Improved Algorithm for Dynamic b-Matching

Sayan Bhattacharya[1], **Manoj Gupta**[2], **and Divyarthi Mohan**[3]

1 **University of Warwick, UK**
`s.bhattacharya@warwick.ac.uk`
2 **IIT Gandhinagar, India**
`gmanoj@iitgn.ac.in`
3 **Princeton University, USA**
`dm23@cs.princeton.edu`

Abstract

Recently there has been extensive work on maintaining (approximate) maximum matchings in dynamic graphs. We consider a generalisation of this problem known as the *maximum b-matching*: Every node v has a positive integral capacity b_v, and the goal is to maintain an (approximate) maximum-cardinality subset of edges that contains at most b_v edges incident on every node v. The maximum matching problem is a special case of this problem where $b_v = 1$ for every node v.

Bhattacharya, Henzinger and Italiano [ICALP '15] showed how to maintain a $O(1)$ approximate maximum b-matching in a graph in $O(\log^3 n)$ amortised update time. Their approximation ratio was a large (double digit) constant. We significantly improve their result both in terms of approximation ratio as well as update time. Specifically, we design a randomised dynamic algorithm that maintains a $(2+\epsilon)$-approximate maximum b-matching in expected amortised $O(1/\epsilon^4)$ update time. Thus, for every constant $\epsilon \in (0,1)$, we get expected amortised $O(1)$ update time. Our algorithm generalises the framework of Baswana, Gupta, Sen [FOCS '11] and Solomon [FOCS '16] for maintaining a maximal matching in a dynamic graph.

1998 ACM Subject Classification F.2 Analysis of Algorithms and Problem Complexity

Keywords and phrases dynamic data structures, graph algorithms

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

In dynamic graph algorithms, we want to maintain a solution to an optimisation problem on an input graph that is undergoing a sequence of edge insertions/deletions. The time taken to modify the solution after an *update* (edge insertion/deletion) is called the *update time* of the dynamic data structure. The challenge is to design dynamic data structures whose update times are significantly faster than recomputing the solution from scratch after each update using the best static algorithm. In recent years, there has been extensive work on dynamic graph algorithms for (approximate) maximum matching [10, 1, 12, 8, 2, 9, 3, 11, 6, 5]. A matching $M \subseteq E$ in a graph $G = (V, E)$ is a subset of edges that do not share a common endpoint. In this problem, the goal is to maintain a matching of (approximately) maximum size in an input graph undergoing a sequence of edge insertions/deletions.

The first significant result on dynamic matching was due to Onak and Rubinfeld [10], who gave a randomised data structure with approximation ratio $O(1)$ and amortised update time of $O(\log^2 n)$. Baswana, Gupta and Sen [1] improved this approximation ratio to 2 and the amortised update time to $O(\log n)$. Very recently, in a breakthrough result Solomon [12] built upon the algorithmic framework of Baswana, Gupta and Sen [1] to present a randomised data structure with approximation ratio 2 and *constant amortised update time*. This is the first paper to achieve constant (and hence optimal) update time for *any* graph problem. All the data structures described so far are randomised. There are deterministic data structures for this problem, with $(2+\epsilon)$-approximation ratio and $O(\text{poly} \log n)$ amortised update time, that are due to Bhattacharya, Henzinger and Italiano [5] and Bhattacharya, Henzinger and Nanongkai [6]. They show how to maintain a $(2+\epsilon)$-approximate maximum *fractional* matching in $O(\log n/\epsilon^2)$ amortised update time. Then they *round* these fractional weights deterministically in a dynamic setting to get an integral matching.

We focus on a generalisation of the dynamic matching problem. We are given an input graph $G = (V, E)$ with $n = |V|$ nodes. At each time-step, an edge is inserted into/deleted from the graph. Each node $v \in V$ has a positive integral *capacity* b_v. The node-set V and the capacities $\{b_v\}$ remain unchanged over time. A b-matching is a subset of edges $\mathcal{M} \subseteq E$ that contains at most b_v edges incident on every node $v \in V$. Note that if $b_v = 1$ for every node $v \in V$, then $\mathcal{M}$ is a matching. Our goal is to maintain a b-matching of (approximately) maximum size in this dynamic graph G. This problem was first considered by Bhattacharya, Henzinger and Italiano [4]. Extending the dynamic data structure for fractional matching [5], they first showed how to maintain a fractional b-matching. Then they randomly sampled the edges from this fractional b-matching, with probabilities proportional to their edge-weights, and got an integral b-matching. This leads to a randomised data structure with a (large, double digit) constant approximation ratio and $O(\log^3 n)$ amortised update time with high probability. Recently, Gupta et. al [7] improved the update time in [4] for the set cover problem. However, their result does not imply our b-matching result.

Our result. We significantly improve upon the previous result on dynamic b-matching by Bhattacharya et al. [4], both in terms of the approximation ratio and update time. Specifically, we present a randomised data structure with approximation ratio $(2+\epsilon)$ and expected amortised update time $O(1/\epsilon^4)$. Thus, for small constant ϵ, we get an approximation ratio that is arbitrarily close to 2, and constant (optimal) update time.

Our technique. We build upon the work of Baswana, Gupta, Sen [1] and Solomon [12]. Before proceeding any further, we briefly review their framework for dynamic matching. A matching $M \subseteq E$ is *maximal* if every unmatched edge $(u, v) \in E \setminus M$ has at least one matched endpoint. The size of a maximal matching is a 2-approximation to the size of the maximum matching. Note that we can easily maintain a maximal matching in a dynamic graph in $O(d)$ update time, where d is an upper bound on the maximum degree of a node. This holds since after the insertion/deletion of an edge (u, v), we only need to check the neighbours of u and v to preserve maximality, and there are at most $O(d)$ such neighbours. To improve the update time, the papers [1, 12] use the following idea. We pick a neighbour y of x uniformly at random from the set N_x, where N_x is the set of all neighbours of x, and add the edge (x, y) to the matching M. If the sequence of edge insertions/deletions in G is oblivious to the random bits used by the algorithm, then in expectation this edge (x, y) will not be deleted from G before half of the current edges in N_x gets deleted. And when such an edge $(x, z) \in N_x \setminus \{(x, y)\}$ gets deleted, the algorithm does not need to perform any computation for the node x (since the node remains matched in M). Thus, although the algorithm takes $O(|N_x|)$ time to find a new *mate* for x, it does not need to do anything for the next $|N_x|/2$ edge deletions incident on x, in expectation. This helps bring down the amortised update time to $O(1)$. This is the high level idea behind the dynamic algorithms in [1, 12]. The actual algorithms, however, are significantly more complicated, for the following reason. In the preceding summary, we said that the node x picks a neighbour y uniformly at random from the set N_x. But what if the node y itself is matched to some other node z (i.e., $(y, z) \in M$)? In that case, in order to add the edge (x, y) to the matching M, we first need to unmatch the edge (y, z) by removing it from M. But this means that we will now need to find a new mate for z. In general, this can lead to a long chain of edges alternately being matched and unmatched. To address this concern, the papers [1, 12] construct a *hierarchical partition* of the node-set into $O(\log n)$ levels. Suppose that a node x is at level i, and want to find a new mate for x. The papers [1, 12] ensure that x has roughly α^i many neighbours at strictly lower levels $\{0, \ldots, i-1\}$, for some constant $\alpha > 1$. We now pick a node y uniformly at random *only from* these neighbours of x at levels $j < i$. The papers further ensure that every matched edge has both its endpoints at the same level. Hence, the node (say z) that is matched to y will have $\ell(z) = \ell(y) < i = \ell(x)$. We now match the edge (x, y), bring the node y up to level i, and unmatch the edge (y, z). The node z now has to find a new mate for itself. But note that the level of z is strictly less than the level of x. Hence, this chain of alternate matchings and unmatchings of edges cannot go on for more than $O(\log n)$ levels.

In order to extend the above framework to maximum b-matching, we have to overcome several technical difficulties. First, since a node can have multiple matched edges incident on it in a b-matching, we can no longer ensure that both the endpoints of every matched edge are on the same level. Instead, we maintain an invariant which states that if a node v has b_v matched edges incident on it, then at least one of these matched edges, say (u, v), must have its other endpoint u at a level that is not larger than the level of v (see Invariant 3). Second, unlike the papers [1, 12], we can no longer ensure that if a node z becomes unmatched while we are finding a new mate for a different node x, then $\ell(x) > \ell(z)$. Indeed, there are instances in our algorithm where both x and z can be at the same level, and we need to show that this situation does not occur too often. Finally, in the papers [1, 12], at any point in time there can be multiple nodes in the hierarchical partition that are *dirty*, meaning that they violate one of the invariants. The algorithms in [1, 12] run a WHILE loop, and each iteration of the WHILE loop picks *any arbitrary* dirty node and restores all the invariants it ought to satisfy (this can lead to other nodes becoming dirty). In sharp contrast, the WHILE loop in our dynamic algorithm has to pick the dirty nodes *in a specific order*, preferring one type of dirty nodes over another. This preferential ordering among the dirty nodes turns out to be crucial in analysing the amortised update time of our algorithm. See Sections 2 and 3 for details.

