Deterministic Fully Dynamic Data Structures for Vertex Cover and Matching

Sayan Bhattacharya* Monika Henzinger† 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 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 $O(m^{1/3}/\epsilon^2)$ *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 [13].

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 [5, 10]. Using this algorithm, a $(1+\epsilon)$-approximate maximum matching can be constructed in $O(m/\epsilon)$ time [3]. 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). Under the unique games conjecture, the minimum vertex cover cannot be efficiently approximated within any constant factor better than 2 [7]. Thus, under the unique games conjecture, the 2-approximation in $O(m)$ time by the greedy method 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 $O(\text{polylog}(n))$ would be a breakthrough, since it would immediately improve the longstanding bounds of the static algorithms [5, 9, 10, 11]. The best known update bound for dynamic maximum matching is obtained by a randomized data structure of Sankowski [14], 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 last few years there has been a growing interest in designing efficient dynamic data structures for maintaining approximate solutions to both these problems.

Previous work. A maximal matching can be maintained in $O(n)$ worst-case update time by a trivial deterministic algorithm. Ivković and Lloyd [6] showed how to improve this bound to $O((n+m)^{\sqrt{2}/2})$. Onak

*University of Vienna, Faculty of Computer Science. Email: jucse.sayan@gmail.com. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement no. 340506.

†University of Vienna, Faculty of Computer Science. Email: monika.henzinger at univie.ac.at. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement no. 340506.

‡Università di Roma "Tor Vergata", Rome, Italy. Email: giuseppe.italiano@uniroma2.it. Partially supported by MIUR, the Italian Ministry of Education, University and Research, under Project AMANDA (Algorithmics for MAssive and Networked DAta).

and Rubinfeld [13] 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 [12] showed that a 3/2-approximate maximum matching can be maintained dynamically in $O(\sqrt{m})$ worst-case time per update. Their data structure maintains a maximal matching and thus achieves the same update bound also for 2-approximate minimum vertex cover. Furthermore, Gupta and Peng [4] 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 [13] gave a deterministic data structure that maintains an $O(\log n)$-approximate minimum vertex cover in $O(\log^2 n)$ amortized update time.

Very recently, Abboud and Vassilevska 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).

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 in [4, 12] is a natural barrier for deterministic data structures. In particular, in their pioneering work on these problems, Onak and Rubinfeld [13] asked:

- "*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., [8]). 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 $O(m^{1/3}/\epsilon^2)$ 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.

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, 13] 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

Problem	Approximation Guarantee	Update Time	Data Structure	Reference
MM & MVC	$O(1)$	$O(\log^2 n)$ amortized	randomized	[13]
MM & MVC	2	$O(\log n)$ amortized	randomized	[2]
MM	1.5	$O(\sqrt{m})$ worst-case	deterministic	[12]
MVC	2	$O(\sqrt{m})$ worst-case	deterministic	[12]
MM	$1+\epsilon$	$O(\sqrt{m}/\epsilon^2)$ worst-case	deterministic	[4]
MVC	$2+\epsilon$	$O(\log n/\epsilon^2)$ 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**

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

the framework of Onak and Rubinfeld [13]. 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 [13] 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.

2 Deterministic Fully Dynamic Vertex Cover

The 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 already 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.

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 1). If the (α, β)-partition satisfies an additional property (Invariant 1), then from it we can easily derive a $2\alpha\beta$-approximation to the minimum vertex cover (Theorem 2.2). In Section 2.3, we present a natural deterministic algorithm for maintaining such an (α, β)-partition, and analyze it in Sections 2.5, 2.6 by setting $\alpha \leftarrow 1+2\epsilon$ and $\beta \leftarrow 1+\epsilon$. Our main result is summarized in the theorem below.

