SIAM J. Comput.
Vol. 47, No. 3, pp. 859–887

DETERMINISTIC FULLY DYNAMIC DATA STRUCTURES FOR VERTEX COVER AND MATCHING[*]

SAYAN BHATTACHARYA[†], MONIKA HENZINGER[‡], AND GIUSEPPE F. ITALIANO[§]

Abstract. We present the first deterministic data structures for maintaining approximate minimum vertex cover and maximum matching in a fully dynamic graph $G = (V, E)$, with $|V| = n$ and $|E| = m$, in $o(\sqrt{m})$ time per update. In particular, for minimum vertex cover, we provide deterministic data structures for maintaining a $(2+\epsilon)$ approximation in $O(\log n/\epsilon^2)$ amortized time per update. For maximum matching, we show how to maintain a $(3+\epsilon)$ approximation in $O(\min(\sqrt{n}/\epsilon, m^{1/3}/\epsilon^2)$ *amortized* time per update and a $(4 + \epsilon)$ approximation in $O(m^{1/3}/\epsilon^2)$ *worst-case* time per update. Our data structure for fully dynamic minimum vertex cover is essentially near-optimal and settles an open problem by Onak and Rubinfeld [in *42nd ACM Symposium on Theory of Computing*, Cambridge, MA, ACM, 2010, pp. 457–464].

Key words. fully dynamic algorithms, minimum vertex conver, maximum matching, dynamic data structures, primal-dual method

AMS subject classification. 05C85

DOI. 10.1137/140998925

1. Introduction.

Finding maximum matchings and minimum vertex covers in undirected graphs are classical problems in combinatorial optimization. Let $G = (V, E)$ be an undirected graph, with $m = |E|$ edges and $n = |V|$ nodes. A *matching* in G is a set of vertex-disjoint edges; i.e., no two edges share a common vertex. A *maximum matching*, also known as maximum cardinality matching, is a matching with the largest possible number of edges. A matching is *maximal* if it is not a proper subset of any other matching in G. A subset $V' \subseteq V$ is a *vertex cover* if each edge of G is incident to at least one vertex in V'. A *minimum vertex cover* is a vertex cover of smallest possible size.

The Micali–Vazirani algorithm for maximum matching runs in $O(m\sqrt{n}\,)$ time [13, 18]. Using this algorithm, a $(1 + \epsilon)$-approximate maximum matching can be constructed in $O(m/\epsilon)$ time [8]. The same bound can also be obtained using scaling algorithms [9, 10]. Finding a minimum vertex cover, on the other hand, is NP-hard. Still, these two problems remain closely related as their LP-relaxations are duals of each other. Furthermore, a maximal matching, which can be computed in $O(m)$ time in a greedy fashion, is known to provide a 2-approximation both to maximum matching and to minimum vertex cover (by using the endpoints of the maximal matching).

[*]Received by the editors December 5, 2014; accepted for publication (in revised form) March 28, 2018; published electronically June 20, 2018. An extended abstract of this paper appeared in ACM-SIAM Symposium on Discrete Algorithms (SODA) 2015.
http://www.siam.org/journals/sicomp/47-3/99892.html
Funding: The first author was supported by the European Research Council under the European Union's Seventh Framework Programe (FP7/2007-2013) / ERC Grant Agreement 340506. The second author was supported by the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement 317532 and from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC Grant Agreement 340506. The third author was partially supported by MIUR, the Italian Ministry of Education, University and Research, under Project AMANDA (Algorithmics for MAssive and Networked DAta).
[†]Faculty of Computer Science, University of Vienna, Austria. Current address: University of Warwick, United Kingdom (S.Bhattacharya@warwick.ac.uk).
[‡]Faculty of Computer Science, University of Vienna, Austria (monika.henzinger@univie.ac.at).
[§]Università di Roma "Tor Vergata," Rome, Italy (giuseppe.italiano@uniroma2.it).

Under the unique games conjecture, the minimum vertex cover cannot be efficiently approximated within any constant factor better than 2 [15]. Thus, under the unique games conjecture, the 2-approximation in $O(m)$ time is the optimal guarantee for this problem.

In this paper, we consider a dynamic setting, where the input graph is being updated via a sequence of edge insertions/deletions. The goal is to design data structures that are capable of maintaining the solution to an optimization problem faster than recomputing it from scratch after each update. If $P \neq NP$, we cannot achieve polynomial time updates for minimum vertex cover. We also observe that achieving fast update times for maximum matching appears to be a particularly difficult task: In this case, an update bound of $O(\text{poly}\log(n))$ would be a breakthrough since it would immediately improve the long-standing bounds of various static algorithms [13, 17–19]. The best-known update bound for dynamic maximum matching is obtained by a randomized data structure of Sankowski [22], which has $O(n^{1.495})$ time per update. In this scenario, if one wishes to achieve fast update times for dynamic maximum matching or minimum vertex cover, approximation appears to be inevitable. Indeed, in the past few years there has been a growing interest in designing efficient dynamic data structures for maintaining approximate solutions to both these problems.

1.1. Previous work. A maximal matching can be maintained in $O(n)$ worst-case update time by a trivial deterministic algorithm. Ivković and Lloyd [14] showed how to improve this bound to $O((n+m)^{\sqrt{2}/2})$. Onak and Rubinfeld [21] designed a randomized data structure that maintains constant factor approximations to maximum matching and to minimum vertex cover in $O(\log^2 n)$ amortized time per update with high probability, with the approximation factors being large constants. Baswana, Gupta, and Sen [2] improved these bounds by showing that a maximal matching, and thus a 2-approximation of maximum matching and minimum vertex cover, can be maintained in a dynamic graph in amortized $O(\log n)$ update time with high probability.

Subsequently, turning to deterministic data structures, Neiman and Solomon [20] showed that a 3/2-approximate maximum matching can be maintained dynamically in $O(\sqrt{m})$ worst-case time per update. They maintain a maximal matching and thus achieve the same update bound also for 2-approximate minimum vertex cover. Furthermore, Gupta and Peng [12] presented a deterministic data structure to maintain a $(1+\epsilon)$ approximation of a maximum matching in $O(\sqrt{m}/\epsilon^2)$ worst-case time per update. We also note that Onak and Rubinfeld [21] gave a deterministic data structure that maintains an $O(\log n)$-approximate minimum vertex cover in $O(\log^2 n)$ amortized update time. See Table 1 for a summary of these results.

Very recently, Abboud and Williams [1] showed a conditional lower bound on the performance of any dynamic matching algorithm. There exists an integer $k \in [2, 10]$ with the following property: If the dynamic algorithm maintains a matching with the property that every augmenting path in the input graph (w.r.t. the matching) has length at least $(2k-1)$, then an amortized update time of $o(m^{1/3})$ for the algorithm will violate the 3-SUM conjecture (which states that the 3-SUM problem on n numbers cannot be solved in $o(n^2)$ time).

1.2. Our results. From the above discussion, it is clear that for both fully dynamic constant approximate maximum matching and minimum vertex cover, there is a huge gap between state-of-the-art deterministic and randomized performance guarantees: The former gives $O(\sqrt{m})$ update time, while the latter gives $O(\log n)$ update time. Thus, it seems natural to ask whether the $O(\sqrt{m})$ bound achieved

TABLE 1
Dynamic data structures for approximate (integral) maximum matching (MM) and minimum vertex cover (MVC).

Problem	**Approximation guarantee**	**Update time**	**Data structure**	**Reference**
MM & MVC	$O(1)$	$O(\log^2 n)$ amortized	Randomized	[21]
MM & MVC	2	$O(\log n)$ amortized	Randomized	[2]
MM	1.5	$O(\sqrt{m})$ worst-case	Deterministic	[20]
MVC	2	$O(\sqrt{m})$ worst-case	Deterministic	[20]
MM	$1+\epsilon$	$O(\sqrt{m}/\epsilon^2)$ worst-case	Deterministic	[12]
MVC	$2+\epsilon$	$O(\log n/\epsilon^2)$ amortized	Deterministic	This paper
MM	$3+\epsilon$	$O(\sqrt{n}/\epsilon)$ amortized	Deterministic	This paper
MM	$3+\epsilon$	$O(m^{1/3}/\epsilon^2)$ amortized	Deterministic	This paper
MM	$4+\epsilon$	$O(m^{1/3}/\epsilon^2)$ worst-case	Deterministic	This paper

in [12, 20] is a natural barrier for deterministic data structures. In particular, in their pioneering work on these problems, Onak and Rubinfeld [21] asked the following:

- "*Is there a deterministic data structure that achieves a constant approximation factor with polylogarithmic update time?*"

We answer this question in the affirmative by presenting a deterministic data structure that maintains a $(2 + \epsilon)$-approximation of a minimum vertex cover in $O(\log n/\epsilon^2)$ amortized time per update. Since it is impossible to get better than 2-approximation for minimum vertex cover in polynomial time, our data structure is near-optimal (under the unique games conjecture). As a by-product of our approach, we can also maintain, deterministically, a $(2 + \epsilon)$-approximate maximum *fractional* matching in $O(\log n/\epsilon^2)$ amortized update time. Note that the vertices of the fractional matching polytope of a graph are known to be half integral; i.e., they have only $\{0, 1/2, 1\}$ coordinates (see, e.g., [16]). This implies immediately that the value of any fractional matching is at most $3/2$ times the value of the maximum integral matching. Thus, it follows that we can maintain *the value* of the maximum (integral) matching within a factor of $(2+\epsilon)\cdot(3/2) = (3+O(\epsilon))$, deterministically, in $O(\log n/\epsilon^2)$ amortized update time.

Next, we focus on the problem of maintaining an integral matching in a dynamic setting. For this problem, we show how to maintain a $(3 + \epsilon)$-approximate maximum matching in $O(\min(\sqrt{n}/\epsilon, m^{1/3}/\epsilon^2))$ amortized time per update and a $(4 + \epsilon)$-approximate maximum matching in $O(m^{1/3}/\epsilon^2)$ worst-case time per update. Since $m^{1/3} = o(n)$, we provide the first deterministic data structures for dynamic matching whose update time is sublinear in the number of nodes. Table 1 puts our main results in perspective with previous work.

1.3. Our techniques. To see why it is difficult to deterministically maintain a dynamic (say maximal) matching, consider the scenario when a matched edge incident to a node u gets deleted from the graph. To recover from this deletion, we have to scan through the adjacency list of u to check if it has any free neighbor z. This takes time proportional to the degree of u, which can be $O(n)$. Both the papers [2, 21] use randomization to circumvent this problem. Roughly speaking, the idea is to match the node u to one of its free neighbors z picked *at random* and show that even if this step takes $O(\deg(u))$ time, in expectation the newly matched edge (u, z) survives the next $\deg(u)/2$ edge deletions in the graph (assuming that the adversary is not aware of the random choices made by the data structure). This is used to bound the amortized update time.

Our key insight is that we can maintain a large *fractional matching* deterministically. Suppose that in this fractional matching, we pick each edge incident to u to an extent of $1/\deg(u)$. These edges together contribute at most one to the objective. Thus, we do not have to do anything for the next $\deg(u)/2$ edge deletions incident to u, as these deletions reduce the contribution of u towards the objective by at most a factor of two. This gives us the desired amortized bound. Inspired by this observation, we take a closer look at the framework of Onak and Rubinfeld [21]. Roughly speaking, they maintain a hierarchical partition of the set of nodes V into $O(\log n)$ levels such that the nodes in all but the lowest level, taken together, form a valid vertex cover V^*. In addition, they maintain a matching M^* as a *dual certificate*. Specifically, they show that $|V^*| \leq \lambda \cdot |M^*|$ for some constant λ, which implies that V^* is a λ-approximate minimum vertex cover. Their data structure is randomized since, as discussed above, it is particularly difficult to maintain the matching M^* deterministically in a dynamic setting. To make the data structure deterministic, instead of M^*, we maintain a *fractional matching* as a dual certificate. Along the way, we improve the amortized update time of [21] from $O(\log^2 n)$ to $O(\log n/\epsilon^2)$ and their approximation guarantee from some large constant λ to $2+\epsilon$.

Our approach gives near-optimal bounds for fully dynamic minimum vertex cover, and, as we have already remarked, it maintains a *fractional matching*. Next, we consider the problem of maintaining an approximate maximum *integral matching*, for which we are able to provide deterministic data structures with improved (polynomial) update time. Towards this end, we introduce the concept of a *kernel* of a graph, which we believe is of independent interest. Intuitively, a kernel is a subgraph with two important properties: (i) each node has bounded degree in the kernel, and (ii) a kernel approximately preserves the size of the maximum matching in the original graph. Our key contribution is to show that a kernel always exists and that it can be maintained efficiently in a dynamic graph undergoing a sequence of edge updates.

1.4. Subsequent work. Subsequent to the publication of the conference version of this paper, there has been multiple follow-up work on this problem. Solomon [23] obtained a randomized algorithm for maintaining a maximal matching in $O(1)$ amortized update time. As far as deterministic data structures are concerned, Bernstein and Stein [3, 4] showed how to maintain a $(3/2+\epsilon)$-approximate maximum matching in $O(m^{1/4}/\epsilon^{2.5})$ amortized update time. Bhattacharya, Henzinger, and Nanongkai [6, 7] gave an algorithm for maintaining a $(2+\epsilon)$-approximate maximum (integral) matching in $O(\text{poly}(\log n, 1/\epsilon))$ amortized update time and an algorithm for maintaining a $(2+\epsilon)$-approximate maximum fractional matching and minimum vertex cover in $O(\log^3 n/\text{poly}(1/\epsilon))$ worst-case update time. Finally, Bhattacharya, Chakrabarty, and Henzinger [5] and Gupta et al. [11] showed how to maintain a $O(1)$-approximate maximum fractional matching and minimum vertex cover in $O(1)$ amortized update time.

2. Deterministic fully dynamic vertex cover.

In this section, we consider the following dynamic setting. An input graph $G=(V,E)$ has $|V|=n$ nodes and zero edges in the beginning. Subsequently, it keeps getting updated due to the insertions of new edges and the deletions of existing edges. The edge updates, however, occur one at a time, while the set V remains fixed. The goal is to maintain an approximate vertex cover of G in this fully dynamic setting.