2 Our Dynamic Algorithm for Maximum b-Matching

Let $G = (V, E)$ denote the input graph whose edge-set E changes dynamically. Every node $v \in V$ has a positive integral *capacity* b_v associated with it. The node-set V and the capacities $\{b_v\}$ do not change over time. Let $n = |V|$ be the number of nodes in G. A subset of edges $\mathcal{M} \subseteq E$ is a *b-matching* iff for all nodes $v \in V$, we have $|\{(u, v) \in \mathcal{M} | \ (u, v) \in E\}| \leq b_v$. We say that an edge $e \in E$ is *matched* in $\mathcal{M}$ iff $e \in \mathcal{M}$.

Throughout this paper, we fix any constant $\epsilon \in (0, 1/2)$ and define $\alpha = 5/\epsilon$. We will maintain a partition of the node-set V into $L + 2$ *levels* $\{-1, 0, \ldots, L\}$, where $L = \lceil \log_\alpha n \rceil$. Let $\ell(v) \in \{-1, \ldots, L\}$ denote the level of a node v. The value of $\ell(v)$ changes over time. In our dynamic algorithm, whenever a node v moves to a level j, we will require that it has at most $2b_v \cdot \alpha^{j+1}$ neighbours u with $\ell(u) \leq j$. Further, any node v at level -1 has at most $O(b_v \cdot \alpha^0) = O(b_v)$ many neighbours at level -1.

The *level of an edge* is the maximum level among its endpoints, i.e., $\ell(u, v) = \max(\ell(u), \ell(v))$ for all $(u, v) \in E$. For every node $v \in V$ and every level $j \in [-1, L]$, we let $\mathcal{E}_v^j = \{(u, v) \in E : \ell(u, v) = j\}$ be the set of edges incident on v that are at level j. Since $\ell(u, v) \geq \ell(v)$ for all edges $(u, v) \in E$, it follows that $\mathcal{E}_v^j = \emptyset$ for all levels $j < \ell(v)$. We say that an edge $(u, v) \in E$ is *owned by* its endpoint v iff $\ell(u) < \ell(v)$. Equivalently, we say that v is the *owner* of the edge (u, v). We let $\mathcal{O}_v = \{(u, v) \in E : \ell(u) < \ell(v)\}$ denote the set of edges owned by $v \in V$ (Note that if both the end-points are at the same level then no vertex owns the edge). In a b-matching $\mathcal{M} \subseteq E$, we let $\mathcal{M}_v = \{(u, v) \in \mathcal{M}\}$ denote the set of all matched edges incident on a node $v \in V$. For any level $i \in [-1, L]$, we define $\mathcal{M}_v(i) = \{(u, v) \in \mathcal{M}_v : \ell(u, v) = i\}$ as the set of all matched edges incident on a node v at level i. We say that a node $v \in V$ is Full in a b-matching $\mathcal{M} \subseteq E$ iff $|\mathcal{M}_v| = b_v$, and v is Deficient in $\mathcal{M}$ iff $|\mathcal{M}_v| < (1 - \epsilon)b_v$. The node v is Non-deficient in $\mathcal{M}$ iff $|\mathcal{M}_v| \geq (1 - \epsilon)b_v$. Finally, for every node $v \in V$, if v is Full, then we define Base(v) to be the smallest level j for which there is a matched edge $(u, v) \in \mathcal{M}_v$ at level $\ell(u, v) = j$. Else if v is not Full, then we set Base$(v) = \ell(v)$.

We maintain a b-matching $\mathcal{M} \subseteq E$ in G satisfying the three invariants below. The approximation guarantee of the algorithm follows from the first two invariants (see Theorem 4). Invariant 3 will be useful in bounding the amortised update time. The main result in this paper follows from Theorems 4, 5.

▶ Invariant 1. Every node $v \in V$ at level $\ell(v) \geq 0$ is non-deficient.

▶ Invariant 2. Every unmatched edge with level -1 has at least one Non-deficient endpoint.

▶ Invariant 3. For every node $v \in V$, we have $|\{(u, v) \in \mathcal{M} : \ell(u) > \ell(v)\}| < b_v$.

We will assume that the initial graph is empty. So, all the three invariants hold. We now compare our invariants with those in ([1], [12]), that is in the maximal matching case when $b_v = 1$ for all $v \in V$. non-deficient vertices are *free* vertices in this simple case. The first invariant then say that all the vertices at level ≥ 0 are matched . The second invariant says that each un-matched edge at level -1 has both the end point free. In ([1],[12]), there can be no edge at level -1, since only *free* vertices settle at level -1. That is why the corresponding invariant for maximal matching is that *there is no edge with level -1.* We have generalized this invariant for b-matching. The last invariant implies that if at any point of time all the other endpoints of the matched edges incident on v have levels greater than v, then v has to move to a higher level. In the simple maximal matching case, this implies that end-points of the matched edges should always be at the same level. The reader can check that these three invariants are maintained in ([1],[12]), and here we generalize them for b-matching.

▶ **Theorem 4.** *Invariants 1, 2 imply that $\mathcal{M}$ is a $(2 + \epsilon)$-approximate maximum b-matching.*

Proof. (Sketch) Invariants 1, 2, ensure that $\mathcal{M}$ is *almost maximal*, as every unmatched edge has one endpoint x with $|\mathcal{M}_x| \geq (1 - \epsilon)b_x$. This implies $(2 + \epsilon)$-approximation. ◀

▶ **Theorem 5.** *There is a randomised algorithm that maintains a b-matching in a dynamic graph satisfying Invariants 1, 2, 3. It has an amortised update time of $O(1/\epsilon^4)$ in expectation.*

2.1 Handling the insertion/deletion of an edge

Data structures. Each node $x \in V$ maintains the edge-sets $\mathcal{M}_x, \mathcal{M}_x(j), \mathcal{E}_x^j$ and $\mathcal{O}_x$ as doubly linked lists, and the node also maintains the sizes of these sets. From the size of $\mathcal{M}_x$, we can infer whether the node x is Deficient, Non-Deficient or Full. With appropriate pointers, an edge can be inserted into or deleted from a linked list in $O(1)$ time. Further, each node $x \in V$ maintains its level $\ell(x)$ and the value of Base(x).

Insertion/deletion of an edge (u, v). When an edge (u, v) is inserted into or deleted from the graph $G = (V, E)$, we first update the relevant data structures in $O(1)$ time. For the time being, to highlight the main idea behind our algorithm, we assume that the insertion/deletion of the edge (u, v) does not lead to a violation of Invariant 2 (which pertains to the subgraph induced by the nodes at level -1). In Section 2.1.4, we will show how to handle the nodes in level -1 and how to maintain Invariant 2. Right now we focus on maintaining the remaining two invariants. Accordingly, suppose that after the insertion/deletion of the edge (u, v), at least one of its endpoints violates Invariant 1 or 3. In general, if a node $x \in V$ violates Invariant 1 or 3, then we say that the node x is DIRTY. A dirty node is either DEFICIENT (if it violates Invariant 1) or FULL-FROM-ABOVE (if it violates Invariant 3). Thus, a node x is FULL-FROM-ABOVE iff BASE$(x) > \ell(x)$ and $|\mathcal{M}_x| = b_x$. Due to the insertion/deletion of an edge, its endpoints can become DIRTY. To ensure that no node remains DIRTY, we call the following subroutine in Figure 1.

```
01. WHILE the set of DIRTY nodes is nonempty
02.     IF the set of FULL-FROM-ABOVE nodes is nonempty, THEN
03.         Let $x$ be a FULL-FROM-ABOVE node with the largest value of BASE$(x)$.
04.         Call the subroutine FIX-DIRTY$(x)$.   // See Section 2.1.1.
05.     ELSE
06.         Let $x$ be any DIRTY node that is DEFICIENT at level $\ell(x) \geq 0$.
07.         Call the subroutine FIX-DIRTY$(x)$.   // See Section 2.1.2.
```

Figure 1 Fixing the dirty nodes.

We highlight two important aspects of this subroutine. First, during an iteration of the WHILE loop in Figure 1, the node x can move to a new level due to the call to FIX-DIRTY(x). Immediately after the call to FIX-DIRTY(x), the node x is no longer DIRTY. But the call to FIX-DIRTY(x) might lead to some other nodes becoming DIRTY, and these newly DIRTY nodes will be handled in subsequent iterations of the WHILE loop. Second, the WHILE loop in Figure 1 always gives preference to the FULL-FROM-ABOVE nodes over the DIRTY nodes that are DEFICIENT at a nonnegative level. Furthermore, among the FULL-FROM-ABOVE nodes, the WHILE loop picks the one with the largest BASE(.) value. The WHILE loop picks a DIRTY and DEFICIENT node only if there is no FULL-FROM-ABOVE node. This aspect of our algorithm will ensure that Lemma 9 holds, which, in turn, will play a crucial role in the analysis of the amortised update time.

2.1.1 The subroutine FIX-DIRTY(x) on a FULL-FROM-ABOVE node x

Suppose that the subroutine FIX-DIRTY(x) is called on a FULL-FROM-ABOVE node x at level $\ell(x) = i$ with BASE$(x) = j$ (say). Since the node x is FULL-FROM-ABOVE, we have $j > i$. We first move the node x up from level i to level j, and update all the relevant data structures (as described in the beginning of Section 2.1). We next check the number of edges in $\mathcal{E}_x^j$, and, accordingly, we consider two cases.

Case 1. $|\mathcal{E}_x^j| \leq 2b_x \cdot \alpha^{j+1}$. In this case, the node x stays at level j and we terminate the subroutine. At this stage the node x has $|\mathcal{M}_x| = b_x$ and BASE$(x) = j = \ell(x)$. Hence, the node x is no longer DIRTY.