THEOREM 2.1. *For every $\epsilon \in (0, 1]$, we can deterministically maintain a $(2+\epsilon)$-approximate 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 1. *An* (α, β)-partition *of the graph G partitions its node-set V into subsets $V_0 \ldots V_L$, where $L = \lceil \log_\beta(n/\alpha) \rceil$ and $\alpha, \beta > 1$. For $i \in \{0, \ldots, L\}$, we identify the subset V_i as the i^{th}* 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 i^{th} 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 $[i,j]$.

$$\mathcal{N}_v = \{u \in V : (u,v) \in E\} \quad \forall v \in V. \tag{2.1}$$

$$\mathcal{N}_v(i) = \{u \in \mathcal{N}_v \cap V_i\} \quad \forall v \in V; i \in \{0,\ldots,L\} \tag{2.2}$$

$$\mathcal{N}_v(i,j) = \bigcup_{k=i}^{j} \mathcal{N}_v(k) \quad \forall v \in V; i,j \in \{0,\ldots,L\}, i \leq j. \tag{2.3}$$

Similarly, define the notations $\mathcal{D}_v$ and $\mathcal{D}_v(i,j)$. Note that $\mathcal{D}_v$ is the degree of a node $v \in V$.

$$\mathcal{D}_v = |\mathcal{N}_v| \tag{2.4}$$

$$\mathcal{D}_v(i,j) = |\mathcal{N}_v(i,j)| \tag{2.5}$$

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 were to go to the i^{th} 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 $v \in V$.

$$W_v = \sum_{u \in \mathcal{N}_v} \beta^{-\max(\ell(u),\ell(v))}. \tag{2.6}$$

$$W_v(i) = \sum_{u \in \mathcal{N}_v} \beta^{-\max(\ell(u),i)} \quad \forall i \in \{0,\ldots,L\}. \tag{2.7}$$

LEMMA 2.1. *Every (α, β)-partition of the graph G satisfies the following conditions for all nodes $v \in V$.*

$$W_v(L) \leq \alpha \tag{2.8}$$

$$W_v(L) \leq \cdots \leq W_v(i) \leq \cdots \leq W_v(0) \tag{2.9}$$

$$W_v(i) \leq \beta \cdot W_v(i+1) \quad \forall i \in \{0,\ldots,L-1\}. \tag{2.10}$$

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

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

We now fix any level $i \in \{0,\ldots,L-1\}$ and show that the (α, β)-partition satisfies equation 2.9.

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

Finally, we prove equation 2.10.

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

Fix any 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 2.1 states that $W_v(i) \leq \alpha$ 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.

We will maintain a specific type of (α, β)-partition, where each node is assigned to a level in a way that satisfies Invariant 1.

INVARIANT 1. *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 1. 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 1 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 2.2. *Consider an (α, β)-partition of the graph G that satisfies Invariant 1. 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 1 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 every edge $(u,v) \in E$ to an extent of $x(u,v) = w(u,v)/(\alpha\beta) \in [0,1]$. 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 $|M_f|$.

$$|V^*| = \sum_{v \in V^*} 1 \leq \sum_{v \in V^*} W_v = \sum_{v \in V^*} \sum_{u \in \mathcal{N}_v} w(u,v)$$

$$\leq \sum_{v \in V} \sum_{u \in \mathcal{N}_v} w(u,v) = 2 \cdot \sum_{(u,v) \in E} w(u,v) = (2\alpha\beta) \cdot |M_f|$$

2.2 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 or not a given node is part of this vertex cover. In $O(1)$ time we can also report the size of the vertex cover.

2.3 Handling the insertion/deletion of an edge. A node is called *dirty* if it violates Invariant 1, 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 t^{th} update in G. By induction hypothesis, at this instant every node is clean. Then the t^{th} 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 call the subroutine in Figure 1.

```
01. WHILE there exists a dirty node v
02.     IF W_v > αβ, THEN
            // If true, then by equation 2.8 ℓ(v) < L.
03.         Increment the level of v
            by setting ℓ(v) ← ℓ(v) + 1.
04.     ELSE IF (W_v < 1 and ℓ(v) > 0), THEN
05.         Decrement the level of v
            by setting ℓ(v) ← ℓ(v) − 1.
```

Figure 1: RECOVER().

Consider any node $v \in V$ and suppose that $W_v = W_v(\ell(v)) > \alpha\beta$. In this event, equation 2.8 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 02), 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 1 holds.