The rest of this section is organised as follows. In section 2.1, we introduce the notion of an (α,β)-partition of $G=(V,E)$. This is a hierarchical partition of the set V into $L+1$ levels, where $L=\lceil \log_\beta(n/\alpha)\rceil$ and $\alpha,\beta>1$ are two parameters (Definition 2.2). If the (α,β)-partition satisfies an additional property (Invariant 2.4), then from it we can easily derive a $2\alpha\beta$-approximation to the minimum vertex cover (Theorem 2.5). In section 2.2, we present the relevant data structures that we use to maintain an (α, β)-partition. In section 2.3, we present a natural deterministic algorithm for maintaining such an (α, β)-partition. Finally, in section 2.4, we analyze the amortized update time of the algorithm using a carefully chosen potential function. We show that for $\alpha = 1+3\epsilon$ and $\beta = 1+\epsilon$, the algorithm takes $O\left((t/\epsilon)\log_{1+\epsilon} n\right)$ time to handle t edge updates starting from an empty graph (Theorem 2.8). This leads to the main result of this section, which is stated in the theorem below.

THEOREM 2.1. *For every $\epsilon \in (0,1)$, we can deterministically maintain a $(2+\epsilon)$-approximate minimum vertex cover in a fully dynamic graph, the amortized update time being $O(\log n/\epsilon^2)$.*

2.1. The (α, β)-partition and its properties.

DEFINITION 2.2. *An* (α, β)-partition *of the graph G partitions its node set V into subsets $V_0 \cdots V_L$, where $L = \lceil \log_\beta(n/\alpha) \rceil$ and $\alpha, \beta > 1$. For $i \in \{0, \dots, L\}$, we identify the subset V_i as the ith* level *of this partition and denote the* level of a node v *by $\ell(v)$. Thus, we have $v \in V_{\ell(v)}$ for all $v \in V$. Furthermore, the partition assigns a weight $w(u,v) = \beta^{-\max(\ell(u),\ell(v))}$ to every edge $(u,v) \in V$.*

Define $\mathcal{N}_v$ to be the set of neighbors of a node $v \in V$. Given an (α, β)-partition, let $\mathcal{N}_v(i) \subseteq \mathcal{N}_v$ denote the set of neighbors of v that are in the ith level, and let $\mathcal{N}_v(i,j) \subseteq \mathcal{N}_v$ denote the set of neighbors of v whose levels are in the range $[i,j]$:

$$\mathcal{N}_v = \{u \in V : (u,v) \in E\} \quad \text{for all } v \in V \tag{1}$$

$$\mathcal{N}_v(i) = \{u \in \mathcal{N}_v \cap V_i\} \quad \text{for all } v \in V; i \in \{0, \dots, L\} \tag{2}$$

$$\mathcal{N}_v(i,j) = \bigcup_{k=i}^{j} \mathcal{N}_v(k) \quad \text{for all } v \in V; i, j \in \{0, \dots, L\}, i \le j. \tag{3}$$

Similarly, define the notations D_v, $D_v(i)$, and $D_v(i,j)$. Note that D_v is the degree of a node $v \in V$:

$$D_v = |\mathcal{N}_v| \tag{4}$$

$$D_v(i) = |\mathcal{N}_v(i)| \tag{5}$$

$$D_v(i,j) = |\mathcal{N}_v(i,j)|. \tag{6}$$

Given an (α, β)-partition, let $W_v = \sum_{u \in \mathcal{N}_v} w(u,v)$ denote the total weight a node $v \in V$ receives from the edges incident to it. We also define the notation $W_v(i)$. It gives the total weight the node v would receive from the edges incident to it *if the node v itself was to go to the ith level.* Thus, we have $W_v = W_v(\ell(v))$. Since the weight of an edge (u,v) in the hierarchical partition is given by $w(u,v) = \beta^{-\max(\ell(u),\ell(v))}$, we derive the following equations for all nodes $v \in V$:

$$W_v = \sum_{u \in \mathcal{N}_v} \beta^{-\max(\ell(u),\ell(v))} \tag{7}$$

$$W_v(i) = \sum_{u \in \mathcal{N}_v} \beta^{-\max(\ell(u),i)} \quad \text{for all } i \in \{0, \dots, L\}. \tag{8}$$

Fix a node $v \in V$, and focus on the value of $W_v(i)$ as we go down from the highest level $i = L$ to the lowest level $i = 0$. Lemma 2.3 states that $W_v(i) \le \alpha$ when $i = L$, that $W_v(i)$ keeps increasing as we go down the levels one after another, and that $W_v(i)$ increases by at most a factor of β between consecutive levels.

LEMMA 2.3. *Every (α, β)-partition of the graph G satisfies the following conditions for all nodes $v \in V$:*

$$W_v(L) \leq \alpha \tag{9}$$

$$W_v(L) \leq \cdots \leq W_v(i) \leq \cdots \leq W_v(0) \tag{10}$$

$$W_v(i) \leq \beta \cdot W_v(i+1) \quad \textit{for all } i \in \{0, \ldots, L-1\}. \tag{11}$$

Proof. Fix any (α, β)-partition and any node $v \in V$. We prove the first part of the lemma as follows:

$$\begin{aligned} W_v(L) &= \sum_{u \in \mathcal{N}_v} \beta^{-\max(\ell(u), L)} \\ &= \sum_{u \in \mathcal{N}_v} \beta^{-L} \leq n \cdot \beta^{-L} \leq n \cdot \beta^{-\log_\beta(n/\alpha)} = \alpha. \end{aligned}$$

We now fix any level $i \in \{0, \ldots, L-1\}$ and show that the (α, β)-partition satisfies (10):

$$\begin{aligned} W_v(i+1) &= \sum_{u \in \mathcal{N}_v} \beta^{-\max(\ell(u), i+1)} \\ &\leq \sum_{u \in \mathcal{N}_v} \beta^{-\max(\ell(u), i)} = W_v(i). \end{aligned}$$

Finally, we prove (11):

$$\begin{aligned} W_v(i) &= \sum_{u \in \mathcal{N}_v} \beta^{-\max(\ell(u), i)} = \beta \cdot \sum_{u \in \mathcal{N}_v} \beta^{-1-\max(\ell(u), i)} \\ &\leq \beta \cdot \sum_{u \in \mathcal{N}_v} \beta^{-\max(\ell(u), i+1)} = \beta \cdot W_v(i+1). \end{aligned}$$ ◻

We will maintain a specific type of (α, β)-partition, where each node is assigned to a level in a way that satisfies Invariant 2.4.

INVARIANT 2.4. *For every node $v \in V$, if $\ell(v) = 0$, then $W_v \leq \alpha \cdot \beta$. Else, if $\ell(v) \geq 1$, then $W_v \in [1, \alpha\beta]$.*

Consider any (α, β)-partition satisfying Invariant 2.4. Let $v \in V$ be a node in this partition that is at level $\ell(v) = k \in \{0, \ldots, L\}$. It follows that $\sum_{u \in \mathcal{N}_v(0,k)} w(u, v) = |\mathcal{N}_v(0, k)| \cdot \beta^{-k} \leq W_v \leq \alpha\beta$. Thus, we infer that $|\mathcal{N}_v(0, k)| \leq \alpha\beta^{k+1}$. In other words, Invariant 2.4 gives an upper bound on the number of neighbors a node v can have that lie on or below $\ell(v)$. We will crucially use this property in the analysis of our algorithm.

THEOREM 2.5. *Consider an (α, β)-partition of the graph G that satisfies Invariant 2.4. Let $V^* = \{v \in V : W_v \geq 1\}$ be the set of nodes with weight at least one. The set V^* is a feasible vertex cover in G. Further, the size of the set V^* is at most $2\alpha\beta$ times the size of the minimum-cardinality vertex cover in G.*

Proof. Consider any edge $(u, v) \in E$. We claim that at least one of its endpoints belong to the set V^*. Suppose that the claim is false and we have $W_u < 1$ and $W_v < 1$. If this is the case, then Invariant 2.4 implies that $\ell(u) = \ell(v) = 0$ and $w(u, v) = \beta^{-\max(\ell(u), \ell(v))} = 1$. Since $W_u \geq w(u, v)$ and $W_v \geq w(u, v)$, we get $W_u \geq 1$

and $W_v \geq 1$, and this leads to a contradiction. Thus, we infer that the set V^* is a feasible vertex cover in the graph G.

Next, we construct a *fractional matching* M_f by picking $x(u,v) = w(u,v)/(\alpha\beta)$ copies of every edge $(u,v) \in E$. Since for all nodes $v \in V$ we have $\sum_{u \in \mathcal{N}_v} x(u,v) = \sum_{u \in \mathcal{N}_v} w(u,v)/(\alpha\beta) = W_v/(\alpha\beta) \leq 1$, we infer that M_f is a valid fractional matching in G. The size of this matching is given by $|M_f| = \sum_{(u,v)\in E} x(u,v) = (1/(\alpha\beta)) \cdot \sum_{(u,v)\in E} w(u,v)$. We now bound the size of V^* in terms of $|M_f|$:

$$\begin{aligned} |V^*| &= \sum_{v \in V^*} 1 \leq \sum_{v \in V^*} W_v = \sum_{v \in V^*} \sum_{u \in \mathcal{N}_v} w(u,v) \\ &\leq \sum_{v \in V} \sum_{u \in \mathcal{N}_v} w(u,v) = 2 \cdot \sum_{(u,v)\in E} w(u,v) = (2\alpha\beta) \cdot |M_f|. \end{aligned}$$

The approximation guarantee now follows from the LP duality between minimum fractional vertex cover and maximum fractional matching. ◻

Query time. We store the nodes v with $W_v \geq 1$ as a separate list. Thus, we can report the set of nodes in the vertex cover in $O(1)$ time per node. Using appropriate pointers, we can report in $O(1)$ time whether a given node is part of this vertex cover. In $O(1)$ time, we can also report the size of the vertex cover.

2.2. Data structures. We now describe the data structures that we will use to maintain an (α,β)-partition in a dynamic graph:

- A counter to keep track of the current value of $\ell(v)$
- A counter to keep track of the current value of W_v
- For every level $i > \ell(v)$, the set of nodes $\mathcal{N}_v(i)$ as a doubly linked list
- For level $i = \ell(v)$, the set of nodes $\mathcal{N}_v(0,i)$ as a doubly linked list

The insertion/deletion of an edge (u,v) in G changes the weights of its endpoints $\{u,v\}$. Thus, after the insertion/deletion of an edge (u,v) in G, we might discover that one or both of its endpoints violates Invariant 2.4. Such a node (which violates Invariant 2.4) is called *dirty*, and we store the set of dirty nodes as a doubly linked list. For every node $v \in V$, we maintain a bit $\textsc{Status}[v] \in \{\text{dirty}, \text{clean}\}$ that indicates whether the node is dirty. Every dirty node stores a pointer to its position in the list of dirty nodes.

The phrase *"neighborhood lists of v"* refers to the set $\bigcup_{i=\ell(v)+1}^{L} \mathcal{N}_v(i) \bigcup \mathcal{N}_v$ $(0,\ell(v))$. For every edge (u,v), we maintain two bidirectional pointers: One links the edge to the position of v in the neighborhood lists of u, while the other links the edge to the position of u in the neighborhood lists of v. These pointers allow us to insert or delete any edge from any list in constant time.

2.3. Handling the insertion/deletion of an edge. Recall that a node is called *dirty* if it violates Invariant 2.4 and *clean* otherwise. Since the graph $G = (V,E)$ is initially empty, every node is clean and at level zero before the first update in G. Now consider the time instant just prior to the *tth* update in G. By the induction hypothesis, at this instant every node is clean. Then the *tth* update takes place, which inserts (resp., deletes) an edge (x,y) in G with weight $w(x,y) = \beta^{-\max(\ell(x),\ell(y))}$. This increases (resp., decreases) the weights W_x, W_y by $w(x,y)$. Due to this change, the nodes x and y might become dirty. To recover from this, we run the $\textsc{While}$ loop in Figure 1, which repeatedly tries to fix the dirty nodes using the following simple greedy heuristic: Whenever it finds a dirty node whose weight is too large (resp., small), it increments (resp., decrements) the level of that node.

```
01. WHILE there exists a dirty node $v$
02.     IF $W_v > \alpha\beta$, THEN                // If true, then by (9) we have $\ell(v) < L$.
03.         Increment the level of $v$ by setting $\ell(v) \leftarrow \ell(v) + 1$, and update the
              relevant data structures.
04.     ELSE IF ($W_v < 1$ and $\ell(v) > 0$), THEN
05.         Decrement the level of $v$ by setting $\ell(v) \leftarrow \ell(v) - 1$, and update the
              relevant data structures.
```

FIG. 1. *Handling the dirty nodes.*

Consider any iteration of the WHILE loop in Figure 1, which tries to fix a dirty node $v \in V$. Suppose that $W_v = W_v(\ell(v)) > \alpha\beta$. In this event, (9) implies that $W_v(L) < W_v(\ell(v))$, and hence we have $L > \ell(v)$. In other words, when the procedure described in Figure 1 decides to increment the level of a dirty node v (step 03), we know for sure that the current level of v is strictly less than L (the highest level in the (α, β)-partition). Next, consider a node $z \in \mathcal{N}_v$. If we change $\ell(v)$, then this may change the weight $w(v, z)$, and this in turn may change the weight W_z. Thus, a single iteration of the WHILE loop in Figure 1 may lead to some clean nodes becoming dirty and some other dirty nodes becoming clean. If and when the WHILE loop terminates, however, we are guaranteed that every node is clean and that Invariant 2.4 holds.