Case 2. $|\mathcal{E}_x^j| > 2b_x \cdot \alpha^{j+1}$. In this case, we find the *minumum* level $k \in [j+1, L]$ such that the number of edges (x, y) with $\ell(y) \leq k$ lies in the range $(2b_x \cdot \alpha^k, 2b_x \cdot \alpha^{k+1}]$. Such a level exists since $b_x \cdot \alpha^{L+1} \geq n >$ degree of x. We now move the node x from level j to level k. In doing so, x becomes the owner of all the edges whose other endpoints are at level $< k$ and we update all the relevant data structures. Next, we *unmatch* all edges $(x, y) \in \mathcal{M}_x$ whose other endpoints are at levels $\ell(y) < k$, and update all the relevant data structures. Let λ_x be the value of $|\mathcal{M}_x|$ at this stage. We now select $(b_x - \lambda_x)$ edges from $\mathcal{O}_x$ uniformly at random, and add them to $\mathcal{M}$ by calling RANDOM-SETTLE(x)(See Section 2.1.3). At this stage we have $|\mathcal{M}_x| = b_x$ and BASE$(x) = k = \ell(x)$, and hence x is no longer DIRTY. We terminate the subroutine at this point. This procedure can lead to other nodes becoming DIRTY, for two reasons. (1) When we unmatch an edge (x, y) with $\ell(y) < k$, the node y might become DEFICIENT and DIRTY. (2) Suppose that while executing RANDOM-SETTLE(x), we add an edge $(x, y) \in \mathcal{O}_x$ to $\mathcal{M}$. Then the node y can become FULL-FROM-ABOVE (and DIRTY). Else, it might happen that $|\mathcal{M}_y|$ becomes equal to $(b_y + 1)$ after we add the edge (x, y) to $\mathcal{M}$. Then we will need to unmatch some edge $(y, z) \in \mathcal{M}_y$, so as to ensure that $\mathcal{M}$ remains a valid b-matching. But this might lead to z becoming DEFICIENT and DIRTY. These newly DIRTY nodes will be handled in subsequent iterations of the WHILE loop in Figure 1.

▶ **Lemma 6.** *Suppose that we call FIX-DIRTY(x) on a* FULL-FROM-ABOVE *node x, and this moves the node x up from level i to level $k > i$. Then it takes $O(b_x \cdot \alpha^{k+1})$ time.*

Proof. (Sketch) Only the edges $(x, y) \in E$ with $\ell(y) \leq k$ get affected as the node x moves to level k. The data structure associated with every other edge remains unchanged. Thus, the total runtime of the subroutine is proportional to the number of edges in $\{(x, y) \in E : \ell(y) \leq k\}$, plus the time taken by the potential call to RANDOM-SETTLE(x). The latter quantity is bounded in Lemma 10. It is easy to check that the former quantity is at most $2b_x\alpha^{k+1}$, for otherwise the node x would have moved up to a level larger than k. ◀

2.1.2 The subroutine FIX-DIRTY(x) on a DEFICIENT node x

Suppose that the subroutine FIX-DIRTY(x) is called on a DEFICIENT node x at level $\ell(x) = j \geq 0$. We first check the number of edges in $\mathcal{E}_x^j$. Accordingly, we consider two cases.

Case 1. $|\mathcal{E}_x^j| > 2b_x \cdot \alpha^{j+1}$. Here, we perform the same steps as in Case 2 in Section 2.1.1.

Case 2. $|\mathcal{E}_x^j| \leq 2b_x \cdot \alpha^{j+1}$. In this case, we try to find a maximal level k in range $[0, j]$ such that the number of edges (x, y) with $\ell(y) < k$ lies in the range $(2b_x \cdot \alpha^k, 2b_x \cdot \alpha^{k+1}]$. If no such level exists, then we set $k = -1$. Next, we move the node x from level j to level k. In doing so, x dis-owns all the edges (and the other endpoints becomes the owner) that are at level $[k, j]$ and we update the relevant data structures. We then unmatch all the edges (x, y) whose other endpoints are at levels $\ell(y) < j$, and update the relevant data structures. Let λ_x be the value of $|\mathcal{M}_x|$ at this stage. If $k \geq 0$, then we call RANDOM-SETTLE(x), which selects $(b_x - \lambda_x)$ edges uniformly at random from $\mathcal{O}_x = \{(x, y) \in E : \ell(y) < k\}$, and adds those edges to $\mathcal{M}$. At this point $|\mathcal{M}_x| = b_x$ and BASE$(x) = k = \ell(x)$, and hence x is no longer DIRTY. So we terminate the subroutine. If $k = -1$, then instead of calling RANDOM-SETTLE(x), we follow the procedure in Section 2.1.4.

▶ **Lemma 7.** *Suppose that we call FIX-DIRTY(x) on a* DEFICIENT *node x at level $j \geq 0$. If the subroutine moves the node x to a level $k > j$, then it takes $O(b_x \cdot \alpha^{k+1})$ time.*

Proof. Exactly similar to the proof of Lemma 6. ◀

▶ **Lemma 8.** *Suppose that we call FIX-DIRTY(x) on a* DEFICIENT *node x at level $j \geq 0$. If the subroutine moves the node x to a level $k \leq j$, then it takes $O(b_x \cdot \alpha^{j+1})$ time.*

Proof. (Sketch) Only the edges $(x, y) \in E$ with $\ell(y) \leq j$ get affected as the node x moves to level k. The data structure associated with every other edge remains unchanged. Thus, the total running time of the subroutine is proportional to the number of edges (x, y) with $\ell(y) \leq j$, plus the time for the call to RANDOM-SETTLE(x) or the procedure in Section 2.1.4. The latter quantity is bounded in Lemma 10 and in Section 2.1.4. As we are in Case 2, the former quantity is by definition at most $2b_x \cdot \alpha^{j+1}$. ◀

2.1.3 The subroutine RANDOM-SETTLE(x)

```
01. WHILE $|\mathcal{M}_x| < b_x$
02.     Pick a uniformly random edge $(x, y) \in \mathcal{O}_x \setminus \mathcal{M}_x$, and add it to the $b$-matching.
        Specifically, set $\mathcal{M} \leftarrow \mathcal{M} \cup \{(x, y)\}$, and update the relevant data structures.
03.     IF $|\mathcal{M}_y| = b_y + 1$, THEN
04.         Select an edge $(y, z) \in \mathcal{M}_y \setminus \{(x, y)\}$ which minimises the value of $\ell(y, z)$,
            and unmatch the edge $(y, z)$. Specifically, set $\mathcal{M} \leftarrow \mathcal{M} \setminus \{(y, z)\}$,
            and update the relevant data structures.
```

Figure 2 RANDOM-SETTLE(x)

Suppose that we call RANDOM-SETTLE(x) on a node x at level $\ell(x) = k \geq 0$. From Sections 2.1.1 and 2.1.2, we infer that $2b_x \cdot \alpha^k < |\mathcal{O}_x| \leq 2b_x \cdot \alpha^{k+1}$ at this point in time. Let $\lambda_x = |\mathcal{M}_x|$ be the number of matched edges incident on x just before the call to the subroutine. The subroutine selects $(b_x - \lambda_x)$ edges uniformly at random from $\mathcal{O}_x$ and adds them to $\mathcal{M}$. Thus, at the end of the subroutine we have $|\mathcal{M}_x| = b_x$. The following lemma crucially uses our preferential treatment to FULL-FROM-ABOVE nodes over DEFICIENT nodes.

▶ **Lemma 9.** *If a call to RANDOM-SETTLE(x) matches an edge (x, y) and unmatches an edge (y, z) (Step (04) in Figure 2), then we have $\ell(z) < \ell(x)$ during that call.*

Proof. Consider two possible cases, depending on the state of the node x.
Case 1. The node x is DEFICIENT just before the call to FIX-DIRTY(x) in Figure 1. Then at this stage there is no FULL-FROM-ABOVE node. This holds since the WHILE loop in Figure 1 always picks a FULL-FROM-ABOVE node over a DEFICIENT node. In particular, the node y is not FULL-FROM-ABOVE. Since (y, z) is a matched edge that minimises the value of $\ell(y, z)$, and since y is not FULL-FROM-ABOVE, we must have $\ell(z) \leq \ell(y, z) = \ell(y)$. Finally, note that $(x, y) \in \mathcal{O}_x$, and hence we get: $\ell(z) \leq \ell(y) < \ell(x)$.
Case 2. The node x is FULL-FROM-ABOVE just before the call to FIX-DIRTY(x) in Figure 1. Then at this stage the node x has the highest BASE(x) value among all the FULL-FROM-ABOVE nodes. Suppose that BASE(x) $= j$ at this point in time. Since the subroutine FIX-DIRTY(x) called the subroutine RANDOM-SETTLE(x), we must have been in Case 2 of Section 2.1.1. This means that the node x moves to a level strictly larger than j. Thus, we have $\ell(x) > j$ during the call to RANDOM-SETTLE(x). We now consider two cases, depending on the state of the node y. (a) If the node y is not FULL-FROM-ABOVE during the call to RANDOM-SETTLE(x), then using an argument exactly similar to the one used in Case 1, we infer that: $\ell(z) \leq \ell(y, z) = \ell(y) < \ell(x)$. This concludes the proof. (b) Else if the node y is FULL-FROM-ABOVE during the call to RANDOM-SETTLE(x), then we claim that BASE(y) $\leq j$ at the same time-instant. This is true since the WHILE loop in Figure 1 picked the node x over y, and so we must have $j =$ BASE(x) $\geq$ BASE(y) just before the call to FIX-DIRTY(x). Since (y, z) is an edge in $\mathcal{M}$ that minimises the value $\ell(y, z)$, we get: $\ell(z) \leq \ell(y, z) =$ BASE(y) $\leq j < \ell(x)$. This concludes the proof. ◀