Comparison with the framework of Onak and Rubinfeld [13]. As described below, there are two significant differences between our framework and that of [13]. Consequently, many of the technical details of our approach (illustrated in Section 2.6) differ from the proof in [13].

First, in the hierarchical partition of [13], 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 [13] 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 till date 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 Data structures. We now describe the relevant data structures that will be used to implement our algorithm.

- We maintain for each node $v \in V$:
  - A counter LEVEL[v] to keep track of the current level of v. Thus, we set LEVEL[v] $\leftarrow \ell(v)$.
  - A counter WEIGHT[v] to keep track of the weight of v. Thus, we set WEIGHT[v] $\leftarrow W_v$.
  - For every level $i >$ LEVEL[v], the set of nodes $\mathcal{N}_v(i)$ in the form of a doubly linked list NEIGHBORS$_v$[i]. For every level $i \leq$ LEVEL[v], the list NEIGHBORS$_v$[i] is empty.

– For level $i = \text{LEVEL}[v]$, the set of nodes $\mathcal{N}_v(0,i)$ in the form of a doubly linked list $\text{NEIGHBORS}_v[0,i]$. For every level $i \neq \text{LEVEL}[v]$, the list $\text{NEIGHBORS}_v[0,i]$ is empty.

- When the graph G gets updated due to an edge insertion/deletion, we may discover that a node violates Invariant 1. Such a node is called *dirty*, and we store the set of such nodes as a doubly linked list DIRTY-NODES. For every node $v \in V$, we maintain a bit $\text{STATUS}[v] \in \{\text{dirty}, \text{clean}\}$ that indicates if the node is dirty or not. Every dirty node stores a pointer to its position in the list DIRTY-NODES.

- We use the phrase *"neighborhood lists of v"* to refer to the collection of linked lists $\bigcup_{i=0}^{L} \{\text{NEIGHBORS}_v[0,i], \text{NEIGHBORS}_v[i]\}$. 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. Using these pointers, we can update the neighborhood lists of u and v when the edge (u,v) is inserted into (resp. deleted from) the graph, or when the node v increases (resp. decreases) its level by one.

2.5 Bounding the amortized update time. In the full version of this paper, we present a detailed implementation of our algorithm using the data structures described in Section 2.4. We also prove that for any $\epsilon \in [0,1]$, $\alpha = 1+2\epsilon$ and $\beta = 1+\epsilon$, it takes $O(t \log n/\epsilon^2)$ time to handle t edge insertions/deletions in G starting from an empty graph. This gives an amortized update time of $O(\log n/\epsilon^2)$, and by Theorem 2.2, a $(2+10\epsilon)$-approximation to the minimum vertex cover in G. Specifically we show that after an edge insertion or deletion the data structure can be updated in time $O(1)$ plus the time to adjust the levels of the nodes, i.e., the time for procedure RECOVER. To bound the latter we show that it takes time $\Theta(1+\mathcal{D}_v(0,i))$, when node v changes from level i to level $i+1$ or level $i-1$ and prove the bound on the total time spent in procedure RECOVER using a potential function based argument. Due to space constraints, we give a (slightly) simplified variant of this argument here, which gives an amortized bound on *the number of times we have to change the weight of an already existing edge*. This number is $\mathcal{D}_v(0,i)$, when node v changes from level i to level $i+1$ and $\mathcal{D}_v(0,i-1)$, when node v changes from level i to level $i-1$.[1] Specifically, we prove Theorem 2.3 in Section 2.6, which implies that on average we change the weights of $O(L/\epsilon) = O(\log n/\epsilon^2)$ edges per update in G.

THEOREM 2.3. *Set $\alpha \leftarrow 1+2\epsilon$, $\beta \leftarrow 1+\epsilon$. In the beginning, when G is an empty graph, initialize a counter* $\text{COUNT} \leftarrow 0$. *Subsequently, each time we change the weight of an already existing edge in the hierarchical partition, set* $\text{COUNT} \leftarrow \text{COUNT}+1$. *Then* $\text{COUNT} = O(tL/\epsilon)$ *just after we handle the t^{th} update in G.*

The proof in Section 2.6 uses a carefully chosen potential function. As the formal analysis is quite involved, some high level intuitions are in order. Below, we give a brief overview of our approach. To highlight the main ideas, in contrast with Theorem 2.3, we assume that α and β are some sufficiently large constants.