LEMMA 2.6. *Suppose that some iteration of the* WHILE *loop in Figure* 1 *increases the level of a node v from i to $i+1$. Then it takes $\Theta(1 + D_v(0, i))$ time to update the relevant data structures during this iteration.*

Proof. When the node v moves up from level i to level $i + 1$, we need to update v's position in the neighborhood lists of a node $u \in \mathcal{N}_v$ iff $\ell(u) \leq i$. On the other hand, we can update the neighborhood lists of v itself in constant time by simply noting that $\mathcal{N}_v(0, i+1) = \mathcal{N}_v(0, i) \cup \mathcal{N}_v(i + 1)$. Hence, the total time spent on this iteration is proportional to $1 + D_v(0, i)$, where the term 1 comes from the fact that it takes constant time to implement the iteration even if $D_v(0, i)$. ◻

LEMMA 2.7. *Suppose that some iteration of the* WHILE *loop in Figure* 1 *decreases the level of a node v from i to $i-1$. Then it takes $\Theta(1 + D_v(0, i))$ time to update the relevant data structures during this iteration.*

Proof. When the node v moves down from level i to level $i-1$, we need to update v's position in the neighborhood lists of a node $u \in \mathcal{N}_v$ iff $\ell(u) < i$. However, when v is at level i, our data structure keeps all the nodes $u \in \mathcal{N}_v$ with $\ell(u) \leq i$ in one doubly linked list $\mathcal{N}_v(0, i)$. Thus, when v moves down from level i to level $i - 1$, we need to scan through the nodes in the list $\mathcal{N}_v(0, i)$ one after the other. While considering any node $u \in \mathcal{N}_v(0, i)$, we need to check if $\ell(u) < i$, and, if yes, then we need to update v's position in the neighborhood lists of u. We can update the neighborhood lists of v itself during the same scan since basically all we need to do is to split the list $\mathcal{N}_v(0, i)$ into two parts: $\mathcal{N}_v(0, i - 1)$ and $\mathcal{N}_v(i)$. Accordingly, the total time spent on this iteration is proportional to $1 + D_v(0, i)$, where the term 1 comes from the fact that it takes constant time to implement the iteration even if $D_v(0, i)$. ◻

Bounding the amortized update time: An overview. It is by no means obvious that the WHILE loop in Figure 1 will eventually terminate after a finite number of steps. Thus, it is somewhat surprising that in section 2.4 we are able to prove the following guarantee: The algorithm described above for handling the insertion/deletion of an edge in the input graph has an amortized update time $O(\log n/\epsilon^2)$ for $\alpha = 1 + 3\epsilon$ and $\beta = \epsilon$. We now give a high-level intuitive explanation for this result.

For simplicity, we assume that both α and β are very large constants. Furthermore, we consider the following modification of the While loop in Figure 1. If $W_v > \alpha\beta$, then we move the node v up to a level i, where $W_v(i) \in [\beta, \alpha\beta]$. On the other hand, if $W_v < 1$ and $\ell(v) > 0$, then we move v down to a level i, where $W_v(i) \in [\beta, \alpha\beta]$. If no such level i, exists, then we move v down to level 0, where it must have $W_v(0) \in [0, \alpha\beta]$. We can change the level of v in this way due to Lemma 2.3, which implies that (1) the weight W_v changes by at most a factor of β when we change the level of v by one and (2) $W_v(L) \leq \alpha$. One key property of this modified algorithm is as follows: For every level $k \in [0, L]$, we have $W_v(k) \in [\beta, \alpha\beta]$ whenever a node v moves from any level $i \neq k$ to level k.

Define the *level* of an edge (u, v) as $\ell(u, v) = \max(\ell(u), \ell(v))$, and note that the weight of an edge is completely determined by its level. As a proxy for the actual update time, we focus on upper bounding the number of times the modified algorithm changes the level (and therefore the weight) of an edge in the (α, β)-partition. We refer to this quantity as the *update cost* of the modified algorithm and explain informally why the amortized update cost is $O(L) = O(\log n)$.

If it were the case that the modified algorithm never decreases the level of an edge in the (α, β)-partition, then we would immediately get an amortized update cost of $O(L)$ for the following reason. The level of an edge can take only $L + 1$ different values. Therefore, during a sequence of t edge insertions/deletions starting from an empty graph, we would see the level of an edge increasing at most $t(L + 1)$ times, leading to an amortized update cost of $O(L)$. Unfortunately for us, this claim is too good to be true. The level of an edge can also decrease during the course of our modified algorithm, which breaks the previous argument. Nevertheless, we can show that if we look at a sufficiently long time interval, then the number of times the level of an edge drops is significantly smaller than the number of times the level of an edge increases plus the number of edge deletions. Then using a more sophisticated version of the previous argument, we derive that the amortized update cost is $O(L)$. It now remains to explain why the number of level decreases of edges is significantly smaller than the number of level increases plus the number of edge deletions.

The level of an edge decreases only when one of its endpoints moves down to a lower level. Accordingly, consider a time step t when a node v moves from some level i down to some other level $j < i$. It follows that $W_v(i) < 1$ at time step t. When the node v moves down from level i to level j, it only decreases the levels of the edges $(u, v) \in E$ with $\ell(u) < i$. Just before time step t, each such edge had a weight $1/\beta^{\max(\ell(u),\ell(v))} = 1/\beta^i$, and we had $W_v(i) < 1$. Hence, at time step t, the node v has at most β^i neighbors u with $\ell(u) < i$. In other words, when the node v moves down from level i to level j, at most β^i many edges incident on v decreases their levels. Now, consider the last time step $t' < t$, when the node v moved to level i. In particular, the node v moves to level i at time step t', stays at level i during the time interval $[t', t]$, and moves down to level $j < i$ at time step t. From the description of our modified algorithm, it follows that $W_v(i) \in [\beta, \alpha\beta]$ at time t'. Thus, the weight W_v decreases by at least $\beta - 1$ during the time interval $[t', t]$. This decrease in the weight of v happens because of multiple occurrences of the following two events: (1) An edge (u, v) incident on v gets deleted from the input graph, and (2) an edge (u, v) incident on v has its other endpoint u increase its own level $\ell(u)$ from some $k \leq i$ to some $k' > i$. Each of these events decreases the weight W_v by at most $1/\beta^i$ since the node v remains at level i during the time interval $[t', t]$. Thus, at least $(\beta - 1) \cdot \beta^i$ of these events must take place during the time interval $[t', t]$. To summarize, whenever a node v moves down from a level i, it decreases the levels of at most β^i many incident edges. However, for such an event to take place, at least $(\beta - 1)\beta^i$ edges incident on v must either get deleted or increase their levels. This implies that the number of level decreases of edges is much smaller than the number of level increases plus the number of edge deletions.

Comparison with the framework of Onak and Rubinfeld [21]. As described below, there are two significant differences between our framework and that of [21]. Consequently, many of the technical details of our approach (illustrated in section 2.4) differ from the proof in [21].

First, in the hierarchical partition of [21], the invariant for a node y consists of $O(L)$ constraints: For each level $i \in \{\ell(y), \ldots, L\}$, the quantity $|\mathcal{N}_y(0, i)|$ has to lie within a certain range. This is the main reason for their amortized update time being $\Theta(\log^2 n)$. Indeed, when a node y becomes dirty, unlike in our setting, they have to spend $\Theta(\log n)$ time just to figure out the new level of y.

Second, along with the hierarchical partition, the authors in [21] maintain a matching as a *dual certificate* and show that the size of this matching is within a constant factor of the size of their vertex cover. As pointed out in section 1, this is the part where they crucially need to use randomization, as until now, there is no deterministic data structure for maintaining a large matching in polylog amortized update time. We bypass this barrier by implicitly maintaining a *fractional matching* as a dual certificate. Indeed, the weight $w(y, z)$ of an edge (y, z) in our hierarchical partition, after suitable scaling, equals the fractional extent by which the edge (y, z) is included in our fractional matching.

2.4. Bounding the amortized update time: Detailed analysis. We devote this section to the proof of the following theorem, which bounds the update time of our algorithm. Throughout this section, we set $\alpha \leftarrow 1 + 3\epsilon$ and $\beta \leftarrow 1 + \epsilon$, where $\epsilon \in (0, 1)$ is a small positive constant.

THEOREM 2.8. *The algorithm in section* 2.3 *maintains an* (α, β)*-partition that satisfies Invariant* 2.4. *For every* $\epsilon \in (0, 1), \alpha = 1 + 3\epsilon$, *and* $\beta = 1 + \epsilon$, *the algorithm takes* $O\left((t/\epsilon)\log_{1+\epsilon} n\right)$ *time to handle* t *edge updates starting from an empty graph.*

Consider the following thought experiment. We have a *bank account*, and initially, when there are no edges in the graph, the bank account has zero balance. For each subsequent edge insertion/deletion, at most $20L/\epsilon$ dollars are deposited to the bank account, and for each unit of computation performed by our algorithm, at least one dollar is withdrawn from it. We show that the bank account, never runs out of money, and this gives a running time bound of $O(tL/\epsilon) = O\left((t/\epsilon)\log_\beta(n/\alpha)\right) = O\left((t/\epsilon)\log_{1+\epsilon} n\right)$ for handling t edge updates starting from an empty graph:

Let $\mathcal{B}$ denote the total amount of money (or potential) in the bank account at the present moment. We keep track of $\mathcal{B}$ by distributing an ϵ-fraction of it among the nodes and the current set of edges in the graph:

$$\mathcal{B} = (1/\epsilon) \cdot \left(\sum_{e \in E} \Phi(e) + \sum_{v \in V} \Psi(v) \right). \tag{12}$$

In the above equation, the amount of money (or potential) associated with an edge $e \in E$ is given by $\Phi(e)$, and the amount of money (or potential) associated with a node $v \in V$ is given by $\Psi(v)$. To ease notation, for each edge $e = (u, v) \in E$, we use the symbols $\Phi(e), \Phi(u, v)$, and $\Phi(v, u)$ interchangeably.

We call a node $v \in V$ *passive* if we have $\mathcal{N}_v = \emptyset$ throughout the duration of a time interval that starts at the beginning of the algorithm (when $E = \emptyset$) and ends at the present moment. Let $V_{passive} \subseteq V$ denote the set of all nodes that are currently passive.

At every point in time, all the potentials $\Phi(u,v), \Psi(v)$ are determined by the two invariants stated below.

INVARIANT 2.9. *For every (unordered) pair of nodes $\{u,v\}$, $u,v \in V$, we have*

$$\Phi(u,v) = \begin{cases} (1+\epsilon) \cdot (L - \max(\ell(u), \ell(v))) & \text{if } (u,v) \in E; \\ 0 & \text{if } (u,v) \notin E. \end{cases}$$

INVARIANT 2.10. *For every node $v \in V$, we have*

$$\Psi(v) = \begin{cases} \epsilon \cdot (L - \ell(v)) + \left(\beta^{\ell(v)+1}/(\beta-1)\right) \cdot \max(0, \alpha - W_v) & \text{if } v \notin V_{passive}; \\ 0 & \text{if } v \in V_{passive}. \end{cases}$$

Initialization. When the algorithm starts, the graph has zero edges, all the nodes are at level 0, and every node is passive. At that moment, Invariant 2.10 sets $\Psi(v) = 0$ for all nodes $v \in V$. Consequently, (12) implies that the potential $\mathcal{B}$ is also set to zero. This is consistent with our requirement that initially the bank account ought to have zero balance.

Insertion of an edge. The time taken to handle an edge insertion, ignoring the time spent on the WHILE loop in Figure 1, is $\Theta(1)$. According to our framework, we are allowed to deposit at most $20L/\epsilon$ dollars to the bank account, and a withdrawal of one dollar from the same account is sufficient to pay for the computation performed. Thus, an edge insertion should not increase the potential $\mathcal{B}$ by more than $20L/\epsilon - 1$. We show below that this is indeed the case.

When an edge (u,v) is inserted into the graph, the potential $\Phi(u,v)$ increases by at most $(1+\epsilon)L$. Next, we consider two possible scenarios to bound the increase in $\Psi(v)$:

1. The node v was passive prior to the insertion of the edge. Clearly, in this case, the node is at level zero, and $\Psi(v) = 0$ before the insertion. After the insertion, the node is not passive anymore, and we have $\Psi(v) \le \epsilon L + \alpha\beta/(\beta - 1) = \epsilon L + (1+3\epsilon)(1+\epsilon)/\epsilon \le 9L$. Since $\epsilon \in (0,1)$ and $L = \log_\beta(n/\alpha) = \log_{1+\epsilon}(n/(1+3\epsilon))$, the last inequality holds as long as $n \ge 20$. We conclude that the potential $\Psi(v)$ increases by at most $9L$.
2. The node v was not passive prior to the insertion of the edge. Clearly, in this case, the node v is also not passive afterwards. The weight W_v increases and the quantity $\max(0, \alpha - W_v)$ decreases due to the insertion, which means that the potential $\Psi(v)$ decreases.

Similarly, we conclude that either $\Psi(u)$ increases by at most $9L$ or it actually decreases. The potentials of the remaining nodes and edges do not change. Hence, by (12), the net increase in $\mathcal{B}$ is at most $18L/\epsilon$.

Deletion of an edge. The analysis is very similar to the one described above. The time taken to handle an edge deletion, ignoring the time spent on the WHILE loop in Figure 1, is $\Theta(1)$. According to our framework, we are allowed to deposit at most $20L/\epsilon$ dollars to the bank account, and a withdrawal of one dollar from the same account is sufficient to pay for the computation performed. Thus, an edge deletion should not increase the potential $\mathcal{B}$ by more than $20L/\epsilon - 1$. We show below that this is indeed the case.

When an edge (u,v) is deleted from the graph, the potential $\Phi(u,v)$ decreases. Next, note that the weight W_v decreases by at most $\beta^{-\ell(v)}$, and so the quantity $\max(0, \alpha - W_v)$ increases by at most $\beta^{-\ell(v)}$. As the node v was not passive before the edge deletion, it is also not passive afterwards. Thus, $\Psi(v)$ increases by at most $\left(\beta^{\ell(v)+1}/(\beta-1)\right) \cdot \beta^{-\ell(v)} = 2/\epsilon \leq 2L$. The last equality holds as long as $n \geq 20$. We similarly conclude that $\Psi(u)$ increases by at most $2L$. The potentials of the remaining nodes and edges do not change. Hence, by (12), the net increase in $\mathcal{B}$ is at most $4L/\epsilon$.

It remains to analyze the time spent on the While loop in Figure 1.