▶ **Lemma 10.** *A call to the subroutine RANDOM-SETTLE(x) takes $O(b_x \cdot \alpha^{\ell(x)+1})$ time.*

Proof. (Sketch) Let $\lambda_x = |\mathcal{M}_x|$ be the number of matched edges incident on x just before the call to the RANDOM-SETTLE(x). The subroutine picks $(b_x - \lambda_x)$ edges uniformly at random from its set of owned edges $\mathcal{O}_x$. This takes $O(|\mathcal{O}_x|)$ time. Further, steps (03) – (04) in each iteration of the WHILE loop in Figure 2 takes $O(1)$ time. Thus, the total time taken by the subroutine is $O(|\mathcal{O}_x| + b_x) = O(b_x \cdot \alpha^{\ell(x)+1})$. The last equality holds since $|\mathcal{O}_x| \leq 2b_x \cdot \alpha^{\ell(x)+1}$ (see the discussion in the beginning of Section 2.1.3). ◀

2.1.4 Handling the nodes in level -1: Maintaining Invariant 2

Every node at level -1 can be in two states: *dead* or *alive.* A node x becomes alive when:
(1) x moves down to level -1 from a higher level. In this case, it scans all its incident edges at level -1 and tries to add them to $\mathcal{M}$ till $|\mathcal{M}_x|$ becomes equal to b_x. If it cannot find sufficient number of edges at level -1 to be FULL, then it remains alive, otherwise it becomes dead. The total computation cost for this step is $O(b_x)$ since x has at most $O(b_x \cdot \alpha^0)$ neighbours in level -1. This cost is charged to Lemma 8.
(2) x is dead at level -1, and it becomes DEFICIENT due to either an unmatching or a deletion of an incident edge. In case (1), x became dead only when it was FULL. Thus, at least ϵb_x edges incident on x have been either deleted from the graph or removed from $\mathcal{M}$. In the former event, we give one unit of credit to x for each edge deletion incident on it. In the latter event, we give one unit of credit to x for every unmatching of an edge incident on it at level $j \geq 0$ from the FIX-DIRTY$(\cdot)$ procedure that removes this edge from the matching. Thus, node x accumulates at least ϵb_x worth of credit by the time it becomes DEFICIENT. Hence, it can now scan all its incident edges at level -1, as there are $O(b_x)$ such edges.

Finally, if x is not FULL, and an edge (x, y) is added at level -1 where y is also not FULL, then we add the edge (x, y) to the b-matching. This takes $O(1)$ time. A node x at level -1 becomes dead when it is FULL. Once x becomes dead, we wait till it becomes DEFICIENT again. So there is no processing done on x as long as it remains dead. The only non-trivial processing done on x is when it becomes alive, and this takes $O(b_x)$ time. We amortise this cost over the ϵb_x many edge deletions/unmatchings that lead to the change in the state of x.

3 Bounding the Amortised Update Time of Our Algorithm

We fix a sequence of T edge insertions/deletions starting from an empty graph. We show that our algorithm takes $O(T/\epsilon^4)$ time in expectation to handle these T edge insertions/deletions. This implies an amortised update time of $O(1/\epsilon^4)$. Without any loss of generality, we assume that the graph G becomes empty at the end of this sequence of edge insertion/deletions.[1] Further, to highlight the main ideas, we only focus on bounding the time spent on handling the nodes at levels $j \geq 0$. From the discussion in Section 2.1.4, it is easy to infer that the amortised time spent on the nodes at level -1 is at most $O(1/\epsilon)$ per update.
Epochs. The notion of an *epoch* will play a key role in our analysis of the amortised update time. We say that a node $x \in V$ *creates an epoch* when it matches a new edge (x, y) during a given iteration of the WHILE loop in RANDOM-SETTLE(v). Note that this is the only way an edge can be added to the b-matching at levels ≥ 0. At most b_v matched edges can be created in RANDOM-SETTLE(v). For the j-th edge selected by v, define the epoch of j-th edge, γ_j to be the contiguous time interval for which this edge remains in the matching. While creating an epoch γ_j, we select an edge (x, y) uniformly at random from $\mathcal{O}_x \setminus \mathcal{M}$ and add this edge to $\mathcal{M}$ (see Step (02) in Figure 2). This random edge (x, y) is called the *candidate edge* of epoch γ_j. We denote the *level of epoch* γ_j by $\ell(\gamma_j)$. This is same as the level of x during the call to RANDOM-SETTLE(x) in Figure 2. We say that an epoch

[1] Otherwise, we can ourselves delete the existing edges from G one after the other, and get a new sequence of at most $2T$ edge insertions/deletions. We then bound the total update time on this new sequence.

is *destroyed* when its candidate edge is removed from the b-matching $\mathcal{M}$. This happens either when the candidate edge gets deleted from the graph, or when the candidate edge gets unmatched during the course of our algorithm. As the graph G is empty at the end of T edge insertions/deletions, every epoch gets destroyed at some point in time.

Overview of the analysis. For every node $v \in V$ and level $i \in [0, L]$, let $X_{v,i}$ be a random variable that denotes the total number of epochs created by v at level i, as we handle the fixed sequence of T edge insertions/deletions. In Lemma 11, we express the total update time as a function of these random variables $\{X_{v,i}\}$. In Lemma 12, we upper bound the expected value of this function in terms of T. The bound on the expected amortised update time of our algorithm follows from Lemmas 11 and 12 (see Corollary 13).

▶ **Lemma 11.** *It takes $\sum_{v,i} X_{v,i} \cdot O(\alpha^{i+1}/\epsilon)$ time to handle T edge insertions/deletions.*

Proof. (Sketch) The time taken is dominated by the WHILE loop in Figure 1. Hence, it suffices to bound the time spent on the calls to FIX-DIRTY(v) over all $v \in V$. We will charge the total time spent in FIX-DIRTY to the start or an end of an epoch in the following way:

Scenario 1. We have $\ell(v) = i$, v is DEFICIENT, and FIX-DIRTY(v) moves the node v up to some level $j > i$ and calls RANDOM-SETTLE. By Lemma 7 the runtime of FIX-DIRTY(v) is $O(b_v\alpha^{j+1})$. When v comes to level j, it calls RANDOM-SETTLE(v) and starts at least ϵb_v new epochs at level j. So, the computation cost charged per new epoch created is $O(\alpha^{j+1}/\epsilon)$.

Scenario 2. We have $\ell(v) = i$, v is DEFICIENT, and FIX-DIRTY(v) moves v down to some level $j \leq i$. By Lemma 8 the runtime of FIX-DIRTY(v) is $O(b_v\alpha^{i+1})$. Since, v became deficient at level i, at least ϵb_v epochs involving v have been destroyed (since the last time v came to level i, FIX-DIRTY makes it FULL). Note that all such epochs have level $\geq i$. So, the computation cost charged to all epochs destroyed is $O(\alpha^{i+1}/\epsilon)$.

Scenario 3. The hard case is as follows: $\ell(v) = i$, v is FULL-FROM-ABOVE, and FIX-DIRTY(v) moves the node v up to some level $j =$ BASE(v) (or at $j >$BASE(v), Case 2, Section 2.1.1). By Lemma 6 the runtime of FIX-DIRTY(v) is $O(b_v\alpha^{j+1})$. However, it is not necessary that v starts ϵb_v epochs when it arrives at level i. So, we cannot charge this computation cost new epoch of v as no new epochs are created at level j. To overcome this problem, we crucially use the fact that v will eventually become deficient at some higher level. This holds since there are only $\log n$ levels in our partition and v cannot always be FULL-FROM-ABOVE. Formally, define a phase ϕ of v to be the maximal time interval in which FIX-DIRTY is not called on v. A phase of v is called DEFICIENT if v becomes deficient during the phase, otherwise it is called FULL-FROM-ABOVE. Define a directed graph $\mathcal{H}_v = (\mathcal{U}_v, \mathcal{E}_v)$ where $\mathcal{U}_v$ is the set of all phases of v. For two phases $\phi_1, \phi_2 \in \mathcal{U}_v$, we have $(\phi_1, \phi_2) \in \mathcal{E}_v$ iff ϕ_1 is a FULL-FROM-ABOVE phase and ϕ_2 is the phase that begins just after ϕ_1 ends. If $l(\phi)$ denote the level of v when the phase ϕ starts, then $\ell(\phi_1) < \ell(\phi_2)$. The graph $\mathcal{H}_v$ is a collection of paths and each path ends with a DEFICIENT phase. Let $\Phi = (\phi_1, \phi_2, \ldots, \phi_k)$ be a *maximal* path in $\mathcal{H}_v$. So ϕ_k is a DEFICIENT phase and ϕ_i is a FULL-FROM-ABOVE phase for all $i < k$, and hence $\ell(\phi_1) < \cdots < \ell(\phi_k)$. Let Y_Φ be the total computation cost (due to FIX-DIRTY(v)) in phases in Φ. We get: $Y_\Phi \leq \sum_{i=1}^{k} 2b_v \cdot \alpha^{\ell(\phi_i)+1} = O(b_v \cdot \alpha^{\ell(\phi_k)+1})$. So all the computation cost can be charged to the epochs destroyed during the deficient phase of v, that is $\phi(k)$. When v move to ϕ_k, it is full (or it calls RANDOM-SETTLE(v) to become full. So at least ϵb_v epochs involving v must be destroyed for v to become deficient (All such epochs have level $\geq i$). So we can charge the computation cost Y_ϕ to these ϵb_v epochs and the cost associated with each epoch is $O(\alpha^{i+1}/\epsilon)$.