Define the level of an edge (y,z) to be $\ell(y,z) = \max(\ell(y),\ell(z))$, and note that the weight $w(y,z)$ decreases (resp. increases) iff the edge's level $\ell(y,z)$ goes up (resp. down). There is a potential associated with both nodes and edges. Note that we use the terms "tokens" and "potential" interchangeably.

Each edge e has exactly $2(L-\ell(e))$ tokens. These tokens are assigned as follows. Whenever a new edge is inserted, it receives $2(L-\ell(e))$ tokens. When e moves up a level, it gives one token to each endpoint. Whenever e is deleted, it gives one token to each endpoint. Whenever e moves down a level because one endpoint, say u, moves down a level, e receives two tokens from u.

Initially and whenever a node moves a level higher, it has no tokens. *Whenever a node u moves up a level*, only its adjacent edges to the same or lower levels have to be updated as their level changes. The level of all other edges is unchanged. Recall that each such edge gives 1 token to u, which in turn uses this token to pay for updating the edge. *Whenever a node u moves down a level*, say from k to $k-1$, it has at most β^k adjacent edges at level k or below. These are all the edges whose level needs to be updated (which costs a token) and whose potential needs to be increased by two tokens. In this case we show that u has enough tokens, (i) to pay for the work involved in the update, (ii) to give two tokens to each of the at most β^k adjacent edges, *and* (iii) to still have a sufficient number of tokens for being on level $k-1$.

Whenever the level of a node v is not modified but its weight W_v decreases because the weight of the adjacent edge (u,v) decreases, the level of (u,v) must have increased and (u,v) gives one token to v (the other one goes to u). Note that this implies that a change in

[1] The proof actually shows a stronger result assuming that the level change of node v from i to $i-1$ causes $\Theta(\mathcal{D}_v(0,i))$ many edges to change their level.

W_v by at most $\beta^{-\ell(v)}$ increases the potential of v by 1, i.e., the "conversion rate" between weight changes and token changes is $\beta^{\ell(v)}$. *Whenever the level of v does not change but its weight W_v increases* as the level of (u, v) has decreased, no tokens are transferred between v and (u, v). (Technically the potential of v might fall slightly but the change might be so small that we ignore it.) Formally we achieve these potential function changes by setting the potential of every node in V_0 to 0 and for every other node to $\beta^{\ell(v)} \cdot \max(0, \alpha - W_v)$.

Thus, the crucial claim is that a node v that moves down to level $k-1$ has accumulated a sufficent number of tokens, i.e., at least $X := 3\beta^k + \beta^{k-1}\max(0, \alpha - W_v(k-1))$ tokens. *Case 1:* Assume first that v's immediately preceeding level was on level $k-1$, i.e. that v just had moved up from level $k-1$. Recall that, by the definition of the potential function, v had *no* tokens when it moved up from level $k-1$. However, in this case we know that W_v was least $\alpha\beta$ and, thus, after adjusting the weight of its adjacent edges to the level change, W_v is still at least α after the level change. Node v only drops to level $k-1$ if $W_v < 1$, i.e., while being on level k its weight must have dropped by at least $\alpha - 1$. By the above "conversion rate" between weight and tokens this means that v must have received at least $\beta^k(\alpha - 1)$ tokens while it was on level k, which at least X for large enough α. *Case 2:* Assume next that node v was at level $k+1$ immediately before level k. Right after dropping from level $k+1$ node v owned $\beta^k(\alpha - W_v(k))$ tokens. As v has not changed levels since, it did not have to give any tokens to edges and did not have to pay for any updates of its adjacent edges. Instead it might have received some tokens from inserted or deleted adjacent edges. Thus, it still ownes at least $\beta^k(\alpha - W_v(k))$ tokens. As $W_v(k) \leq W_v(k-1)$ and $W_v(k) < 1$ when v drops to level $k-1$, this number of tokens is at least X for $\beta \geq 2$ and $\alpha \geq 3\beta + 1$.