2.4.1. Analysis of the time spent on the While loop in Figure 1. Fix any single iteration of the While loop in Figure 1, which changes the level of a dirty node v by one. Throughout this section, we use the phrase *the iteration* to refer to this specific iteration of the While loop in Figure 1. We use the superscript 0 (resp., 1) on a symbol to denote its state at the time instant immediately prior to (resp., after) the iteration. Further, we preface a symbol with δ to denote the net decrease in its value due to this iteration. For example, consider the potential $\mathcal{B}$. We have $\mathcal{B} = \mathcal{B}^0$ immediately before the iteration starts and $\mathcal{B} = \mathcal{B}^1$ immediately after the iteration ends. We also have $\delta\mathcal{B} = \mathcal{B}^0 - \mathcal{B}^1$. We will prove the following theorem.

Theorem 2.11. *We have $\delta\mathcal{B} > 0$ and $\delta\mathcal{B} \geq T$, where T denotes the time spent on the iteration. In other words, the money withdrawn from the bank account during the iteration is sufficient to pay for the computation performed during the iteration.*

The iteration affects only the potentials of the nodes $u \in \mathcal{N}_v \cup \{v\}$ and that of the edges $e \in \{(u,v) : u \in \mathcal{N}_v\}$. This observation, coupled with (12), gives us the following guarantee:

$$\delta\mathcal{B} = (1/\epsilon) \cdot \left(\delta\Psi(v) + \sum_{u \in \mathcal{N}_v} \delta\Phi(u,v) + \sum_{u \in \mathcal{N}_v} \delta\Psi(u) \right). \tag{13}$$

The iteration does not change (a) the neighborhood structure of the node v and (b) the level and the overall degree of any node $u \neq v$. Hence, we get

$$\mathcal{N}_v^0(i) = \mathcal{N}_v^1(i) \quad \text{for all } i \in \{0, \ldots, L\} \tag{14}$$

$$\ell^0(u) = \ell^1(u) \quad \text{for all } u \in V \setminus \{v\} \tag{15}$$

$$D_u^0 = D_u^1 \quad \text{for all } u \in V \setminus \{v\}. \tag{16}$$

Accordingly, to ease notation, we do not put any superscript on the following symbols, as the quantities they refer to remain the same throughout the duration of the iteration:

$$\begin{cases} \mathcal{N}_v, D_v & \\ \mathcal{N}_v(i), D_v(i), W_v(i) & \text{for all } i \in \{0, \ldots, L\} \\ \mathcal{N}_v(i,j), D_v(i,j) & \text{for all } i, j \in \{0, \ldots, L\}, i \leq j \\ \ell(u), D_u & \text{for all } u \in V \setminus \{v\}. \end{cases}$$

Since the node v is dirty just before the iteration, it follows that neither the node v nor the nodes $u \in \mathcal{N}_v$ are passive.[1] Applying Invariant 2.10, we get

(17)
$$\Psi(u) = \epsilon \cdot (L - \ell(u)) + \left(\beta^{\ell(u)+1}/(\beta-1)\right) \cdot \max(0, \alpha - W_u) \quad \text{for all nodes } u \in \mathcal{N}_v \cup \{v\}.$$

[1] A passive node is not adjacent to any edges, and thus its weight is zero and it has no neighbors.

We divide the proof of Theorem 2.11 into two possible cases, depending upon whether the iteration increments or decrements the level of v. The main approach to the proof remains the same in each case. We first give an upper bound on the time T spent on the iteration. Next, we separately lower bound each of the following quantities: $\delta\Psi(v)$, $\delta\Phi(u,v)$ for all $u \in \mathcal{N}_v$ and $\delta\Psi(u)$ for all $u \in \mathcal{N}_v$. Finally, applying (13), we derive that $\delta\mathcal{B} \geq T$.

Case 1*: The iteration increases the level of the node v from k to $(k+1)$.*

LEMMA 2.12. *We have $T \leq 1 + D_v(0,k)$.*

Proof. Follows from Lemma 2.6. ∎

LEMMA 2.13. *We have $\delta\Psi(v) = \epsilon$.*

Proof. Since the iteration increments the level of v, step 02 of Figure 1 guarantees that $W_v^0 = W_v(k) > \alpha\beta$. Next, from Lemma 2.3, we get $W_v^1 = W_v(k+1) \geq \beta^{-1} \cdot W_v(k) > \alpha$. Since both $W_v^0, W_v^1 > \alpha$, we get

$$\Psi^0(v) = \epsilon \cdot (L-k) + \left(\beta^{k+1}/(\beta-1)\right) \cdot \max(0, \alpha - W_v^0) = \epsilon \cdot (L-k),$$
$$\Psi^1(v) = \epsilon \cdot (L-k-1) + \left(\beta^{k+2}/(\beta-1)\right) \cdot \max(0, \alpha - W_v^1) = \epsilon \cdot (L-k-1).$$

It follows that $\delta\Psi(v) = \Psi^0(v) - \Psi^1(v) = \epsilon$. ∎

LEMMA 2.14. *For every node $u \in \mathcal{N}_v$, we have*

$$\delta\Phi(u,v) = \begin{cases} (1+\epsilon) & \textit{if } u \in \mathcal{N}_v(0,k); \\ 0 & \textit{if } u \in \mathcal{N}_v(k+1,L). \end{cases}$$

Proof. If $u \in \mathcal{N}_v(0,k)$, then we have $\Phi^0(u,v) = (1+\epsilon)\cdot(L-k)$, and $\Phi^1(u,v) = (1+\epsilon)\cdot(L-k-1)$. It follows that $\delta\Phi(u,v) = \Phi^0(u,v) - \Phi^1(u,v) = (1+\epsilon)$.

In contrast, if $u \in \mathcal{N}_v(k+1,L)$, then we have $\Phi^0(u,v) = \Phi^1(u,v) = (1+\epsilon)\cdot(L - \ell(u))$. Hence, we get $\delta\Phi(u,v) = \Phi^0(u,v) - \Phi^1(u,v) = 0$. ∎

LEMMA 2.15. *For every node $u \in \mathcal{N}_v$, we have*

$$\delta\Psi(u) \geq \begin{cases} -1 & \textit{if } u \in \mathcal{N}_v(0,k); \\ 0 & \textit{if } u \in \mathcal{N}_v(k+1,L). \end{cases}$$

Proof. Consider any node $u \in \mathcal{N}_v(k+1,L)$. Since $k < \ell(u)$, we have $w^0(u,v) = w^1(u,v)$, and this implies that $W_u^0 = W_u^1$. Thus, we get $\delta\Psi(u) = 0$.

Next, fix any node $u \in \mathcal{N}_v(0,k)$. Note that $\delta W_u = \delta w(u,v) = \beta^{-k} - \beta^{-(k+1)} = (\beta-1)/\beta^{k+1}$. Using this observation and the fact that $\ell(u) \leq k$, we infer that

$$\delta\Psi(u) \geq -\left(\beta^{\ell(u)+1}/(\beta-1)\right) \cdot \delta W_u = -\beta^{\ell(u)+1}/\beta^{k+1} \geq -1.$$ ∎

Proof of Theorem 2.11 *(for Case* 1*).* From Lemmas 2.13, 2.14, and 2.15 and from (13), we derive the following bound:

$$\begin{aligned} \delta\mathcal{B} &= (1/\epsilon) \cdot \left(\delta\Psi(v) + \sum_{u \in \mathcal{N}_v} \delta\Phi(u,v) + \sum_{u \in \mathcal{N}_v} \delta\Psi(u) \right), \\ &\geq (1/\epsilon) \cdot \left(\epsilon + (1+\epsilon)\cdot D_v(0,k) - D_v(0,k)\right), \\ &= 1 + D_v(0,k). \end{aligned}$$

The theorem (for Case 1) now follows from Lemma 2.12. ∎

Case 2: The iteration decreases the level of the node v from k to $k-1$.

Claim 2.16. We have $W_v^0 = W_v(k) < 1$ and $D_v(0,k) \leq \beta^k$.

Proof. As the iteration decrements the level of v, step 04 of Figure 1 ensures that $W_v^0 = W_v(k) < 1$. Since $\ell^0(v) = k$, we have $w^0(u,v) \geq \beta^{-k}$ for all $u \in \mathcal{N}_v$. We conclude that

$$1 > W_v^0 \geq \sum_{u \in \mathcal{N}_v(0,k)} w^0(u,v) \geq \beta^{-k} \cdot D_v(0,k).$$

Thus, we get $D_v(0,k) \leq \beta^k$. ∎

Lemma 2.17. *We have $T \leq \beta^k$.*

Proof. Since the iteration decrements the level of v from k to $k-1$, Lemma 2.7 implies that the iteration takes $\Theta(1 + D_v(0,k))$ time. Applying Claim 2.16, we now infer that $T \leq \beta^k$. ∎

Lemma 2.18. *For every node $u \in \mathcal{N}_v$, we have $\delta\Psi(u) \geq 0$.*

Proof. Fix any node $u \in \mathcal{N}_v$. As the level of the node v decreases from k to $k-1$, we infer that $w^0(u,v) \leq w^1(u,v)$, and, accordingly, we get $W_u^0 \leq W_u^1$. Since $\Psi(u) = \epsilon \cdot (L - \ell(u)) + \beta^{\ell(u)} \cdot \max(0, \alpha - W_u)$, we derive that $\Psi^0(u) \geq \Psi^1(u)$. Thus, we have $\delta\Psi(u) = \Psi^0(u) - \Psi^1(u) \geq 0$. ∎

Lemma 2.19. *For every node $u \in \mathcal{N}_v$, we have*

$$\delta\Phi(u,v) = \begin{cases} 0 & \text{if } u \in \mathcal{N}_v(k,L); \\ -(1+\epsilon) & \text{if } u \in \mathcal{N}_v(0,k-1). \end{cases}$$

Proof. Fix any node $u \in \mathcal{N}_v$. We prove the lemma by considering two possible scenarios:

1. We have $u \in \mathcal{N}_v(k,L)$. As the iteration decreases the level of the node v from k to $k-1$, we infer that $\Phi^0(u,v) = \Phi^1(u,v) = (1+\epsilon) \cdot (L - \ell(u))$. Hence, we get $\delta\Phi(u,v) = \Phi^1(u,v) - \Phi^0(u,v) = 0$.
2. We have $u \in \mathcal{N}_v(0,k-1)$. Since the iteration decreases the level of node v from k to $k-1$, we infer that $\Phi^0(u,v) = (1+\epsilon) \cdot (L-k)$ and $\Phi^1(u,v)=(1+\epsilon) \cdot (L-k+1)$. Hence, we get $\delta\Phi(u,v)=\Phi^1(u,v)-\Phi^0(u,v)=-(1+\epsilon)$. ∎

We partition W_v^0 into two parts: x and y. The first part denotes the contributions towards W_v^0 by the neighbors of v that lie below level k, while the second part denotes the contribution towards W_v^0 by the neighbors of v that lie on or above level k. Thus, we get the following equations:

$$W_v^0 = x + y \leq 1 \tag{18}$$

$$x = \sum_{u \in \mathcal{N}_v(0,k-1)} w^0(u,v) = \beta^{-k} \cdot D_v(0,k-1) \tag{19}$$

$$y = \sum_{u \in \mathcal{N}_v(k,L)} w^0(u,v). \tag{20}$$

Lemma 2.20. *We have $\sum_{u \in \mathcal{N}_v} \delta\Phi(u,v) = -(1+\epsilon) \cdot x \cdot \beta^k$.*

Proof. Lemma 2.19 implies that $\sum_{u \in \mathcal{N}_v} \delta\Phi(u,v) = -(1+\epsilon) \cdot D_v(0,k-1)$. Applying (19), we infer that $D_v(0,k-1) = x \cdot \beta^k$. The lemma follows. ∎

Lemma 2.21. *We have*

$$\delta\Psi(v) = -\epsilon + (\alpha - x - y) \cdot \left(\beta^{k+1}/(\beta-1)\right) - \max\left(0, \alpha - \beta x - y\right) \cdot \left(\beta^k/(\beta-1)\right).$$

Proof. By (18), we have $W_v^0 = x + y < 1$. Since $\ell^0(v) = k$, we get

$$\Psi^0(v) = \epsilon(L-k) + (\alpha - x - y)\cdot\left(\beta^{k+1}/(\beta-1)\right). \tag{21}$$

As the node v decreases its level from k to $k-1$, we have

$$w^1(u,v) = \begin{cases} \beta\cdot w^0(u,v) & \text{if } u \in \mathcal{N}_v(0,k-1);\\ w^0(u,v) & \text{if } u\in\mathcal{N}_v(k,L).\end{cases}$$

Accordingly, we have $W_v^1 = \beta\cdot x + y$, which implies the following equation:

$$\Psi^1(v) = \epsilon(L-k+1) + \max(0, \alpha-\beta x - y)\cdot\left(\beta^k/(\beta-1)\right). \tag{22}$$

Since $\delta\Psi(v) = \Psi^0(v) - \Psi^1(v)$, the lemma now follows from (21) and (22). ∎

Proof of Theorem 2.11 *(for Case* 2*).* We consider two possible scenarios depending upon the value of $(\alpha-\beta x-y)$. We show that in each case, $\delta\mathcal{B}\geq\beta^k$. The theorem (for Case 2) then follows from Lemma 2.17.