From Scenario 1,2 and 3, the computation cost charged to each epoch at level i is $O(\alpha^{i+1}/\epsilon)$. So the total time taken by our algorithm is $\sum_{v,i} X_{v,i} O(\alpha^{i+1}/\epsilon)$ ◀

▶ **Lemma 12.** *We have* $\sum_{v,i} E[X_{v,i}].O(\alpha^{i+1}/\epsilon) = O(\alpha^4 T)$.

▶ **Corollary 13.** *The expected amortised update time of our algorithm is* $O(1/\epsilon^4)$.

For the rest of Section 3, we focus on justifying Lemma 12. In Section 3.1, we classify the epochs into three types. Due to space constraints, in Section 3.2 we present a (handwavy) intuition behind the proof of Lemma 12. For a complete formal proof, see Appendix A.

3.1 Natural, induced and forced epochs

We say that an epoch is *natural* iff it gets destroyed when its candidate edge is deleted from the graph. If an epoch is not natural, then it gets destroyed when its candidate edge is removed from $\mathcal{M}$ (but does not get deleted from G). This can happen under four possible situations: (1) In Case 2 of Section 2.1.1, when a node x moves up from level j to level $k > j$, we unmatch all its incident edges $(x, y) \in E$ whose other endpoints are at levels $\ell(y) < k$. (2) In Case 1 of Section 2.1.2, when a node x moves up from a level j to a level $k > j$, we unmatch all its incident edges $(x, y) \in E$ whose other endpoints are at levels $\ell(y) < k$. (3) During a call to the subroutine RANDOM-SETTLE(x), we unmatch an edge (y, z) because y gets matched to x. See Step (04) in Figure 1. (4) In Case 2 of Section 2.1.2, when a node x moves down from a level j, we unmatch all its incident edges (x, y) whose other endpoints are at levels $\ell(y) < j$. The epochs whose candidate edges get unmatched under situations (1), (2) and (3) are called *induced* epochs. In contrast, the epochs whose candidate edges get unmatched under situation (4) are called *forced* epochs.

3.2 Intuition behind the proof of Lemma 12: A token based argument

Let $\mathcal{V}$ be the set of all epochs. Suppose that we assign $\mathcal{T}(\gamma) = \alpha^{\ell(\gamma)+1}$ tokens to every epoch $\gamma \in \mathcal{V}$. Since the computation cost charged to γ (by Lemma 11) is $O(\alpha^{l(\gamma)+1}/\epsilon)$, each token is charged with $O(1/\epsilon)$ amount of computation cost. In Lemma 14, we show that the sum $\sum_{\gamma \in \mathcal{V}} \mathcal{T}(\gamma)$ is at most $O(1/\epsilon)$ times the total number of tokens assigned to the natural epochs, which is given by the sum $\sum_{\gamma \in \mathcal{N}} \mathcal{T}(\gamma)$. In the paragraph below, we present a (handwavy) intuition which implies that $E\left[\sum_{\gamma \in \mathcal{N}} \mathcal{T}(\gamma)\right] = O(\alpha \cdot T)$. These observations, taken together, justify Lemma 12. (This intuition based argument implies that the total expected running time of the algorithm is $O(\alpha \cdot T/\epsilon^2)$. In the Appendix A, we give a technically precise argument in which there is an extra α multiplicative factor in the running time).

Suppose that each time an edge (x, y) gets deleted from G, we generate 2 *dollars* and give 1 dollar to each of the endpoints $\{u, v\}$. For the sequence of T edge insertions/deletions in G, the total number of dollars generated is $O(T)$. We will try to convince the reader that a natural epoch γ at level $\ell(\gamma) = i$ receives $\Theta(\alpha^{\ell(\gamma)})$ dollars under this scheme, in expectation. This will give us: $E\left[\sum_{\gamma \in \mathcal{N}} \mathcal{T}(\gamma)\right] = O(\alpha \cdot T)$. Accordingly, suppose that a node x at level $\ell(x) = i$ creates an epoch γ as per Step (02) in Figure 2. From the discussion in the beginning of Section 2.1.3, we get $|\mathcal{O}_x \setminus \mathcal{M}| > b_x \cdot \alpha^i$ at this point in time. We say that each edge $e \in \mathcal{O}_x \setminus \mathcal{M}$ is a *witness* for the epoch γ. Since the sequence of edge insertions/deletions in G is oblivious to the random bits used by the algorithm, in expectation half of these *witness-edges* will get deleted before the candidate edge (x, y) of the epoch γ. Thus, intuitively, if the epoch is natural, then in expectation at least $(b_x/2) \cdot \alpha^i$ of its witness edges get deleted during the lifespan of the epoch. For each of these deletions of the witness edges, the node x receives 1 dollar. Note that each such edge can be a witness to at most b_x epochs of x (since $|\mathcal{M}_x| \le b_x$). Consider the time-instant when an edge (x, y)

gets deleted from G and the node x receives 1 dollar. If we distribute this 1 dollar received by x evenly among all the epochs of x which have (x, y) as a witness, then each of those epochs will receive at least $1/b_x$ dollars. From the preceding discussion, recall that at least $(b_x/2) \cdot \alpha^i$ many witness edges get deleted from G during the lifespan of a natural epoch γ of x at level i. Hence, as promised, we conclude that such an epoch will receive at least $(1/b_x) \cdot (b_x/2) \cdot \alpha^i = \Theta(\alpha^i)$ dollars during its lifespan, in expectation.

▶ **Lemma 14.** *Let $\mathcal{V}$ be the set of all epochs, and let $\mathcal{N} \subseteq \mathcal{V}$ be the set of all natural epochs. Assign $\mathcal{T}(\gamma) = \alpha^{\ell(\gamma)+1}$ tokens to every epoch $\gamma \in \mathcal{V}$. Then $\sum_{\gamma \in \mathcal{V}} \mathcal{T}(\gamma) = O(1/\epsilon) \cdot \sum_{\gamma \in \mathcal{N}} \mathcal{T}(\gamma)$.*

We devote the rest of this section to the proof of Lemma 14. Towards this end, we construct a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. To distinguish this from the input graph $G = (V, E)$, we refer to each vertex in V as a *meta-node* and each edge in $\mathcal{E}$ as a *meta-edge*. The set of meta-nodes $\mathcal{V}$ corresponds to the set of all epochs. The set of meta-edges $\mathcal{E}$ is defined as follows. Each natural epoch has no incoming meta-edge. Each induced or forced epoch has exactly one incoming meta-edge. Thus, without any loss of generality, for every meta-edge $(x, y) \in \mathcal{E}$ directed from x towards y, we say that y is the parent of x and x is a child of y. Note that each meta-node has at most one parent, but it can have multiple children. We will now describe the rules that specify the parent of each induced or forced epoch.

The parent of an induced epoch is specified as follows. Note that an induced epoch is destroyed under three possible scenarios: These are situations (1), (2) and (3) as described in Section 3.1. In situation (3), we define the parent of the induced epoch (with (y, z) as the candidate edge) to be the epoch of x which lead to its destruction (i.e., the epoch with (x, y) as the candidate edge). Both in situation (1) and situation (2), a node x moves up to a higher level, unmatches some incident edges (thereby destroying some induced epochs), and adds some new incident edges to $\mathcal{M}$ (thereby creating some new epochs). As per the descriptions in Sections 2.1.1 and 2.1.2, the node x becomes FULL after these operations. Hence, the number of new epochs created during these steps is at least the number of induced epochs that get destroyed. Accordingly, for each induced epoch γ that gets destroyed, we find a unique epoch γ' that gets created, and make γ' the parent of γ. This way of defining parents for the induced epochs ensures the following property.

▶ Claim 1. If an epoch γ' is the parent of an induced epoch γ, then $\ell(\gamma') > \ell(\gamma)$. Further, an epoch γ' can have at most two children that are induced epochs.

Proof. Let γ' be the parent of an induced epoch γ. Let x and z respectively be the nodes that created the epochs γ' and γ. If γ is destroyed during situation (3) in Section 3.1, then Lemma 9 implies that $\ell(\gamma') = \ell(x) > \ell(y, z) = \ell(\gamma)$ when γ is destroyed. Else if γ is destroyed during situation (1) or (2) in Section 3.1, then the descriptions in Sections 2.1.1, 2.1.2 lead us to the following conclusion: The node x moves up from a level l to some higher level $l' > l$. The epoch γ' is created at level l'. Further, the candidate edge of the epoch γ had both endpoints at a level strictly less than l' when γ was created. Thus, we again get $\ell(\gamma') > \ell(\gamma)$. Finally, an epoch γ' can have at most one child that is an induced epoch which gets destroyed during situation (1) or (2) in Section 3.1, and at most one child that is an induced epoch which gets destroyed during situation (3) in Section 3.1. The claim follows. ◀

We now define the parents of forced epochs. The forced epochs are destroyed during situation (4) in Section 3.1. Thus, we consider the following scenario. A node x is at level $\ell(x) = j \geq 0$, and it becomes DEFICIENT at time t_1 (say). At this stage we have $|\mathcal{M}_x| < (1 - \epsilon) \cdot b_x$. Hence, as the node x moves down, at most $(1 - \epsilon) \cdot b_x$ many forced epochs get destroyed. Let F denote the set of these forced epochs. We have $|F| \leq (1 - \epsilon) \cdot b_x$.