To summarize, whenever an edge is inserted it receives a sufficient number of tokens to pay the cost of future upwards level changes, but also to give a token to its "lower" endpoint every time its level increases. These tokens accumulated at the "lower" endpoints are sufficient to pay for level decreases of these endpoints because (a) nodes move up to a level when their weight on the new level is at least $\alpha > 1$ but only move down when their weight falls below 1 and (b) the weight of edges on the same and lower levels drops by a factor of β between two adjacent levels. Thus $\beta^k(\alpha - 1)$ many edge deletions or edge weight decreases of edges adjacent to node v are necessary to cause v to drop from level k to level $k-1$ (each giving one token to v), while there are only β^{k-1} many edges on levels below k that need to be updated when v drops. Thus, the cost of v's level drop is β^{k-1} and the new potential needed for v on level $k-1$ is $\beta^{k-1}(\alpha - 1)$, but v has collected at least $\beta^k(\alpha - 1)$ tokens, which, by suitable choice of β and α, is sufficient.

2.6 Proof of Theorem 2.3. Recall that the level of an edge (y, z) is defined as $\ell(y, z) = \max(\ell(y), \ell(z))$. 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 L/ϵ dollars are deposited to the bank account; and every time our algorithm changes the level of an already existing edge, 1 dollar is withdrawn from it. We show that the bank account never runs out of money, and this implies that $\textsc{Count} = O(tL/\epsilon)$ after t edge insertions/deletions 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.

$$\mathcal{B} = (1/\epsilon) \cdot \left(\sum_{e \in E} \Phi(e) + \sum_{v \in V} \Psi(v) \right) \tag{2.11}$$

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

Invariant 2. *For every edge $(u, v) \in E$, we have:*

$$\Phi(u, v) = (1 + \epsilon) \cdot (L - \max(\ell(u), \ell(v)))$$

Invariant 3. *For every node $v \in V$, we have:*

$$\Psi(v) = \left(\beta^{\ell(v)+1}/(\beta - 1) \right) \cdot \max(0, \alpha - W_v)$$

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 3 sets $\Psi(v) = 0$ for all nodes $v \in V$. Consequently, equation 2.11 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. Theorem 2.3, therefore, will follow if we can prove the next two lemmas. Their proofs appear in Section 2.6.1 and Section 2.6.2 respectively.

Lemma 2.2. *Consider the insertion (resp. deletion) of an edge (u, v) in G. It creates (resp. destroys) the weight $w(u, v) = \beta^{-\max(\ell(u), \ell(v))}$, creates (resp.*

destroys) the potential $\Phi(u,v)$, and increases (resp. decreases) the potential $\Psi(u)$ (resp. $\Psi(v)$). Due to these changes, the total potential $\mathcal{B}$ increases by at most L/ϵ.

LEMMA 2.3. *During every single iteration of the* WHILE *loop in Figure 1, the total increase in* COUNT *is no more than the net decrease in the potential $\mathcal{B}$.*

2.6.1 Proof of Lemma 2.2. If an edge (u,v) is inserted into G, then the potential $\Phi(u,v)$ is created and gets a value of at most $(1+\epsilon)L$ units. As the weight W_u (resp. W_v) increases, it follows that the potential $\Psi(u)$ (resp. $\Psi(v)$) does not increase. All the other potentials remain unchanged. Thus, the net increase in $\mathcal{B}$ is at most $(1/\epsilon)\cdot(1+\epsilon)L = (1+1/\epsilon)L \leq L/\epsilon$.

In contrast, if an edge (u,v) is deleted from G, then the potential $\Phi(u,v)$ is destroyed. The weight W_u (resp. W_v) decreases by at most $\beta^{-\ell(u)}$ (resp. $\beta^{-\ell(v)}$). Hence, each of the potentials $\Psi(u), \Psi(v)$ increases by at most $\beta/(\beta-1) = 1+1/\epsilon \leq (1+\epsilon)/\log(1+\epsilon) \leq ((\log n)-1)/\log(1+\epsilon) \leq L$ for $n \geq 8$. The potentials of the remaining nodes and edges do not change. Hence, by equation 2.11, the net increase in $\mathcal{B}$ is at most L/ϵ.