1. Suppose that $(\alpha-\beta x-y)<0$. From Lemmas 2.18, 2.20, and 2.21 and from (13), we derive

$$\begin{aligned}
\epsilon\cdot\delta\mathcal{B} &= \sum_{u\in\mathcal{N}_v}\delta\Psi(u) + \sum_{u\in\mathcal{N}_v}\delta\Phi(u,v) + \delta\Psi(v),\\
&\geq 0 - (1+\epsilon)\cdot x\cdot\beta^k - \epsilon + (\alpha-x-y)\cdot\beta^{k+1}/(\beta-1),\\
&\geq -\epsilon - (1+\epsilon)\cdot\beta^k + (\alpha-1)\cdot\beta^{k+1}/(\beta-1), && (18)\\
&= -\epsilon + \frac{\beta^k}{(\beta-1)}\cdot\left\{(\alpha-1)\cdot\beta - (1+\epsilon)(\beta-1)\right\},\\
&= -\epsilon + 2\cdot(1+\epsilon)\cdot\beta^k, && (\text{since } \alpha = 1+3\epsilon \text{ and } \beta = 1+\epsilon)\\
&\geq \epsilon\cdot\beta^k. && (\text{since } \beta>1, \epsilon<1)
\end{aligned}$$

2. Suppose that $(\alpha-\beta x-y)\geq 0$. From Lemmas 2.18, 2.20, and 2.21 and from (13), we derive

$$\begin{aligned}
\epsilon\cdot\delta\mathcal{B} &= \sum_{u\in\mathcal{N}_v}\delta\Psi(u) + \sum_{u\in\mathcal{N}_v}\delta\Phi(u,v) + \delta\Psi(v),\\
&\geq 0 - (1+\epsilon)\cdot x\cdot\beta^k - \epsilon + (\alpha-x-y)\cdot\beta^{k+1}/(\beta-1) - (\alpha-\beta x-y)\cdot\beta^k/(\beta-1),\\
&= -\epsilon + \frac{\beta^k}{(\beta-1)}\cdot\left\{(\alpha-x-y)\cdot\beta - (1+\epsilon)\cdot x\cdot(\beta-1) - (\alpha-\beta x-y)\right\},\\
&= -\epsilon + \frac{\beta^k}{(\beta-1)}\cdot\left\{(\alpha-x-y)\cdot(\beta-1) - \epsilon\cdot x\cdot(\beta-1)\right\},\\
&\geq -\epsilon + \frac{\beta^k}{(\beta-1)}\cdot\left\{(\alpha-1)\cdot(\beta-1) - \epsilon\cdot(\beta-1)\right\}, && (\text{since } 0\leq x+y\leq 1)\\
&= -\epsilon + 2\cdot\epsilon\cdot\beta^k, && (\text{since } \alpha=1+3\epsilon \text{ and } \beta=1+\epsilon)\\
&\geq \epsilon\cdot\beta^k. && (\text{since } \beta>1)
\end{aligned}$$
∎

3. Dynamic matching.

3. Dynamic matching. In this section, we present algorithms for maintaining an (approximately) maximum matching in a dynamic graph $G=(V,E)$ with $|V|=n$ nodes and $|E|=m$ edges. Note that the value of m changes with time as edges keep getting inserted into or deleted from the graph G. But the value of n always remains the same, as we do not allow insertion or deletion of nodes. Our main results are stated in the theorems below.

THEOREM 3.1. *We can maintain a $(3+\epsilon)$-approximate maximum matching in a dynamic graph $G = (V, E)$ in $O\left(\min\left(\sqrt{n}/\epsilon, m^{1/3}/\epsilon^2\right)\right)$ amortized update time.*

THEOREM 3.2. *We can maintain a $(4+\epsilon)$-approximate maximum matching in a dynamic graph $G = (V, E)$ in $O(m^{1/3}/\epsilon^2)$* worst-case *update time.*

To explain the main idea behind our approach, we first describe a naive algorithm for maintaining a *maximal* matching.[2] Let $M \subseteq E$ be the maximal matching maintained by our naive algorithm. When an edge (u, v) is inserted into the input graph G, we add the edge (u, v) to the matching M iff both the endpoints $x \in \{u, v\}$ were unmatched in M just before the insertion. When an unmatched edge $(u, v) \in E \setminus M$ gets deleted from G, we do not change the matching M. Finally, when a matched edge $(u, v) \in M$ gets deleted from the graph G, we consider each of its endpoints $x \in \{u, v\}$ one after the other. While considering the node x, we scan all the neighbors of x in the input graph G, searching for an unmatched node. If we are successful in finding such a neighbor y of x that is unmatched, then we add the edge (x, y) to the matching M. Otherwise, it means that every neighbor of x is matched in M, and in this case, we let the node x remain unmatched. Clearly, this algorithm ensures that the matching $M \subseteq E$ is maximal. It takes $O(1)$ time in the worst case to handle an edge insertion in the input graph. In contrast, the time taken to handle the deletion of an edge (u, v) from the input graph is at most the sum of the degrees of the two endpoints u and v. Since a maximal matching gives 2-approximation to maximum matching, we get the following theorem.

THEOREM 3.3 (folklore). *Consider a dynamic graph $G = (V, E)$, where the maximum degree of any node is bounded by $d \geq 1$ at every point in time. Then there exists an algorithm for maintaining a 2-approximate maximum matching in G that handles every edge insertion in $O(1)$ worst-case time and every edge deletion in G in $O(d)$ worst-case time.*

Neiman and Solomon [20] extended the above idea to show how to maintain a 3/2-approximate maximum matching in a bounded degree graph. The following theorem is a direct corollary of their work.

THEOREM 3.4 ([20]). *Consider a dynamic graph $G = (V, E)$, where the maximum degree of any node is bounded by $d \geq 1$ at every point in time. Then there exists an algorithm for maintaining a 3/2-approximate maximum matching in G that handles every edge insertion in $O(d)$ worst-case time and every edge deletion in G also in $O(d)$ worst-case time.*

One crucial difference between Theorems 3.3 and 3.4 is that the former can handle an edge insertion in $O(1)$ worst-case time but the latter cannot. This difference will turn out to be significant later on when we focus on bounding the *worst-case* update time of our dynamic matching algorithm. The main message we take away from the above two theorems is that maintaining an approximately maximum matching is easy in bounded degree graphs. Thus, it seems natural to pursue the following approach for general graphs:

[2]A maximal matching $M \subseteq E$ has the following property: Every edge $(u, v) \in E$ in the input graph has at least one endpoint matched in M. It is well known that the size of any maximal matching gives 2-approximation to the size of the maximum matching.

- (Step 1) Maintain a bounded degree subgraph $\kappa(G) = (V, \kappa(E))$, $\kappa(E) \subseteq E$, of the input graph $G = (V, E)$. We call the subgraph $\kappa(G)$ a *kernel* of G.
- (Step 2) Show that the kernel $\kappa(G)$ approximately preserves the size of the maximum matching in G.
- (Step 3) Maintain an approximately maximum matching in $\kappa(G)$ using Theorem 3.3 or 3.4.

We define the notion of a kernel in section 3.1. In Theorem 3.12, we show that a kernel indeed preserves the size of the maximum matching within a constant factor. The proof of Theorem 3.12 appears in section 3.2. In section 3.3, we present dynamic algorithms for maintaing a kernel. We maintain an approximately maximum matching on top of this kernel using Theorem 3.3 or 3.4. The first of our main results—Theorems 3.1—follows from Corollaries 3.21 and 3.23. Finally, in section 3.4, we present a dynamic matching algorithm with efficient worst-case update time, which proves Theorem 3.2.

3.1. The kernel and its properties. We now introduce the notion of a *kernel* in the input graph. Intuitively, a kernel is a bounded degree subgraph that preserves the size of the maximum matching within a constant factor. To appreciate the precise definition of a kernel, we first need to revisit the notion of a maximal c-matching. Consider any integer $c \geq 0$. A maximal c-matching in the input graph $G = (V, E)$ is a *maximal* subset of edges $E_c \subseteq E$ such that every node $v \in V$ has at most c incident edges in E_c. In other words, a subset of edges $E_c \subseteq E$ is a maximal c-matching iff (1) every node $x \in V$ has at most c incident edges in E_c and (2) every edge $(u, v) \in E \setminus E_c$ has at least one endpoint $x \in \{u, v\}$ such that x has *exactly* c neighbors in E_c. A maximal c-matching can be constructed using a simple linear time greedy algorithm as follows:

- Start by setting $E_c \leftarrow \emptyset$. Now, scan through the edges in E in any arbitrary order. While considering an edge $(u, v) \in E$ during this scan, insert the edge (u, v) into the set E_c iff at the present moment both its endpoints u, v have *strictly* less than c incident edges in E_c.

Consider any maximal c-matching $E_c \subseteq E$. Let $T_c \subseteq V$ denote the subset of nodes with degree *exactly* equal to c in the subgraph $G_c = (V, E_c)$. Let $S_c = V \setminus T_c$ denote the remaining subset of nodes with degree strictly less than c in G_c. We say that the nodes in T_c and S_c are *tight* and *slack*, respectively. Note that any maximal c-matching satisfies the three properties stated below.

PROPERTY 3.5. *Every node has degree at most c in the subgraph G_c.*

PROPERTY 3.6. *Every tight node has degree at least c in the subgraph G_c.*

PROPERTY 3.7. *Every edge in the input graph connecting two slack nodes is included in the subgraph G_c.*

Properties 3.5 and 3.6 follow from the definition of a maximal c-matching and the definition of tight and slack nodes. Property 3.7 holds since the subset $E_c \subseteq E$ is *maximal*, meaning that no more edge can be added to E_c without violating the condition that every node has degree at most c in $G_c = (V, E_c)$. We are now ready to define a kernel, which will be a *relaxation* of the concept of a maximal c-matching.

3.1.1. Defining a kernel. We start by introducing some notations that will be used throughout the rest of section 3. In the input graph $G = (V, E)$, we let $\mathcal{N}_v = \{u \in V : (u, v) \in E\}$ denote the set of *neighbors* of $v \in V$. A *kernel* will be a subgraph of G, and it will be denoted as $\kappa(G) = (V, \kappa(E))$ with $\kappa(E) \subseteq E$. For all

$v \in V$, we define the set $\kappa(\mathcal{N}_v) = \{u \in \mathcal{N}_v : (u, v) \in \kappa(E)\}$, which consists of all the neighbors of v in the kernel $\kappa(G)$. The precise definition of a kernel is stated below.

Definition 3.8. *Fix any integer $c \geq 1$ and any $\epsilon \in (0, 1/3)$. Consider any partition of the node set V into two subsets: $T \subseteq V$ and $S = V \setminus T$. Say that the nodes in T and S are* tight *and* slack*, respectively. The subgraph $\kappa(G)$ is an (ϵ, c)-kernel of G with respect to the partition (T, S) iff it satisfies Invariants 3.9–3.11.*

Invariant 3.9. *$|\kappa(\mathcal{N}_v)| \leq (1+\epsilon)c$ for all $v \in V$; i.e., a node has at most $(1+\epsilon)c$ neighbors in $\kappa(G)$.*

Invariant 3.10. *$|\kappa(\mathcal{N}_v)| \geq (1-\epsilon)c$ for all $v \in T$; i.e., a tight node has at least $(1-\epsilon)c$ neighbors in $\kappa(G)$.*

Invariant 3.11. *For all $u, v \in S$, if $(u, v) \in E$, then $(u, v) \in \kappa(E)$. In other words, if two slack nodes are connected by an edge in G, then that edge must belong to $\kappa(G)$.*

Note that Invariants 3.9, 3.10, and 3.11 are analogous to Properties 3.5, 3.6, and 3.7, respectively. Specifically, a $(0, c)$-kernel is nothing but a maximal c-matching. From Invariant 3.9, it is immediately obvious that an (ϵ, c)-kernel is a subgraph with bounded degree. The following theorem implies that such a subgraph also preserves the size of the maximum matching within a constant factor. The proof of this theorem is derived from a subsequent paper [6]. For the sake of completeness, we present the detailed proof in section 3.2.

Theorem 3.12 ([6]). *Consider any matching $M^* \subseteq E$ in the input graph $G = (V, E)$. Then there exists a matching $M \subseteq \kappa(E)$ in any (ϵ, c)-kernel $\kappa(G) = (V, \kappa(E))$ such that $|M^*| \leq (2 + 42\epsilon) \cdot |M|$.*

3.2. Proof of Theorem 3.12.

The proof uses the notion of a *fractional matching*. Specifically, a fractional matching assigns a nonnegative (possibly fractional) weight $w(e) \in [0, 1]$ to every edge $e \in E$, subject to the constraint that $\sum_{(u,v) \in E} w(u, v) \leq 1$ for all $v \in V$. In other words, in a fractional matching, the total weight received by every node from all its incident edges is at most one. The *size* of a fractional matching is defined as the sum of the weights of all the edges in the input graph. The proof consists of three parts as described below:

1. We carefully construct a fractional matching $\{w(e)\}$ in the input graph $G = (V, E)$. This construction depends on the matching $M^* \subseteq E$ and the kernel $\kappa(G)$.
2. We show that the size of M^* is at most $(2+20\epsilon)$ times the size of this fractional matching $\{w(e)\}$:

$$|M^*| \leq (2 + 20\epsilon) \cdot \sum_{e \in E} w(e). \tag{23}$$

3. We construct a matching $M \subseteq \kappa(E)$ in the kernel and show that the size of the fractional matching $\{w(e)\}$ is at most $(1 + \epsilon)$ times the size of M:

$$\sum_{e \in E} w(e) \leq (1 + \epsilon) \cdot |M|. \tag{24}$$

By (23) and (24), we have $|M^*| \leq (2 + 20\epsilon)(1 + \epsilon) \cdot |M| \leq (2 + 42\epsilon) \cdot |M|$. This concludes the proof of Theorem 3.12. We describe these three parts of the proof in sections 3.2.2, 3.2.3, and 3.2.4, respectively. But first we need to define a few notations that will be used throughout the rest of the proof.

3.2.1. Notations. We define the parameter $\lambda = \lceil (1+\epsilon)c \rceil$. Let $E_T = \{(u,v) \in \kappa(E) : \{u,v\} \cap T \neq \emptyset\}$ denote the set of all kernel edges with at least one tight endpoint. Let $M^*_S = \{(u,v) \in M^* : u, v \in S\}$ denote the set of edges in the matching M^* whose both endpoints are slack. Let $M^*_T = \{(u,v) \in M^* : \{u,v\} \cap T \neq \emptyset\}$ denote the set of edges in the matching M^* with at least one tight endpoint. Since the subsets $T \subseteq V$ and $S = V \setminus T$ partition the node set V and since no two edges in the matching M^* share a common endpoint, we infer that the subsets M^*_T and M^*_S also partition the set of edges in M^*. Finally, let $\kappa_T(\mathcal{N}_v) = \{u \in \mathcal{N}_v \cap T : (u,v) \in \kappa(E)\}$ denote the set of tight neighbors of any node $v \in V$ in the kernel.