Also note that the node x must have been FULL the last time (say t_0) it *settled* at level j after a call to FIX-DIRTY(x). Thus, for situation (4) to occur, the node x must have lost at least ϵb_x many edges in $\mathcal{M}_x$ during the time-interval $[t_0, t_1]$. Let H denote the set of these epochs that were alive at time t_0, were destroyed before time t_1, and whose candidate edges were part of $\mathcal{M}_x$. We have $|H| \geq \epsilon \cdot b_x$.

We claim that $\ell(h) > j$ for every forced epoch $h \in H$. To see why this is true, let (x, y) be the candidate edge for h. Clearly, when h was destroyed the node x stayed put at level j. It follows that since h is a forced epoch, the node y must have created h. The claim is equivalent to the statement that $\ell(y) > j$ when h gets created, and this in turn is equivalent to the statement that $\ell(y) > j$ when h gets destroyed (since $\ell(y)$ does not change during the lifespan of epoch h). To see why the latter statement is true, note that by definition y moves down to a lower level when the epoch h gets destroyed. Thus, as per Case 2 of Section 2.1.2, the node y can unmatch the edge (x, y) only if $\ell(y) > j$ at that time-instant. Hence, we get:

▶ Property 15. Every forced epoch $h \in H$ has level $\ell(h) > j$.

We now *assign* the epochs in F *evenly* among the epochs in H. To be more specific, after this step, each epoch $h \in H$ gets $|F|/|H| \leq (1-\epsilon)/\epsilon$ many epochs $f \in F$ *assigned to it.* We are now ready to define the parents of the forced epochs F. Suppose that an epoch $f \in F$ is assigned to an epoch $h \in H$. If $\ell(h) > j$, then h becomes the parent of f. Else if $\ell(h) = j$, then from Property 15 it follows that either h is a natural epoch or h is an induced epoch. In the former event, again h becomes the parent of f. In the latter event, Claim 1 guarantees that h has a parent, say $p(h)$, and this epoch $p(h)$ becomes the parent of f. Claim 1 also implies that $\ell(p(h)) > \ell(h) = j$. Since $\ell(f) = j$, we immediately get the following claim.

▶ Claim 2. Suppose that an epoch γ' is the parent of a forced epoch γ. Then either (1) $\{\ell(\gamma') > \ell(\gamma)\}$ or (2) $\{\gamma'$ is a natural epoch and $\ell(\gamma') \geq \ell(\gamma)\}$.

Claim 3 holds since: (1) An epoch $h \in H$ gets at most $(1-\epsilon)/\epsilon$ many epochs $f \in F$ assigned to it. (2) An epoch γ can have two children $h, h' \in H$ both of which are induced epochs (see Claim 1), and both h, h' can get at most $(1-\epsilon)/\epsilon$ many epochs $f \in F$ assigned to them.

▶ Claim 3. An epoch can have at most $3(1-\epsilon)/\epsilon$ many forced epochs as its children.

Proof of Lemma 14. In the meta-graph $\mathcal{G}$, every induced or forced epoch has exactly one incoming edge, and every natural epoch has zero incoming edge. Hence, the meta-graph $\mathcal{G}$ is a collection of rooted directed trees, where the tree-edges are directed away from the roots. The root of each tree is a natural epoch, and any internal node is either an induced epoch or a forced epoch. Let $\mathcal{V}'$ be the set of meta-nodes in any such tree with $r \in \mathcal{V}'$ as its root. To prove the lemma, it suffices to show that $\sum_{\gamma \in \mathcal{V}'} \mathcal{T}(\gamma) = O(1/\epsilon) \cdot \mathcal{T}(r)$.

Only the root r is a natural epoch in $\mathcal{V}'$. Hence, Claims 1, 2, 3 imply that: (1) $\ell(\gamma') > \ell(\gamma)$ whenever an *internal* meta-node $\gamma' \in \mathcal{V}'$ is the parent of $\gamma \in \mathcal{V}'$. (2) Any internal meta-node in $\mathcal{V}'$ has at most $2 + 3(1-\epsilon)/\epsilon \leq 3/\epsilon$ children. (3) The root also has at most $2 + 3(1-\epsilon)/\epsilon \leq 3/\epsilon$ children. Let C be the set of children of the root. Then for all $\gamma \in C$, we have $\ell(r) \geq \ell(\gamma)$.

Consider a meta-node $\gamma \in C$. Let $\mathcal{T}^*(\gamma)$ denote the total number of tokens assigned to the meta-nodes in the subtree rooted at γ. From the above discussions, we get: $\mathcal{T}^*(\gamma) \leq \sum_{j=0}^{\ell(\gamma)} \left(\frac{3}{\epsilon}\right)^{\ell(\gamma)-j} \cdot \alpha^{j+1} = O(\alpha^{\ell(\gamma)+1}) = O(\alpha^{\ell(r)+1})$. The first inequality holds since there are at most $(3/\epsilon)^{\ell(\gamma)-j}$ descendents of γ at level $j \leq \ell(\gamma)$, and each of these descendents get α^{j+1} tokens. The second equality holds since $\alpha = 5/\epsilon >> (3/\epsilon)$. The third equality holds since $\ell(\gamma) \leq \ell(r)$. Hence, the total number of tokens assigned to the tree is given by: $\sum_{\gamma \in \mathcal{V}'} \mathcal{T}(\gamma) = \mathcal{T}(r) + \sum_{\gamma \in C} \mathcal{T}^*(\gamma) \leq \alpha^{\ell(r)+1} + (3/\epsilon) \cdot O(\alpha^{\ell(r)+1}) = O(1/\epsilon) \cdot \mathcal{T}(r)$. ◀

References

1 S. Baswana, M. Gupta, and S. Sen. Fully dynamic maximal matching in $O(\log n)$ update time. In *FOCS 2011*.
2 Aaron Bernstein and Cliff Stein. Faster fully dynamic matchings with small approximation ratios. In *SODA 2016*.
3 Aaron Bernstein and Cliff Stein. Fully dynamic matching in bipartite graphs. In *ICALP 2015*.
4 Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. Design of dynamic algorithms via primal-dual method. In *ICALP 2015*.
5 Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. Deterministic fully dynamic data structures for vertex cover and matching. In *SODA 2015*.
6 Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. New deterministic approximation algorithms for fully dynamic matching. In *STOC 2016*.
7 Anupam Gupta, Ravishankar Krishnaswamy, Amit Kumar, and Debmalya Panigrahi. Online and dynamic algorithms for set cover. In *STOC 2017*.
8 Manoj Gupta and Richard Peng. Fully dynamic $(1+\epsilon)$-approximate matchings. In *FOCS 2013*.
9 Ofer Neiman and Shay Solomon. Simple deterministic algorithms for fully dynamic maximal matching. In *STOC 2013*.
10 Krzysztof Onak and Ronitt Rubinfeld. Maintaining a large matching and a small vertex cover. In *STOC 2010*.
11 David Peleg and Shay Solomon. Dynamic $(1+\epsilon)$-approximate matchings: A density-sensitive approach. In *SODA 2016*.
12 Shay Solomon. Fully dynamic maximal matching in constant update time. In *FOCS 2016*.

A Proof of Lemma 12

Lemma 12 states that $\sum_{v,i} E\left[X_{v,i} \cdot (\alpha^{i+1}/\epsilon)\right] = O((\alpha^4) \cdot T)$. In Section 3.2 we saw the intuition behind the proof of the lemma. We charged each epoch at level i with α^{i+1} many tokens and proved that the total number of tokens generated is within a constant factor of the total number of tokens charged to the natural epochs (see Lemma 14). We had given an informal argument as to why the total number of tokens charged to the natural epochs is $O(\alpha \cdot T)$ in expectation.

When a node x creates an epoch γ at level i, the candidate edge of γ, say e, is chosen uniformaly at random from a set of witness edges of size at least $b_x\alpha^i$. Hence against an oblivious adversary at least $(b_x/2) \cdot \alpha^i$ many witness edges are deleted before e is deleted from the graph. Consider the time-instant when an edge (x, y) gets deleted from G and the node x receives 1 dollar. If we distribute this 1 dollar received by x evenly among all the epochs of x which have (x, y) as a witness, then each of those epochs will receive at least $1/b_x$ dollars. We want to claim that the expected number of dollars received by an epoch given that it is a natural epoch is $O(\alpha^i)$. But this argument involves what we call a backward conditioning, that is, we condition on a future time instant. The probability of an epoch being natural depends on the candidate edge $e = (x, y)$ chosen. This is because, at a later time instance, the edge e might be unmatched by the algorithm because some other node changed levels or a picks an edge with a Full endpoint (x or y), thus unmatching the edge e.

We give an alternate token based argument that doesn't use backward conditioning. For every node v and level i we maintain a bank account $B_{v,i}$. Initially $B_{v,i} = 0$. For each edge deletion adjacent to v we generate α^3/b_v dollars that will be added to some bank account. We also transfer money between accounts when an edge is unmatched. Note that we allow negative balance in the bank account. That is, let $B_{v,i} = d$ and we remove $d' > d$ dollars from $B_{v,i}$. Then $B_{v,i} = d - d' < 0$. We relate the number of epochs to the number edge deletions by looking at the change in a bank account due to an epoch.