2.6.2 Proof of Lemma 2.3.
Throughout this section, fix a single iteration of the WHILE loop in Figure 1 and suppose that it changes the level of a dirty node v by one unit. We use the superscript 0 (resp. 1) on a symbol to denote its state at the time instant immediately prior to (resp. after) that specific iteration of the WHILE loop. Further, we preface a symbol with δ to denote the net decrease in its value due to that iteration. For example, consider the potential $\mathcal{B}$. We have $\mathcal{B} = \mathcal{B}^0$ immediately before the iteration begins, and $\mathcal{B} = \mathcal{B}^1$ immediately after iteration ends. We also have $\delta\mathcal{B} = \mathcal{B}^0 - \mathcal{B}^1$.

A change in the level of node v 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 equation 2.11, gives us the following guarantee.

(2.12)
$$\delta\mathcal{B} = (1/\epsilon)\cdot\left(\delta\Psi(v) + \sum_{u\in\mathcal{N}_v}\delta\Phi(u,v) + \sum_{u\in\mathcal{N}_v}\delta\Psi(u)\right)$$

A change in the level of node v does not affect the (a) the neighborhood structure of the node v, and (b) the level and the overall degree of any node $u \neq v$. Thus, we get the following equalities.

$$\mathcal{N}_v^0(i) = \mathcal{N}_v^1(i) \quad \text{for all } i \in \{0,\ldots,L\}. \tag{2.13}$$
$$\ell^0(u) = \ell^1(u) \quad \text{for all } u \in V\setminus\{v\}. \tag{2.14}$$
$$D_u^0 = D_u^1 \quad \text{for all } u \in V\setminus\{v\}. \tag{2.15}$$

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 of the WHILE loop we are concerned about.

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

We divide the proof of Lemma 2.3 into two possible cases, depending upon whether the concerned iteration of the WHILE loop 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 increase in COUNT due to 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 equation 2.12, we derive that $\delta\mathcal{B}$ is sufficiently large to pay for the increase in COUNT.

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

CLAIM 1. *We have* $\text{COUNT}^1 - \text{COUNT}^0 = \mathcal{D}_v(0,k)$.

Proof. When the node v changes its level from k to $(k+1)$, this only affects the levels of those edges that are of the form (u,v), where $u \in \mathcal{N}_v(0,k)$.

CLAIM 2. *We have* $\delta\Psi(v) = 0$.

Proof. Since the node v increases its level from k to $(k+1)$, Step 02 (Figure 1) guarantees that $W_v^0 = W_v(k) > \alpha\beta$. Next, from Lemma 2.1 we infer that $W_v^1 = W_v(k+1) \geq \beta^{-1}\cdot W_v(k) > \alpha$. Since both $W_v^0, W_v^1 > \alpha$, we get: $\Psi^0(v) = \Psi^1(v) = 0$. It follows that $\delta\Psi(v) = \Psi^0(v) - \Psi^1(v) = 0$.

CLAIM 3. *For every node $u \in \mathcal{N}_v$, we have:*

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

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 $\delta\Phi(u,v) = \Phi^0(u,v) - \Phi^1(u,v) = 0$.

CLAIM 4. *For every node $u \in \mathcal{N}_v$, we have:*

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

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, we infer that:

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

From Claims 2, 3, 4 and equation 2.12, we derive the following bound.

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

Thus, Claim 1 implies that the net decrease in the potential $\mathcal{B}$ in no less than the increase in COUNT. This proves Lemma 2.3.

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

CLAIM 5. *We have $W_v^0 = W_v(k) < 1$, $D_v(0,k) \leq \beta^k$.*

Proof. Since the node v decreases its level from k to $(k-1)$, Step 04 (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 $D_v(0,k) \leq \beta^k$.

CLAIM 6. *We have* $\text{COUNT}^1 - \text{COUNT}^0 \leq \beta^k$.