3.2.2. Part 1: Constructing the fractional matching. We construct the fractional matching in two phases. Initially, before the first phase, every edge has weight zero. During the first phase, we increase the weight of every edge in E_T from 0 to $1/\lambda$. Thus, at the end of the first phase, for all nodes $v \in V$, we have $W_v = (1/\lambda) \cdot \kappa_T(\mathcal{N}_v) \leq (1/\lambda) \cdot \kappa(\mathcal{N}_v) \leq 1$. The last inequality follows from Invariant 3.9. In other words, at the end of the first phase, the total weight received by any node from its incident edges is at most one. We now classify the edges in M^*_S into two types, as stated below:

- Type (a): Edges $(u,v) \in M^*_S$, where $W_u + W_v \geq 1$ at the end of the first phase. Such edges have $\kappa_T(\mathcal{N}_u) + \kappa_T(\mathcal{N}_v) \geq \lambda$.
- Type (b): Edges $(u,v) \in M^*_S$, where $W_u + W_v < 1$ at the end of the first phase. Such edges have $\kappa_T(\mathcal{N}_u) + \kappa_T(\mathcal{N}_v) < \lambda$.

In the second phase, we increase the weight of every type (b) edge $(u,v) \in M^*_S$ from zero until the point where $W_u + W_v$ becomes equal to one. Specifically, at the end of the second phase, every type (b) edge $(u,v) \in M^*_S$ has a weight $w(u,v) = 1 - (1/\lambda) \cdot (\kappa_T(\mathcal{N}_u) + \kappa_T(\mathcal{N}_v))$, every edge $(u,v) \in E_T$ has a weight $w(u,v) = 1/\lambda$, and every other edge has zero weight. This completes the construction of the fractional matching. The weight assigned to every edge in the input graph is summarized as

$$w(u,v) = \begin{cases} 1/\lambda & \text{if } (u,v) \in E_T; \\ \max\left(0, 1 - \frac{\kappa_T(\mathcal{N}_u) + \kappa_T(\mathcal{N}_v)}{\lambda}\right) & \text{if } (u,v) \in M^*_S; \\ 0 & \text{otherwise.} \end{cases} \tag{25}$$

From the description above, it is easy to check that $W_v \in [0,1]$ for all nodes $v \in V$ at the end of our construction. Accordingly, the edge weights $\{w(e)\}$ define a feasible fractional matching in the input graph.

3.2.3. Part 2: Lower bounding the size of the fractional matching. We will now show that the size of the matching $M^* \subseteq E$ is at most $(2+\epsilon)$ times the size of the fractional matching $\{w(e)\}$ constructed in part (1). Specifically, we will show that (23) holds.

Let $V(M^*) \subseteq V$ be the set of endpoints of the matched edges in M^*. Since each matched edge has two endpoints, we infer that $|V(M^*)| = 2 \cdot |M^*|$. Similarly, since each edge $(u,v) \in E$ contributes $w(u,v)$ towards the weights of both of its two endpoints, we infer that $\sum_{v \in V} W_v = 2 \cdot \sum_{e \in E} w(e)$. Thus, (23) is equivalent to the following guarantee:

$$|V(M^*)| \leq (2+20\epsilon) \cdot \sum_{v \in V} W_v. \tag{26}$$

We will show that (26) holds by proving that

$$W_u + W_v \geq 1 - 2\epsilon \text{ for every edge } (u,v) \in M^*. \tag{27}$$

If we sum (27) over all the edges $(u,v) \in M^*$, then we get

$$\sum_{x \in V(M^*)} W_x \geq (1-2\epsilon) \cdot |M^*| = (1-2\epsilon) \cdot |V(M^*)|/2.$$

Rearranging the terms in the above inequality, we get

$$|V(M^*)| \leq \frac{2}{(1-2\epsilon)} \cdot \sum_{x \in V(M^*)} W_x \leq (2+20\epsilon) \cdot \sum_{v \in V} W_x.$$

The last inequality holds since $\epsilon \in (0, 1/3)$ according to Definition 3.8. In other words, (27) implies (26), which, in turn, is equivalent to (23). It now remains to prove (27).

Recall the notations defined in section 3.2.1. In that section, we also argued that the set of edges in the matching M^* is partitioned into two subsets $M^*_S \subseteq M^*$ and $M^*_T = M^* \setminus M^*_S$. To complete the proof of (27), we will show the following guarantee:

$$W_u + W_v \geq \begin{cases} 1-2\epsilon & \text{for every edge } (u,v) \in M^*_T; \\ 1 & \text{for every edge } (u,v) \in M^*_S. \end{cases} \tag{28}$$

Consider any edge $(u,v) \in M^*_T$, and without any loss of generality, suppose that $v \in T$. Invariant 3.10 ensures that the node v has at least $(1-\epsilon)c = (1-\epsilon)\lambda/(1+\epsilon) \geq (1-2\epsilon)\lambda$ many neighbors in the kernel. Since $v \in T$, each of these edges $(v,x) \in \kappa(E)$ belongs to the set E_T. Note that (25) implies that each of these edges gets a weight of $1/\lambda$. Hence, we infer that $W_v \geq (1-2\epsilon)\lambda \cdot (1/\lambda) \geq 1-2\epsilon$. Thus, we get $W_u + W_v \geq W_v \geq 1-2\epsilon$. In other words, (27) holds for every edge in M^*_T.

Next, consider any edge $(u,v) \in M^*_S$. We want to lower bound the sum of the node weights $W_u + W_v$. The edges that contribute nonzero weights to this sum can be classified into two types, as described below:

- (1) The edges in the kernel that connect u or v to some tight node. There are exactly $\kappa_T(\mathcal{N}_u) + \kappa_T(\mathcal{N}_v)$ many such edges, and each of them contributes $1/\lambda$ towards the sum $W_u + W_v$ (follows from (25)). Thus, their total contribution towards the sum $W_u + W_v$ is exactly equal to $(1/\lambda) \cdot (\kappa_T(\mathcal{N}_u) + \kappa_T(\mathcal{N}_v))$.
- (2) The edge (u,v). Note that (25) implies that $w(u,v) \geq 1 - (1/\lambda) \cdot (\kappa_T(\mathcal{N}_u) + \kappa_T(\mathcal{N}_v))$. Further, note that the edge (u,v) contributes $2 \cdot w(u,v)$ towards the sum $W_u + W_v$.

Summing over the contributions from these two types of edges, it follows that $W_u + W_v \geq 1$. In other words, (27) holds for every edge in M^*_S as well.

3.2.4. Part 3: Constructing the matching M in the kernel.

We will use the notations from section 3.2.1, and we will prove (24). We first construct a multigraph $\mathcal{G} = (V, \mathcal{E})$ defined on the node set of the input graph $G = (V,E)$ as follows. For every edge $(u,v) \in E_T$, we create a multiedge (u,v) with a single copy and add it to $\mathcal{E}$. Finally, for every edge $(u,v) \in M^*_S$, we create a multiedge (u,v) with $\max(0, \lambda - \kappa_T(\mathcal{N}_u) - \kappa_T(\mathcal{N}_v))$ many copies and add all these copies to $\mathcal{E}$. This concludes the construction of the multigraph $\mathcal{G} = (V, \mathcal{E})$. From the description in section 3.2.2 and (25), it follows that the total weight assigned to all the edges in the input graph is exactly equal to $|\mathcal{E}|/\lambda$. More formally, we have

$$\sum_{e \in E} w(e) = |\mathcal{E}|/\lambda. \tag{29}$$

We will show that the multigraph $\mathcal{G} = (V, \mathcal{E})$ admits a *proper edge coloring* with $\lambda + 1$ colors. Specifically, given a palette of $\lambda + 1$ colors, we will show how to assign a color to every multiedge (counting all the copies) such that no two multiedges with the same color share a common endpoint. The key observation is that the set of multiedges receiving the same color form a matching, and, in particular, different copies of multiedges between the same two nodes get different colors. Hence, a simple counting argument implies that in a proper $\lambda + 1$ coloring of $\mathcal{G}$, there must exist a color that is assigned to at least $|\mathcal{E}|/(\lambda + 1)$ many multiedges. This means that there exists a matching M in $\mathcal{G}$ of size at least $|\mathcal{E}|/(\lambda + 1)$. Clearly, the set of edges M also form a matching in the kernel $\kappa_T(G) = (V, \kappa(E))$: This is true because (1) the multiedges in $\mathcal{G}$ originate from the edge set $E_T \cup M_S^*$ in the input graph $G = (V, E)$, (2) $E_T \subseteq \kappa(E)$ by definition, and (3) $M_S^* \subseteq \kappa(E)$ due to Invariant 3.11. Thus, there exists a matching $M \subseteq \kappa(E)$ in the kernel of size $|M| \geq |\mathcal{E}|/(\lambda+1)$. Now, (29) implies that $\sum_{e \in E} w(e) = |\mathcal{E}|/\lambda \leq ((\lambda+1)/\lambda) \cdot |M| = (1 + 1/\lambda) \cdot |M| \leq (1+\epsilon) \cdot |M|$. This concludes the proof of (24).

It now remains to show that $\mathcal{G}$ admits a proper edge coloring with $\lambda + 1$ colors. Let $G_T = (V, E_T)$ denote the subgraph of the input graph $G = (V, E)$ consisting of all the kernel edges with at least one tight endpoint. Since $E_T \subseteq \kappa(E)$, Invariant 3.9 implies that the degree of every node in G_T is at most λ. Hence, by Vizing's theorem, there exists a proper edge coloring of G_T using only $\lambda + 1$ colors. Note that in the multigraph $\mathcal{G}$, each edge $(u, v) \in E_T$ is present in only one copy. Accordingly, we color the multiedges originating from E_T exactly the same way as done by the proper $\lambda + 1$ coloring of G_T. It now remains to color the multiedges originating from M_S^*. Towards this end, consider any edge $(u, v) \in M_S^*$. If $\kappa_T(\mathcal{N}_u) + \kappa_T(\mathcal{N}_v) \geq \lambda$, then the edge (u, v) contributes *zero* copy of multiedges to $\mathcal{G}$, and hence we do not need to assign any color for such an edge in $\mathcal{G}$. Else, if $\kappa_T(\mathcal{N}_u) + \kappa_T(\mathcal{N}_v) < \lambda$, then the edge (u, v) contributes $\lambda - \kappa_T(\mathcal{N}_u) - \kappa_T(\mathcal{N}_v) = \eta$ (say) copies of multiedges to $\mathcal{G}$. However, there are at most $\kappa_T(\mathcal{N}_u) + \kappa_T(\mathcal{N}_v) = \lambda - \eta$ many multiedges in $\mathcal{G}$ that originate from E_T and have an endpoint in $\{u, v\}$. Since the palette consists of $\lambda + 1$ colors, it means that there are at least $(\lambda + 1) - (\lambda - \eta) = \eta + 1$ colors that are not assigned to any multiedge originating from E_T with an endpoint in $\{u, v\}$. Accordingly, we can properly color the η copies of the multiedge (u, v) with these $\eta + 1$ remaining colors, ensuring that there is no conflict. This shows that there exists a proper $\lambda + 1$ edge coloring in the multigraph $\mathcal{G}$.

3.3. Dynamic algorithms for maintaining a kernel.

Our focus in this section is to present algorithms for maintaining an (ϵ, c)-kernel in a dynamic graph. Towards this end, in section 3.3.1, we first present a generic template for maintaining an (ϵ, c)-kernel and explain some of its important properties. Subsequently, using this template, we present two concrete algorithms in section 3.3.2 for maintaining an (ϵ, c)-kernel and analyze their amortized update times. As corollaries, we obtain algorithms for maintaining an approximately maximum matching in a dynamic graph.

3.3.1. An algorithmic template.

Suppose that just before the insertion/deletion of an edge in G, we have a kernel $\kappa(G) = (V, \kappa(E))$ and a partition of the node set V into subsets $T \subseteq V$ and $S = V \setminus T$ that satisfy Invariants 3.9–3.11. We describe how to update the kernel $\kappa(G)$ and the partition (T, S) following the edge insertion/deletion in G in such a way which ensures that Invariants 3.9–3.11 continue to remain valid.

Insertion of an edge (u, v) in $G = (V, E)$. Suppose that an edge (u, v) is inserted into the input graph. For every endpoint $x \in \{u, v\}$, if $|\mathcal{N}_x| \geq c$, then we ensure that the node x is tight by setting $T \leftarrow T \cup \{x\}$ and $S \leftarrow S \setminus \{x\}$. Next, if both $|\kappa(\mathcal{N}_u)| < c$ and $|\kappa(\mathcal{N}_v)| < c$, then we perform the following operations:

- We insert the edge (u, v) into the kernel. Next, for all $x \in \{u, v\}$, if $|\kappa(\mathcal{N}_x)| = c$, then we set $T \leftarrow T \cup \{x\}$ and $S \leftarrow S \setminus \{x\}$. In other words, if the number of neighbors of any endpoint $x \in \{u, v\}$ happens to become equal to c due to the insertion of the edge (u, v) into the kernel, then we ensure that the endpoint x also becomes tight.

LEMMA 3.13. *If Invariants* 3.9–3.11 *were valid just before the insertion of the edge (u, v) in G, then they continue to remain valid just after the execution of the above procedure following the edge insertion.*

Proof. We first focus on Invariant 3.9. The edge (u, v) gets inserted into the kernel only if each of its endpoints had less than c neighbors in the kernel just before the insertion of the edge in G. Hence, Invariant 3.9 continues to remain valid.

Next, we focus on Invariant 3.10. A node x changes from being slack to being tight due to the above procedure *only if* $|\mathcal{N}_x| = c$. Furthermore, the number of neighbors a node has in the kernel does not decrease due to the above procedure. Hence, Invariant 3.10 continues to remain valid.