Transaction Rules

1. Consider an epoch γ of a node v at level $i \geq 0$ with witness set W. When an witness edge $(u, v) \in W$ is deleted deposit α^3/b_v dollar in $B_{v,i}$. Note that at any instance an edge (u, v) can be in the witness sets of at most b_v many epochs of v and b_u many epochs of u. So a deletion contributes at most $2\alpha^3$ dollars.
2. Consider an epoch γ of v at level i and let γ be either an induced epoch or forced epoch. Let δ be the parent of γ in $\mathcal{G}$ (see Section 3.2). Suppose δ is an epoch of a node u and $\ell(\delta) = j$. From Claims 1 and 2, we have $j \geq i$. When γ is terminated we transfer α^{i+1}/c tokens from $B_{u,j}$ to $B_{v,i}$. Note that if δ is an induced or a forced epoch then $j > i$. We will fix c later, it is an absolute constant > 6.

▶ **Corollary 16.** $\sum_{v,i} B_{v,i} \leq 2\alpha^3 T$

Proof. Follows from transcation rules 1 and 2. ◀

In the following Lemma 17, we relate the expected number of epochs of a node v at level i and to the expected number of dollars in $B_{v,i}$. The proof appears in Section A.1.

▶ **Lemma 17.** *For any node v and $i \geq 0$, we have $(\alpha^i/2c\epsilon) \cdot E[X_{v,i}] \leq E[B_v, i]$.*

Using Lemma 11, the expected running time of the algorithm is $\sum_{v,i} E[X_{v,i}] O(\alpha^{i+1}/\epsilon)$. From corollary 16 and Lemma 17 we get $\sum_{v,i} E[X_{v,i}] \cdot O(\alpha^{i+1}/\epsilon) \leq \sum_{v,i} 2\alpha c E[B_{v,i}] = 4c\alpha^4 T$. Thus we get Lemma 12. We will prove Lemma 17 in Section A.1.

A.1 Proof of Lemma 17

At any given time instant t we condition on the current state of the graph. Now suppose at time t an epoch γ of v is created at level $i \geq 0$ with witness set W. Conditioned on this event, let $\Delta B_{v,i}$ be the amount by which $B_{v,i}$ changes due to the epoch γ and let $\Delta X_{v,i}$ be the change in $X_{v,i}$ due to γ. Trivially $\Delta X_{v,i} = 1$. The bank account $B_{v,i}$ gets affected by γ in the following ways.
1. For every edge $(u, v) \in W$ that gets deleted during the epoch γ, we deposit α^3/b_v dollars into $B_{v,i}$.
2. If γ is a forced/induced epoch, then when γ terminates $B_{v,i}$ gets α^{i+1}/c dollars from its parent.
3. Let δ be a child of γ in $\mathcal{G}$. If γ is a natural epoch $B_{v,i}$ loses at most α^{i+1}/c dollars when δ is terminated. If γ is a forced/induced epoch $B_{v,i}$ loses at most α^i/c dollars when δ is terminated.

Since any epoch has at most $2 + 3(1-\epsilon)/\epsilon \leq 3/\epsilon$ many children (see Claims 1 and 3) we have,

$$\Delta B_{v,i} \geq \frac{\alpha^3}{b_v} \left(\text{NUMBER OF EDGE DELETIONS IN } W \text{ DURING } \gamma\right) - \frac{3\alpha^{i+1}}{c\epsilon}, \text{ if } \gamma \text{ is a natural epoch,} \tag{1}$$

$$\geq \frac{\alpha^{i+1}}{c} - \frac{3\alpha^i}{c \cdot \epsilon}, \text{ if } \gamma \text{ is an induced/forced epoch.} \tag{2}$$

▶ **Lemma 18.** *$E[\Delta B_{v,i}] \geq \alpha^i/2c\epsilon$, (for $\alpha \geq 5/\epsilon$, $c > 6$).*

Lemma 18 implies Lemma 17. For each epoch of v in level i, we have $\Delta X_{v,i}$ =1 and $E[\Delta B_{v,i}] \geq \alpha^i/2c\epsilon$. So $E[B_{v,i}]$ is at least $(\alpha^i/2c\epsilon)E[X_{v,i}]$. We devote the rest of the Section to the proof of Lemma 18.

Proof of Lemma 18

Recall that we conditioned on the state of the graph at time t and the epoch γ was started by choosing an edge uniformly at random. So whether or not γ is a natural epoch depends on the candidate edge $e(\gamma) = (v, w)$, and the future execution of the algorithm. For all $e \in W$ let p_e be the probability of γ being a natural epoch given that e is matched. Since we are dealing with an oblivious adversary the sequences of edge updates is fixed. So we can order the edges in W in order of their deletion, say $e_1, e_2, \ldots, e_{|W|}$. Let $\mathcal{C}^k$ be the event that the k^{th} edge of W was chosen as the candidate edge. If γ is a natural epoch then it gets $k\alpha^3/b_v$ dollars, since k edges are deleted before the epoch ends (see transaction rule 1). Else if γ is an induced/forced epoch it gets at least α^{i+1}/c dollars from its parent (see transaction rule 2). From Claims 1 and 3, we have that γ has at most $3/\epsilon$ many children. So if γ is a natural epoch, then it loses at most $\frac{3\alpha^{i+1}}{c\cdot\epsilon}$ dollars (since the level of any child of $\gamma \leq \ell(\gamma)$). Else γ loses at most $\frac{3\alpha^i}{c\epsilon}$ dollars (since the level of any child of $\gamma < \ell(\gamma)$). Thus from Equations 1 and 2 we have $E[\Delta B_{v,i}|\mathcal{C}^k] \geq ((\frac{k\alpha^3}{b_v} - \frac{3\alpha^{i+1}}{c\epsilon}) \cdot p_{e_k} + (\frac{\alpha^{i+1}}{c} - \frac{3\alpha^i}{c\epsilon})(1 - p_{e_k}))$.

Therefore we have,

$$\begin{aligned} E[\Delta B_{v,i}] &= \sum_{k=1}^{|W|} \Pr([\mathcal{C}^k] E[\Delta B_{v,i}|\mathcal{C}^k]) \\ &\geq \sum_{k=1}^{|W|} \frac{1}{|W|} \left(\left(\frac{k\alpha^3}{b_v} - \frac{3\alpha^{i+1}}{c\epsilon} \right) \cdot p_{e_k} + \left(\frac{\alpha^{i+1}}{c} - \frac{3\alpha^i}{c\epsilon} \right) (1 - p_{e_k}) \right) \end{aligned}$$

Let $S_1 = \sum_{k=1}^{|W|} \frac{1}{|W|} \left((\frac{k\alpha^3}{b_v} - \frac{3\alpha^{i+1}}{c\epsilon}) \cdot p_{e_k} \right)$ and
$S_2 = \sum_{k=1}^{|W|} \frac{1}{|W|} \left((\frac{\alpha^{i+1}}{c} - \frac{3\alpha^i}{c\epsilon})(1 - p_{e_k}) \right)$.

So we have $E[\Delta B_{v,i}] \geq S_1 + S_2$.

Note: $\sum_{k=1}^{|W|} p_{e_k} \leq |W| < 2b_v\alpha^{i+1}$, as $p_e \leq 1$ $\forall e \in W$ (see Section 3.2)

Let us consider the following two case, $\sum_{k=1}^{|W|} p_{e_k} < |W|/2$ and $\sum_{k=1}^{|W|} p_{e_k} \geq |W|/2$.
Case 1: $\sum_{k=1}^{|W|} p_{e_k} < |W|/2$. In this case we have,

$$\sum_{k=1}^{|W|} (1 - p_{e_k}) > |W| - |W|/2 = |W/2$$

$$\begin{aligned} \text{Hence, we get} \quad S_2 &= \frac{1}{|W|} \left(\frac{\alpha^{i+1}}{c} - \frac{3\alpha^i}{c\epsilon} \right) \sum_{k=1}^{|W|} (1 - p_{e^k}) \\ &> \frac{1}{|W|} \left(\frac{\alpha^{i+1}}{c} - \frac{3\alpha^i}{c\epsilon} \right) \cdot \frac{|W|}{2} \\ &> \frac{\alpha^{i+1}}{2c} - \frac{3\alpha^i}{2c\epsilon} \end{aligned} \quad \text{(i)}$$

But S_1 may not be non-negative, so bounding just S_2 is not enough.
Note that $(\frac{k\alpha^3}{b_v} - \frac{3\alpha^{i+1}}{c\epsilon}) \geq 0$ for all $k \geq \frac{b_v\alpha^{i-1}}{c\epsilon} \geq \frac{3b_v\alpha^{i-2}}{c\epsilon}$. Also, we have $p_{e_k} \leq 1$ for all $k \geq 1$, since they are probabilities. Thus $S_1 \geq -\sum_{k=1}^{\frac{b_v\alpha^{i-1}}{c}} \frac{1}{|W|}(\frac{3\alpha^{i+1}}{c\epsilon})$
So it is enough to bound $S_2 - \sum_{k=1}^{\frac{b_v\alpha^{i-1}}{c}} \frac{1}{|W|}\frac{3\alpha^{i+1}}{c\epsilon}$ (ii)

$$\begin{aligned}
S_2 - \sum_{k=1}^{\frac{b_v\alpha^{i-1}}{c}} \frac{3\alpha^{i+1}}{c\epsilon} &> \left(\frac{\alpha^{i+1}}{2c} - \frac{3\alpha^i}{2c\epsilon}\right) - \sum_{k=1}^{\frac{b_v\alpha^{i-1}}{c}} \frac{1}{|W|}\frac{3\alpha^{i+1}}{c\epsilon} \\
&> \left(\frac{\alpha^{i+1}}{2c} - \frac{3\alpha^i}{2c\epsilon}\right) - \left(\frac{b_v\alpha^{i-1}}{c}\right)\left(\frac{1}{b_v\alpha^i}\right)\left(\frac{3\alpha^{i+1}}{c\epsilon}\right) \\
&\qquad (|W| \geq b_v\alpha^i, \text{ see Section 3.2}) \\
&> \frac{\alpha^{i+1}}{2c} - \frac{3\alpha^i}{2c\epsilon} - \frac{3\alpha^i}{c^2\epsilon} \\
&> \frac{\alpha^i}{2c\epsilon} \qquad (\text{for } \alpha \geq 5/\epsilon, c > 6) \qquad \text{(iii)}
\end{aligned}$$

From (i), (ii) and (iii), we have $S_1 + S_2 \geq \frac{\alpha^i}{2c\epsilon}$.

