operations on $\phi_v(i)$ starting from 0. Note that this does not increase the asymptotic bound obtained in Lemma 4.7. Even though $\phi_v(i)$ is not initialized by increment operation, by analyzing the above instruction in this way we can claim that Observation 4.1 still holds.

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 $\text{LEVEL} > i$.
- u or its mate starts owning 2^{i+1} or more edges.

Each of the above factors render the epoch induced. We

shall assign the cost of each induced epoch to the epoch which led to the destruction of the former. For 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 $< i$. 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 $< i$. This definition and Lemma 4.7 immediately lead to the following lemma.

Lemma 4.9: Computation charged to an epoch at level i is $O(i2^i)$.

Henceforth we just proceed along the lines of the analysis of our 2-LEVEL algorithm. We need to calculate the computation *charged* to all the natural epochs that are created during any sequence of t updates. Let us define level of an update, say insertion or deletion of edge (u, v) , as $\max(\text{LEVEL}(u), \text{LEVEL}(v))$. We partition all the t updates into their respective levels. Let t_i edge deletions among these t updates occur at level i . The proof of the following lemma proceeds exactly along the lines of Lemma 3.5.

Lemma 4.10: The number of natural epochs terminated at level i is $O(t_i/2^i)$ on expectation and $O(t_i/2^i + \log n)$ with high probability.

It thus follows from Lemma 4.9 and Lemma 4.10 that the computation *charged* to all natural epochs terminated at level i is $O(it_i)$ in expectation and $O(it_i + i2^i \log n)$ with high probability. Summing up for all the levels, we can conclude the following lemma.

Lemma 4.11: 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.

The total computation cost associated with t updates is equal to the computation cost *charged* to all the natural epochs. It follows from Lemma 4.11 that the expected computation *charged* to all natural epochs which gets terminated during t updates is $O(t \log n)$. The computation *charged* to all the natural epochs which are alive at the end of t updates is anyway bounded by $O(t \log n)$. 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. 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.

6. ACKNOWLEDGMENT

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

REFERENCES

- [1] D. Albers and M. R. Henzinger, "Average case analysis of dynamic graph algorithms," in *SODA*, 1995, pp. 312–321.
- [2] M. R. Henzinger and V. King, "Randomized fully dynamic graph algorithms with polylogarithmic time per operation," *J. ACM*, vol. 46, no. 4, pp. 502–516, 1999.
- [3] J. Holm, K. de Lichtenberg, and M. Thorup, "Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity," *J. ACM*, vol. 48, no. 4, pp. 723–760, 2001.
- [4] Z. Ivkovic and E. L. Lloyd, "Fully dynamic maintenance of vertex cover," in *WG '93: Proceedings of the 19th International Workshop on Graph-Theoretic Concepts in Computer Science*. London, UK: Springer-Verlag, 1994, pp. 99–111.
- [5] S. Micali and V. V. Vazirani, "An $O(\sqrt{|V|}|E|)$ algorithm for finding maximum matching in general graphs," in *FOCS*, 1980, pp. 17–27.
- [6] K. Onak and R. Rubinfeld, "Maintaining a large matching and a small vertex cover," in *STOC*, 2010, pp. 457–464.
- [7] R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.
- [8] M. Parnas and D. Ron, "Approximating the minimum vertex cover in sublinear time and a connection to distributed algorithms," *Theor. Comput. Sci.*, vol. 381, no. 1-3, pp. 183–196, 2007.
- [9] P. Sankowski, "Faster dynamic matchings and vertex connectivity," in *SODA*, 2007, pp. 118–126.

APPENDIX

Alternate proof for Lemma 3.5 : Consider the first epoch which is terminated during a sequence of t updates. Let at the moment of its creation, its owner owned M edges. It can be seen that its duration is a random variable Z distributed uniformly in the range $[1..M]$. Using the law of conditional expectation, that is, $\mathbf{E}[X_t] = \mathbf{E}[\mathbf{E}[X_t|Z]]$, and setting $y_t = \mathbf{E}[X_t]$, we obtain the following useful recurrence.

$$y_t = \begin{cases} \frac{1}{M} \sum_{i=1}^M (y_{t-i} + 1) & \text{for } t \geq M \\ \frac{1}{M} \sum_{i=1}^t (y_{i-1} + 1) & \text{for } 0 < t < M \end{cases} \quad (3)$$

where $y_0 = 0$ and $y_1 = \Pr[Z = 1] = 1/M$. By subtracting the recurrence of y_{t-1} from y_t , we obtain $y_i \leq (1 + \frac{1}{M})^M - 1 \leq e$ for $i \leq M$. Using $y_i \geq y_{i-1}$, one can verify that the solution for the recurrence in Equation 3 is given by $y_t \leq e + c \frac{t}{M}$ for $c \geq 2$.

Notice that this proof does not rely on the independence of the random numbers.