Finally, we focus on Invariant 3.11. The above procedure never changes a node from being tight to being slack. Thus, the only way Invariant 3.11 can get violated is if an edge (u, v) is inserted between two slack nodes u and v in the input graph G. In this event, if $|\mathcal{N}_x| \geq c$ for any $x \in \{u, v\}$ just before the insertion of the edge (u, v) in G, then the procedure ensures that the node x changes from being slack to being tight, which implies that Invariant 3.11 does not apply to the edge (u, v) anymore. Else, if $|\mathcal{N}_x| < c$ for all $x \in \{u, v\}$ just before the insertion of the edge (u, v) in G, then the procedure adds the edge (u, v) to the kernel, which again implies that Invariant 3.11 continues to remain valid for that edge. We therefore conclude that Invariant 3.11 always continues to remain valid at the end of the above procedure. ◻

LEMMA 3.14. *The above procedure for handling an edge insertion in G takes $O(1)$ worst-case time.*

Proof. Immediately follows from the description of the procedure. ◻

Deletion of an edge (u, v) in $G = (V, E)$. Suppose that an edge (u, v) is deleted from the input graph. If the edge (u, v) was not part of the kernel just before getting deleted from G, then we have nothing more to do. The interesting case is when the edge (u, v) belongs to the kernel, and in this case, we first delete (u, v) from the kernel after it gets deleted from G. Next, we perform the following operations for every $x \in \{u, v\} \cap T$ with $|\kappa(\mathcal{N}_x)| < (1 - \epsilon)c$, that is, for every tight endpoint of the edge (u, v) that is left with less than $(1 - \epsilon)c$ neighbors in the kernel due to the deletion of (u, v):

- Make the node x slack by setting $T \leftarrow T \setminus \{x\}$ and $S \leftarrow S \cup \{x\}$.
- Scan through the edges incident on x in the input graph G. While considering any such edge $(x, y) \in E$ during this scan, perform the following operations:
  - If $(x, y) \notin \kappa(E)$ and $|\kappa(\mathcal{N}_x)| < c$ and $|\kappa(\mathcal{N}_y)| < (1 + \epsilon)c$, then
    * insert the edge (x, y) into the kernel. After this insertion, if $|\kappa(\mathcal{N}_y)| \geq c$, then ensure that the node y becomes tight by setting $T \leftarrow T \cup \{y\}$ and $S \leftarrow S \setminus \{y\}$. Finally, after this insertion, if $|\kappa(\mathcal{N}_x)|$ becomes

equal to c, then ensure that x becomes tight again by setting $T \leftarrow T \cup \{x\}$ and $S \leftarrow S \setminus \{x\}$ and then *abort the scan.*

LEMMA 3.15. *If Invariants* 3.9–3.11 *were valid just before the deletion of the edge (u, v) from G, then they continue to remain valid just after the execution of the above procedure following the edge deletion.*

Proof. We first focus on Invariant 3.9. This invariant can get violated only when edges get inserted into the kernel. The only time an edge gets inserted into the kernel due to the above procedure is when we are scanning the edges incident on an endpoint $x \in \{u, v\}$. In that event, however, we insert an edge (x, y) into the kernel only if $|\kappa(\mathcal{N}_x)| < c$ and $|\kappa(\mathcal{N}_y)| < (1 + \epsilon)c$. Hence, the above procedure can never result in a node having more than $(1 + \epsilon)c$ neighbors in the kernel. So Invariant 3.9 continues to remain valid.

We next focus on Invariant 3.10. The above procedure has the following property. If the edge (u, v) belonged to the kernel just before getting deleted from G and if this deletion decreases the value of $|\mathcal{N}_x|$ for any endpoint $x \in \{u, v\}$ below the threshold $(1 - \epsilon)c$, then we immediately ensure that the node x becomes slack. Subsequently, while scanning the neighbors of x in G, we never delete edges from the kernel, and we change a node from being slack to tight only when its degree in the kernel becomes greater than or equal to c. This implies that Invariant 3.10 continues to remain valid.

We finally focus on Invariant 3.11. Due to the above procedure, the only edge that gets deleted from the kernel is the edge (u, v), and the only nodes that can change from being tight to being slack are the endpoints u, v. Thus, Invariant 3.11 can get violated only for an edge incident on $x \in \{u, v\}$, and that too can happen only when the node x changes from being tight to slack due to the above procedure. Now, note that if any endpoint $x \in \{u, v\}$ becomes slack, then the above procedure starts scanning its neighboring edges in the input graph and tries to include as many of these edges in the kernel as possible: This process stops only when either (a) the node x changes again from being slack to being tight or (b) we finish scanning all the edges incident on x in the input graph. In case (a), we clearly need not worry about Invariant 3.11 being violated for any edge incident on the node x. Thus, it remains to consider case (b). In this case, note that every edge $(x, y) \in E$ incident on x either belongs to the kernel or has $|\kappa(\mathcal{N}_y)| = (1 + \epsilon)c$. However, the above procedures for handling the insertion and deletion of an edge in G together ensure that the node y is tight whenever $|\kappa(\mathcal{N}_y)|$ is equal to $(1 + \epsilon)c$. To summarize, in case (b), there is no edge $(x, y) \in E \setminus \kappa(E)$ whose other endpoint y is slack. Hence, the edges incident on x do not violate Invariant 3.11 in case (b). ∎

LEMMA 3.16. *The above procedure for handling an edge deletion in G takes time proportional to the total number of neighbors (in G) of the endpoints $x \in \{u, v\}$ that are scanned.*

Proof. Immediately follows from the description of the procedure. ∎

We now describe three important properties that are satisfied by the above procedures for handling the insertion/deletion of an edge in G. First, note that these procedures pay special attention to a node whenever the number of its neighbors in the kernel increases by one. Specifically, whenever the number of neighbors of a node in the kernel increases by one, the procedures check if the node has at least c neighbors in the kernel. If the answer to this question is in the affirmative and if the node was slack just prior to this event, then the node becomes tight (by moving from the set S to the set T). This implies the following property.

Property 3.17. *When a node changes from being slack to being tight, it has degree at least c in the kernel.*

Next, note that a node never changes from being tight to being slack due to the insertion of an edge in the input graph G. Such an event can take place only if (1) an edge (u, v) gets deleted from G, (2) the edge was also part of the kernel just before its deletion, and (3) this deletion results in the degree in the kernel of some endpoint $x \in \{u, v\}$ dropping below the threshold $(1-\epsilon)c$. If the endpoint x was tight just before the deletion, then we make it slack at this time. This implies the following property.

Property 3.18. *When a node changes from being tight to being slack, it has degree at most* $(1 - \epsilon)c$ *in the kernel.*

The next property holds since we delete an edge from the kernel only when it gets deleted from G.

Property 3.19. *An edge gets deleted from the kernel only if it also gets deleted from the input graph.*

3.3.2. Two algorithms for maintaining an (ϵ, c)-kernel. In this section, we present two algorithms for maintaining an (ϵ, c)-kernel in a dynamic graph and analyze their amortized update times. Each of these algorithms uses the template described in section 3.3.1 and implies an algorithm for maintaining an approximately maximum matching in a dynamic graph. Our results are summarized in Theorems 3.20 and 3.22 and Corollaries 3.21 and 3.23.

Theorem 3.20. *There is an algorithm that maintains an (ϵ, c)-kernel $\kappa(G) = (V, \kappa(E))$ in a dynamic graph $G = (V, E)$. The algorithm has an amortized update time of $O(n/(\epsilon c))$, where $n = |V|$ is the number of nodes in the input graph.*

Proof. Initially, the input graph $G = (V, E)$ is empty, and all the nodes are slack. Thus, at this point in time, we have $E = \emptyset$, $T = \emptyset$, and $S = V$. Subsequently, we handle the insertion/deletion of an edge in G as per the procedures outlined in section 3.3.1. From Lemmas 3.13 and 3.15, we infer that this algorithm always maintains an (ϵ, c)-kernel in G. We now focus on bounding the amortized update time of this algorithm.

Clearly, handling an edge insertion takes $O(1)$ time in the worst case. On the other hand, handling an edge deletion takes $\Omega(1)$ time only if an endpoint (say x) of the edge being deleted changes from being tight to being slack. Further, the time spent on such an event is $O(n)$ since in the worst case, we might need to scan every neighbor of x in G and x can have at most $n - 1$ neighbors. However, Properties 3.17, 3.18, and 3.19 imply that between any two consecutive events like this, at least ϵc edges incident on x must have been deleted from G. Thus, we can charge the $O(n)$ time spent on the node x during this event to ϵc many edge deletions incident on x. This gives an amortized update time of $O(n/(\epsilon c))$. ∎

Corollary 3.21. *We can maintain a $(3 + \epsilon)$-approximate maximum matching in a dynamic graph $G = (V, E)$ in $O(\sqrt{n}/\epsilon)$ amortized update time.*

Proof. Set $c = \sqrt{n}$. We run the algorithm for maintaining an (ϵ, c)-kernel $\kappa(G) = (V, \kappa(E))$ as outlined in the proof of Theorem 3.20. We further maintain a 3/2-approximate maximum matching $M' \subseteq \kappa(E)$ in the kernel $\kappa(G)$ as per Theorem 3.4. Applying Theorem 3.12, we infer that the matching $M' \subseteq \kappa(E) \subseteq E$ is a $(3/2) \cdot (2 + 42\epsilon) = (3 + 21\epsilon)$-approximate maximum matching in the input graph $G = (V, E)$. It now remains to analyze the amortized update time of this procedure.

Theorem 3.20 implies that it takes only $O(n/(\epsilon c)) = O(\sqrt{n}/\epsilon)$ amortized update time to maintain the kernel $\kappa(G)$. For the rest of the proof, we therefore focus on bounding the amortized time spent on maintaining the matching $M' \subseteq \kappa(E)$ in the kernel. Note that due to an edge insertion in the input graph $G = (V, E)$, at most one edge can get inserted into the kernel. In contrast, due to the deletion of an edge (u, v) from G, at most c edges might get inserted into the kernel for every endpoint $x \in \{u, v\}$. However, as discussed in the proof of Theorem 3.20, such an event can occur only after the concerned endpoint x has lost ϵc many edges in G. Thus, on average, $O(1/\epsilon)$ edges get inserted into or deleted from the kernel per edge insertion/deletion in G. Hence, from Theorem 3.4 and Invariant 3.9, we infer that the amortized update time for maintaining the matching M' is at most $(1/\epsilon) \cdot (1 + \epsilon)c = O(\sqrt{n}/\epsilon)$. ∎

THEOREM 3.22. *There is an algorithm that maintains an (ϵ, c)-kernel $\kappa(G) = (V, \kappa(E))$ in a dynamic graph $G = (V, E)$. The algorithm has an amortized update time of $O(c + m/(\epsilon^2 c^2))$, where $m = |E|$ is the current number of edges in the input graph.*

Proof. The algorithm runs in *phases*, where each phase consists of a sequence of consecutive edge insertions/deletions in the input graph. Consider any phase φ, and let m denote the number of edges in the input graph $G = (V, E)$ in the beginning of a phase. Then the phase φ lasts for the next $m/(\epsilon^2 c^2)$ edge insertions/deletions in G. Typically, for large enough c, the quantity $m/(\epsilon^2 c^2)$ is sublinear in m, which implies that the total number of edges in G remains $\Theta(m)$ throughout the duration of the phase.

In the beginning of a phase, we compute a maximal c-matching $M \subseteq E$ in $G = (V, E)$. Let $T \subseteq V$ denote the subset of nodes with degree exactly equal to c in this maximal-c matching, and let $S = V \setminus T$ be the remaining set of nodes with degree less than c in M. From the discussion in section 3.1, it follows that the set of edges M, together with the partition (T, S) of the node set, defines a $(0, c)$-kernel $\kappa(G) = (V, \kappa(E))$ in G in the beginning of this phase, where $\kappa(E) = M$. Using the natural greedy algorithm, constructing this $(0, c)$-kernel requires $\Theta(m)$ time. Since a $(0, c)$-kernel is also an (ϵ, c)-kernel, we have an (ϵ, c)-kernel of G ready for us when the phase begins. During the phase, after each edge insertion/deletion in G, we update the kernel as per the procedures in section 3.3.1. When the current phase ends, we compute a new kernel in the beginning of the next phase and repeat. Lemmas 3.13 and 3.15 imply that this algorithm always maintains an (ϵ, c)-kernel in G. We now bound the amortized update time of this algorithm.

Throughout the following discussion, we repeatedly refer to the procedures for handling an edge insertion/deletion as outlined in section 3.3.1. The reader, therefore, will find it helpful to keep the descriptions of these two procedures in mind. First, note that handling an edge insertion in G in the middle of a phase requires $O(1)$ worst-case time (see Lemma 3.14). In contrast, while handling the deletion of an edge (u, v) from G in the middle of a phase, we spend time $O(1)$ *plus* the total number of neighbors of each endpoint $x \in \{u, v\}$ that are *scanned* by the concerned procedure (see Lemma 3.16). We claim that at most c neighbors of any endpoint $x \in \{u, v\}$ gets scanned after the deletion of an edge (u, v). This implies that an edge deletion in G is handled in $O(c)$ worst-case time in the middle of a phase. To see why the claim is true, note that as per the discussion in the previous paragraph, we have $|\kappa(\mathcal{N}(y))| \leq c$ for every node $y \in V$ in the beginning of the phase. During the phase, the value of $|\kappa(\mathcal{N}(y))|$ never exceeds c due to an edge insertion in G. In contrast, due to an edge deletion in G, the value of $|\kappa(\mathcal{N}(y))|$ increases by one only if some neighbor x of y becomes slack and we *scan* the edge (x, y). We now focus on such a neighbor x:

- Consider a time step (say t) in the middle of the phase when the node x scans the edge (x, y) and adds it to the kernel, thereby increasing the value of $|\kappa(\mathcal{N}_y)|$ by one. From the description of the procedure in section 3.3.1, it follows that the node x changes from being tight to being slack just before starting to scan its incident edges at time t. Let $t' < t$ be the last time step before t when the node x changed from being slack to being tight, and if no such time step exists, then let t' be the time step denoting the beginning of the phase. At time step t', we had $|\kappa(\mathcal{N}_x)| = c$, whereas at time step t, we have $|\kappa(\mathcal{N}_x)| \leq (1-\epsilon)c$. These observations follow from Properties 3.17 and 3.18. Furthermore, as per Property 3.19, the value of $|\kappa(\mathcal{N}_x)|$ decreases by one only when an edge incident on x gets deleted from G. Thus, we conclude that at least ϵc edges incident on x gets deleted from G during the time interval $[t', t]$. We *charge* the increase in the value of $|\kappa(\mathcal{N}_y)|$ at time step t to these edge deletions during time interval $[t', t]$. In other words, each time the value of $|\kappa(\mathcal{N}_y)|$ increases by one while handling an edge deletion in G, we can find ϵc new edge deletions in G that were responsible for this event. Since the current phase lasts only for $\epsilon^2 c^2$ many edge deletions in G and since $|\kappa(\mathcal{N}_y)| \leq c$ in the beginning of the phase, the value of $|\kappa(\mathcal{N}_y)|$ can never exceed $(1+\epsilon)c$ during the phase.