Proof. The node v decreases its level from k to $k-1$. Due to this event, the level of an edge changes iff it is of the form (u,v) with $u \in \mathcal{N}_v(0,k-1)$. Thus, we have $\text{COUNT}^1 - \text{COUNT}^0 = D_v(0,k-1) \leq D_v(0,k) \leq \beta^k$.

CLAIM 7. *For all $u \in \mathcal{N}_v$, 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 $\Psi(u) = \beta^{\ell(u)}\cdot\max(0, \alpha - W_u)$, we derive that $\Psi^0(u) \geq \Psi^1(u)$. Thus, we have $\delta\Psi(u) = \Psi^0(u) - \Psi^1(u) \geq 0$.

CLAIM 8. *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 consider two possible scenarios.

1. We have $u \in \mathcal{N}_v(k,L)$. As the level of the node v decreases 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 level of node v decreases 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 $\delta\Phi(u,v) = \Phi^1(u,v) - \Phi^0(u,v) = -(1+\epsilon)$.

This concludes the proof of the claim.

We now 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. Note that $x = \sum_{u\in\mathcal{N}_v(0,k-1)} w^0(u,v) = \beta^{-k}\cdot D_v(0,k-1)$. Thus, we get the following equations.

$$W_v^0 = x + y \leq 1 \tag{2.16}$$

$$x = \beta^{-k}\cdot D_v(0,k-1) \tag{2.17}$$

$$y = \sum_{u\in\mathcal{N}_v(k,L)} w^0(u,v) \tag{2.18}$$

Equation 2.16 holds due to Claim 5.

CLAIM 9. *We have $\sum_{u\in\mathcal{N}_v}\delta\Phi(u,v) = -(1+\epsilon)\cdot x\cdot\beta^k$.*

Proof. Claim 8 implies that $\sum_{u\in\mathcal{N}_v}\delta\Phi(u,v) = -(1+\epsilon)\cdot D_v(0,k-1)$. Applying equation 2.17, we infer that $D_v(0,k-1) = x\cdot\beta^k$.

CLAIM 10. *We have:*

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

Proof. Equation 2.16 states that $W_v^0 = x + y < 1$. Since $\ell^0(v) = k$, we get:

$$\Psi^0(v) = (\alpha - x - y)\cdot\left(\beta^{k+1}/(\beta-1)\right) \tag{2.19}$$

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.

$$\Psi^1(v) = \max(0, \alpha - \beta x - y)\cdot\left(\beta^k/(\beta-1)\right) \tag{2.20}$$

Since $\delta\Psi(v) = \Psi^0(v) - \Psi^1(v)$, the claim follows from equations 2.19 and 2.20.

We now consider two possible scenarios depending upon the value of $(\alpha - \beta x - y)$. We show that in each case $\delta\mathcal{B} \geq \beta^k$. This, along with Claim 6, implies that $\delta\mathcal{B} \geq \textsc{Count}^1 - \textsc{Count}^0$. This proves Lemma 2.3.

1. Suppose that $(\alpha-\beta x-y) < 0$. From Claims 7, 9, 10 and equation 2.12, we derive:

$$\begin{aligned}
\epsilon \cdot \delta\mathcal{B} &= \sum_{u\in\mathcal{N}_v} \delta\Psi(u) + \sum_{u\in\mathcal{N}_v} \delta\Phi(u,v) + \Psi(v) \\
&\geq -(1+\epsilon)\cdot x \cdot \beta^k + (\alpha - x - y)\cdot \frac{\beta^{k+1}}{(\beta-1)} \\
&\geq -(1+\epsilon)\cdot \beta^k + (\alpha-1)\cdot \beta^{k+1}/(\beta-1) \\
&= \frac{\beta^k}{(\beta-1)} \cdot \{(\alpha-1)\cdot\beta - (1+\epsilon)(\beta-1)\} \\
&= (1+\epsilon)\cdot\beta^k \\
&\geq \epsilon\cdot\beta^k
\end{aligned}$$

   The last equality holds since $\alpha = 1 + 2\epsilon$ and $\beta = 1 + \epsilon$.