Case 2: $\sum_{k=1}^{|W_j|} p_{e_k} \geq |W|/2$.
In this case, the minimum value for S_1 is when $p_{e_k} = 1$ for all $k \leq |W|/2$.

$$\begin{aligned}
S_1 &\geq \sum_{k=1}^{|W|/2} \left(\frac{k\alpha^3}{b_v} - \frac{3\alpha^{i+1}}{c\epsilon}\right)\frac{1}{|W|} \\
&= \left(\frac{|W|(|W|+2)}{8b_v}\right)\cdot\left(\frac{\alpha^3}{|W|}\right) - \frac{3\alpha^{i+1}}{2c\epsilon} \\
&\geq \frac{|W|\alpha^3}{8b_v} - \frac{3\alpha^{i+1}}{2c\epsilon} \\
&\geq \frac{\alpha^{i+3}}{8} - \frac{3\alpha^{i+1}}{2c\epsilon} \qquad (|W| \geq b_v\alpha^i), \text{ see Section 3.2} \\
&\geq \frac{\alpha^{i+1}}{8\epsilon} \qquad (\text{for } c > 6, \alpha \geq 5/\epsilon > \sqrt{5/\epsilon}) \\
&\geq \frac{\alpha^i}{2c\epsilon} \qquad (\text{for } \alpha > 1 > 4/c)
\end{aligned}$$

Since S_2 is non-negative, we have $S_1 + S_2 \geq \frac{\alpha^i}{2c\epsilon}$. ◀.

In the next sections, we describe the low level details of the algorithm that were left out in the main paper due to lack of space. This includes the procedures that (1) Extract Dirty nodes from the Dirty list without adding any asymptotic cost. (2) Finds the Base of a vertex v at without adding any asymptotic cost. (3) Finds the new level of a vertex and the data-structure operations associated with it.

B Extracting Dirty nodes in order

While the Dirty list is non-empty, a Dirty node v is picked from the list and is fixed (see Section 2.1, Figure 1). The Dirty nodes are picked in a specific order as follows.

1. If there are Full-from-above nodes then they are preferred over deficient nodes. Among all the Full-from-above nodes in the Dirty list, the node with the maximum Base is picked.

2. A deficient node v is picked iff there are no FULL-FROM-ABOVE nodes in the dirty list currently.

Picking the DIRTY nodes in order is essential in the proof of Lemma 9, which in turn ensures that Claim 1 holds.

Cost of extracting DIRTY nodes in order. We claim that the total cost to find the appropriate dirty nodes in the specific order is subsumed by the cost of calls to FIX-DIRTY. We maintain a data structure to store and extract the DIRTY nodes efficiently. For each $i \in [0, L]$, we maintain doubly linked list $\mathcal{D}[i]$ which stores all the FULL-FROM-ABOVE nodes v with $\text{BASE}(v) = i$. Once an edge update has been taken care by the algorithm the invariants are restored. Thus $\mathcal{D}[i] = \emptyset$ for all i after the algorithm has handled an edge update.

We maintain p which points to the highest level such that there is FULL-FROM-ABOVE node v with $\text{BASE}(v) = p$. If the current p if pointing to an empty list, i.e. $\mathcal{D}[p] = \emptyset$, we find the highest level $i < p$ below with a non empty $\mathcal{D}[i]$ list. This will take at most p time and it is subsumed by the cost of FIX-DIRTY(v) where v is the previous node that was extracted from $\mathcal{D}[p]$. Let k be the level of v after the call to FIX-DIRTY(v), this implies $k \geq \text{BASE}(v) = p$. From Lemma 6 we have that the cost of FIX-DIRTY(v) is $O(b_v \alpha^{k+1})$.

Finding BASE(v). Recall that the $\text{BASE}(v) = \ell(v)$ for a node v which is not FULL. Suppose a node v at level i becomes FULL at time step t. We scan for $k = i$ to L till we find $\mathcal{M}_v(k) \neq \emptyset$. If v is not FULL-FROM-ABOVE, then $\mathcal{M}_v(i) \neq \emptyset$. Thus we set $\text{BASE}(v) = i$. Else if v is FULL-FROM-ABOVE, then we set $\text{BASE}(v) = k > i$.

Cost of finding BASE(v). When v is not FULL-FROM-ABOVE, we immediately (in O(1) time) find that $\text{BASE}(v) = \ell(v)$. Suppose that v is FULL-FROM-ABOVE. This implies that at time step $t' \leq t$, a node u at a level $j > i$ matched the edge (u, v) during RANDOM-SETTLE(u) in the FIX-DIRTY(u) subroutine and the node v becomes FULL (see Section 2.1). Since the edge $(u, v) \in \mathcal{M}_v$, we have that $\text{BASE}(v) \leq j$. Thus when scanning the levels from i we do not go any further than level j. So we find BASE(v) in j time. We claim that this cost is subsumed by the cost of FIX-DIRTY(u). This is because, the call to FIX-DIRTY(u) costs $O(b_u \alpha^{j+1})$ and the total cost of finding the BASE of each new mate of u is at most $b_u \cdot j$ (since there are at most b_u many new mates).

C Changing the level of a node

The data structures that essentially captures the levels of nodes and edges have to be updated whenever a node v changes its level during a call to FIX-DIRTY(v). Let i be the level of v before the call to FIX-DIRTY(v) and let j be the level of v just after. Let us consider the two cases, (i) the node v moves to level $j > i$ and (ii) the node v moves to level $j < i$.

(i) For each edge $(u, v) \in \mathcal{E}_v^j$ we remove (u, v) from $\mathcal{O}_u$. For all $i \leq k < j$ and all edges $e \in \mathcal{E}_v^k$, we remove e from $\mathcal{E}_v^k$, and add e to $\mathcal{E}_v^j$ and $\mathcal{O}_v$. Then for all edges $(u, v) \in \mathcal{O}_v$, we remove (u, v) from $\mathcal{E}_u^{(\max\{\ell(u), i\})}$ and $\mathcal{O}_u$. For all edges $(u, v) \in \mathcal{O}_v$, we add (u, v) to $\mathcal{E}_u^j$. Finally we set $\ell(v) = j$.

(ii) For all edges $(u, v) \in \mathcal{O}_v$ with $\ell(u) \geq j$, remove (u, v) from $\mathcal{O}_v$, $\mathcal{E}_v^i$ and $\mathcal{E}_u^i$, and add (u, v) to $\mathcal{E}_v^{\ell(u)}$ and $\mathcal{E}_u^{\ell(u)}$. For all the other edges $(u, v) \in \mathcal{O}_v$, remove (u, v) from $\mathcal{E}_v^i$ and $\mathcal{E}_u^i$, and add (u, v) to $\mathcal{E}_v^j$ and $\mathcal{E}_u^j$. Then for all $j < k \leq i$ and for all edges $(u, v) \in \mathcal{E}_v^k$, add (u, v) to $\mathcal{O}_u$. Finally we set $\ell(v) = j$.

C.1 Finding a new level

Let v be a DIRTY node at level i. Depending on the number of neighbours of v at levels $\leq i$, we might move v to a level j above or below. Let us consider the two cases.

I If a DIRTY node v at level i has $|\mathcal{E}_v^i| > 2b_v\alpha^{i+1}$,then we find the smallest level $j > i$ such that $\sum_{k=i}^{j} |\mathcal{E}_v^k| \leq 2b_v\alpha^{j+1}$ as follows. Let $j = i$ and $C = |\mathcal{E}_v^j|$. While $C > 2b_v\alpha^{j+1}$, increment j by 1 and set $C = C + |\mathcal{E}_v^j|$. When the while loop ends (it will end because $b_v\alpha^{L+1} > n$) the value of j gives the required level.

II Else if a DIRTY node v at level i has $|\mathcal{O}_v| \leq 2b_v\alpha^i$, we move v to biggest level $j < i$ such that $|\mathcal{O}_v|$ becomes $> 2b_v\alpha^j$. Let $j = i$ and $C = |\mathcal{O}_v|$. While ($C \leq 2b_v\alpha^j$ and $j \geq 0$) , decrement j by 1 and give up the ownership of the edges define by $C' := |(u, v) \in \mathcal{O}_v : \ell(u) = j|$ to the other endpoints and set $C = C - C'$. When the while loop ends the value of j gives the required level.