Now, let us look back at the scenario where a node x starts scanning its incident edges after becoming slack. Just before starting the scan, the node x had $|\kappa(\mathcal{N}_x)| = (1-\epsilon)c - 1$. During the scan, whenever the node x finds an edge (x, y) that is not in the kernel, the preceding discussion implies that $|\kappa(\mathcal{N}_y)| < (1+\epsilon)c$. In other words, each time the node x finds an incident edge that is not part of the kernel during the scan, it is able to include that edge into the kernel and increase the value of $|\kappa(\mathcal{N}_x)|$ by one. Accordingly, after scanning at most c of its incident edges, the node x will manage to ensure that $|\kappa(\mathcal{N}_x)|$ becomes equal to c. Hence, an edge deletion in G will require $O(c)$ time in the worst case in the middle of the phase.

To summarize, we have concluded that handling an edge insertion/deletion in G in the middle of the phase requires $O(c)$ time in the worst case. Next, recall that computing the initial kernel in the beginning of the phase requires $\Theta(m)$ time, but we can amortize this cost across the $\epsilon^2 c^2$ many edge insertions/deletions during the phase. This gives a total amortized update time of $O(c + m/(\epsilon^2 c^2))$. ◻

COROLLARY 3.23. *We can maintain a $(3+\epsilon)$-approximate maximum matching in a dynamic graph $G = (V, E)$ in $O(m^{1/3}/\epsilon^2)$ amortized update time.*

Proof. We maintain a kernel $\kappa(G) = (V, \kappa(E))$ in the input graph $G = (V, E)$ as per the algorithm outlined in the proof of Theorem 3.22. We set $c = m^{1/3}$. Hence, it takes $O(c + m/(\epsilon^2 c^2)) = O(m^{1/3}/\epsilon^2)$ amortized update time to maintain the kernel. We further maintain a matching $M' \subseteq \kappa(E)$ on the kernel as follows.

In the beginning of a phase, we simply compute a $(3/2)$-approximate maximum matching $M' \subseteq \kappa(E)$ in the kernel. This requires $\Theta(m)$ time. We distribute this cost over the duration of a phase, thereby contributing $O(m/(\epsilon^2 c^2)) = O(m^{1/3}/\epsilon^2)$ towards the amortized update time. During the phase, whenever an edge gets inserted into or deleted from the kernel, we update the matching M' as per Theorem 3.4. Using exactly the same argument as in the proof of Corollary 3.21, we conclude that each edge insertion/deletion in G leads to $O(1/\epsilon)$ edge insertions/deletions in the kernel $\kappa(G)$ on average. Hence, as per Theorem 3.4 and Invariant 3.9, maintaining the matching M' during a phase requires $O((1+\epsilon)c/\epsilon) = O(m^{1/3}/\epsilon)$ amortized update time. To summarize, we spend an overall $O(m^{1/3}/\epsilon^2)$ amortized update time to maintain the matching M' in the kernel $\kappa(G)$. It now remains to analyze the approximation guarantee.

By Theorem 3.12, the kernel $\kappa(G)$ preserves the size of the maximum matching in G up to a factor of $(2+42\epsilon)$. Since M' is a $(3/2)$-approximate maximum matching in the kernel $\kappa(G)$, we infer that M' is also a $(3/2)\cdot(2+42\epsilon) = (3+21\epsilon)$-approximate maximum matching in the input graph G. ◻

3.4. Getting worst-case update time: Proof of Theorem 3.2. To get worst-case update time, we extend the algorithm outlined in the proofs of Theorem 3.22 and Corollary 3.23. Recall that this algorithm runs in *phases*. The key observation is that within a phase, we already have $O(c)$ worst-case update time. Specifically, we have the following lemma.

Lemma 3.24. *Consider the algorithm presented in the proof of Theorem* 3.22 *for maintaining an* (ϵ, c)*-kernel* $\kappa(G) = (V, \kappa(E))$ *in the input graph* $G = (V, E)$*. Within a given phase, the algorithm has* $O(c)$ *worst-case update time. Furthermore, within a given phase, one edge insertion/deletion in* G *can lead to at most* $O(c)$ *edge insertions and* $O(1)$ *edge deletions in* $\kappa(G)$*.*

Proof. Consider the insertion of an edge (u, v) in G in the middle of a phase. This insertion is handled as per the procedure section 3.3.1. By Lemma 3.14, this takes $O(1)$ time, and clearly this leads to at most one edge insertion in the kernel $\kappa(G)$.

Next, consider the deletion of an edge (u, v) from G in the middle of a phase. This deletion is also handled as per the procedure in section 3.3.1. By Lemma 3.16, this takes time proportional to the total number of neighbors of the endpoints $x \in \{u, v\}$ that are *scanned*. From the description of the procedure, it follows that at most $O(c)$ edges get inserted into the kernel during the scan. Finally, in the proof of Theorem 3.22, we argued that at most c edges get scanned for every endpoint $x \in \{u, v\}$. Hence, the procedure takes $O(c)$ worst-case time. ◻

We claim that using Lemma 3.24, one can maintain, *within a given phase*, a $(4+\epsilon)$-approximate maximum matching in G in $O(c) = O(m^{1/3})$ *worst-case* update time. To see why this is true, within a given phase simply maintain a maximal matching $M \subseteq \kappa(E)$ in the kernel using Theorem 3.3. Because of maximality, the matching M will be a 2-approximation to the size of the maximum matching in the kernel $\kappa(G)$. Hence, by Theorem 3.12, the same matching M will be a $2\cdot(2+42\epsilon) = (4+84\epsilon)$-approximate maximum matching in G. By Lemma 3.24, an edge insertion/deletion in G leads to at most $O(c)$ edge insertions and at most one edge deletion in the kernel $\kappa(G)$. By Theorem 3.3, while maintaining the matching M in the kernel $\kappa(G)$, each edge insertion in $\kappa(G)$ is handled in $O(1)$ worst-case time, and each edge deletion in $\kappa(G)$ is handled in $O(c)$ worst-case time. Thus, it takes $O(c)$ time in the worst-case to update the matching in M after an edge insertion/deletion in the input graph G.[3] To summarize, within a given phase, we can maintain a $(4+\epsilon)$-approximate maximum matching in G in $O(c) = O(m^{1/3})$ worst-case update time.

[3]At this point, it should be clear to the reader why we could not get $(3+\epsilon)$-approximation in $O(m^{1/3}/\epsilon^2)$ worst-case update time, for that would require us to maintain a $(3/2)$-approximate matching in the kernel. Theorem 3.4 would then imply that we would have to spend $O(c)$ worst-case time after every edge insertion in the kernel. Coupled with the fact that an edge insertion/deletion in G can lead to $\Omega(c)$ edge insertions in $\kappa(G)$, we would end up with a worst-case update time of $\Omega(c^2)$.

On the other hand, in the beginning of a phase, we have to compute an initial kernel $\kappa(G) = (V, \kappa(E))$, with $\kappa(E) \subseteq E$, and a matching $M \subseteq \kappa(E)$ in this kernel. This takes $\Theta(m)$ time, and we distribute this cost over the $\Theta(\epsilon^2 c^2)$ edge insertions/deletions during a phase. This leads to an amortized update time of $O(m/(\epsilon^2 c^2))$. *This is the only place where we have to use amortization.* However, we can make the update time for this part of the algorithm worst case by using the standard concept of *periodic rebuilding*. Basically, we prepare for the next phase while handling the edge insertions/deletions in the current phase. Note that we need to do $\Theta(m)$ amount of computation at the end of the current phase to get the initial kernel and the matching for the next phase. We perform $\Theta(m/(\epsilon^2 c^2))$ amount of this computation after each edge insertion/deletion in the current phase. Thus, when the current phase ends after $\epsilon^2 c^2$ edge insertions/deletions, we have already done $\Theta(m/(\epsilon^2 c^2)) \cdot (\epsilon^2 c^2) = \Theta(m)$ amount of computation, which is sufficient to create the initial kernel and the matching for the next phase. This way, we get $\Theta(m/(\epsilon^2 c^2))$ worst-case update time for preparing the kernel and the matching in the beginning of a phase. Adding the time for maintaing the kernel and the matching in the middle of a phase, we get a total worst-case update time of $\Theta(c + m/(\epsilon^2 c^2)) = \Theta(m^{1/3}/\epsilon^2)$.

REFERENCES

[1] A. Abboud and V. V. Williams, *Popular conjectures imply strong lower bounds for dynamic problems*, in 55th IEEE Symposium on Foundations of Computer Science, Philadelphia, PA, IEEE Computer Society, 2014, pp. 434–443.

[2] S. Baswana, M. Gupta, and S. Sen, *Fully dynamic maximal matching in $O(\log n)$ update time*, in 52nd IEEE Symposium on Foundations of Computer Science, Palm Springs, CA, IEEE Computer Society, 2011, pp. 383–392.

[3] A. Bernstein and C. Stein, *Fully dynamic matching in bipartite graphs*, in 42nd International Colloquium on Automata, Languages, and Programming, Kyoto, Japan, Springer, 2015, pp. 167–179.

[4] A. Bernstein and C. Stein, *Faster fully dynamic matchings with small approximation ratios*, in 27th ACM-SIAM Symposium on Discrete Algorithms, Arlington, VA, SIAM, 2016, pp. 692–711.

[5] S. Bhattacharya, D. Chakrabarty, and M. Henzinger, *Deterministic fully dynamic approximate vertex cover and fractional matching in $O(1)$amortized update time*, in 19th International Conference on Integer Programming and Combinatorial Optimization, Waterloo, ON, Springer, 2017, pp. 86–98.

[6] S. Bhattacharya, M. Henzinger, and D. Nanongkai, *New deterministic approximation algorithms for fully dynamic matching*, in 48th ACM Symposium on Theory of Computing, Cambridge, MA, ACM, 2016, pp. 398–411.

[7] S. Bhattacharya, M. Henzinger, and D. Nanongkai, *Fully dynamic approximate maximum matching and minimum vertex cover in O(log^3 n) worst case update time*, in 28th ACM-SIAM Symposium on Discrete Algorithms, Barcelona, Spain, SIAM, 2017, pp. 470–489.

[8] R. Duan and S. Pettie, *Approximating maximum weight matching in near-linear time*, in 51st IEEE Symposium on Foundations of Computer Science, Las Vegas, NV, IEEE Computer Society, 2010, pp. 673–682.

[9] R. Duan and S. Pettie, *Linear-time approximation for maximum weight matching*, J. ACM, 61 (2014), pp. 1:1–1:23.

[10] H. N. Gabow and R. E. Tarjan, *Faster scaling algorithms for general graph-matching problems*, J. ACM, 38 (1991), pp. 815–853.

[11] A. Gupta, R. Krishnaswamy, A. Kumar, and D. Panigrahi, *Online and dynamic algorithms for set cover*, in 49th ACM Symposium on Theory of Computing, Montreal, QC, ACM, 2017, pp. 537–550.

[12] M. Gupta and R. Peng, *Fully dynamic $(1+\epsilon)$-approximate matchings*, in 54th IEEE Symposium on Foundations of Computer Science, Berkeley, CA, IEEE Computer Society, 2013, pp. 548–557.

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

[14] Z. Ivkovic and E. L. Lloyd, *Fully dynamic maintenance of vertex cover*, in Graph-Theoretic Concepts in Computer Science, Springer, 1993, pp. 99–111.

[15] S. Khot and O. Regev, *Vertex cover might be hard to approximate to within* $2-\epsilon$, J. Comput. Syst. Sci., 74 (2008), pp. 335–349.

[16] L. Lovász and M. D. Plummer, *Matching Theory*, Akadémiai Kiadó, Budapest, 1986. Also published as Vol. 121 of the North-Holland Mathematics Studies, North-Holland Publishing, Amsterdam.

[17] A. Madry, *Navigating central path with electrical flows: From flows to matchings, and back*, in 54th IEEE Symposium on Foundations of Computer Science, Berkeley, CA, IEEE Computer Society, 2013, pp. 253–262.

[18] S. Micali and V. V. Vazirani, *An $O(\sqrt{|V|}\,|E|)$ algorithm for finding maximum matching in general graphs*, in 21st IEEE Symposium on Foundations of Computer Science, Syracuse, NY, IEEE Computer Society, 1980, pp. 17–27.

[19] M. Mucha and P. Sankowski, *Maximum matchings via Gaussian elimination*, in 45th IEEE Symposium on Foundations of Computer Science, Rome, Italy, IEEE Computer Society, 2004, pp. 248–255.

[20] O. Neiman and S. Solomon, *Simple deterministic algorithms for fully dynamic maximal matching*, in 45th ACM Symposium on Theory of Computing, Palo Alto, CA, ACM, 2013, pp. 745–754.

[21] K. Onak and R. Rubinfeld, *Maintaining a large matching and a small vertex cover*, in 42nd ACM Symposium on Theory of Computing, Cambridge, MA, ACM, 2010, pp. 457–464.

[22] P. Sankowski, *Faster dynamic matchings and vertex connectivity*, in 18th ACM-SIAM Symposium on Discrete Algorithms, New Orleans, LA, SIAM, 2007, pp. 118–126.

[23] S. Solomon, *Fully dynamic maximal matching in constant update time*, in 57th IEEE Symposium on Foundations of Computer Science, New Brunswick, NJ, IEEE Computer Society, 2016, pp. 325–334.