2. Suppose that $(\alpha-\beta x-y) \geq 0$. From Claims 7, 9, 10 and equation 2.12, we derive:

$$\begin{aligned}
\epsilon \cdot \delta\mathcal{B} &= \sum_{u\in\mathcal{N}_v} \delta\Psi(u) + \sum_{u\in\mathcal{N}_v} \delta\Phi(u,v) + \Psi(v) \\
&\geq -(1+\epsilon)\cdot x \cdot \beta^k + (\alpha - x - y)\cdot \frac{\beta^{k+1}}{(\beta-1)} \\
&\quad -(\alpha-\beta x-y)\cdot\beta^k/(\beta-1) \\
&= \frac{\beta^k}{(\beta-1)} \cdot \big\{(\alpha-x-y)\cdot\beta \\
&\quad -(1+\epsilon)\cdot x\cdot(\beta-1) - (\alpha-\beta x-y)\big\} \\
&= \frac{\beta^k}{(\beta-1)} \cdot \big\{(\alpha-x-y)\cdot(\beta-1) \\
&\quad -\epsilon\cdot x\cdot(\beta-1)\big\} \\
&\geq \frac{\beta^k}{(\beta-1)} \cdot \big\{(\alpha-1)(\beta-1) - \epsilon(\beta-1)\big\} \\
&= \epsilon\cdot\beta^k \quad (\text{since } \alpha = 1+2\epsilon, \beta = 1+\epsilon)
\end{aligned}$$

3 Dynamic Matching: Preliminaries

We are given an input graph $G = (V, E)$ that is being updated dynamically through a sequence of edge insertions/deletions. We want to maintain an approximately maximum matching in G. We will present two different algorithms for this problem. They are described in Sections 4 and 5. Both these algorithms, however, will use two key ideas.

1. It is easy to maintain a good approximate matching in a bounded degree graph (see Section 3.1).
2. Every graph contains a subgraph of bounded degree, called its *kernel*, that approximately preserves the size of the maximum matching (see Section 3.2).

In Section 3.3, we give a static algorithm for building a kernel in a graph. In Section 3.4, we present the data structures that will be used in Sections 4 and 5 for maintaining a kernel in a dynamic setting.

Query time. In this paper, all the data structures for dynamic matching explicitly maintain the set of matched edges. Accordingly, using appropriate pointers, we can support the following queries.

- Report the size of the matching M maintained by the data structure in $O(1)$ time.
- Report the edges in M in $O(1)$ time per edge.
- In $O(1)$ time, report whether or not a given edge (u, v) is part of the matching.
- In $O(1)$ time, report whether or not a given node u is matched in M, and if yes, in $O(1)$ time report the node it is matched to.

3.1 Maintaining an approximate matching in a bounded degree graph.

Maintaining a *maximal* matching in a graph with maximum degree δ in time $O(\delta)$ per deletion and $O(1)$ per edge insertion is straightforward: After an insertion check whether the inserted edge can be added to the matching. If a matched edge is deleted, check all neighbors of its endpoints to rematch them if possible. Following the approach from [12], we can show that in $O(\delta)$ worst-case update time we can even maintain a 3/2-*approximate* matching by guaranteeing that no augmenting path has a length less than 5. The basic idea is to keep for each node a list of its neighbors as well as a list of its unmatched neighbors. Due to space constraints, the proofs of the next two theorems appear in the full version of the paper.

THEOREM 3.1. *Consider a dynamic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, and suppose that the maximum degree of a node in $\mathcal{G}$ is always upper bounded by δ. There exists a data structure for maintaining a maximal matching M in $\mathcal{G}$ that handles each edge insertion in $O(1)$ time and each edge deletion in $O(\delta)$ time.*

THEOREM 3.2. *Consider a dynamic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, and suppose that the maximum degree of a node in $\mathcal{G}$ is always upper bounded by δ. There exists a data structure for maintaining a matching M in $\mathcal{G}$ such that (a) every augmenting path in G (w.r.t. M) has length at least five and (b) an edge insertion or deletion in $\mathcal{G}$ takes $O(\delta)$ worst-case time. Condition (a) implies that M is a 3/2-approximate maximum matching in G.*

3.2 The kernel and its properties.

In the input graph $G = (V, E)$, let $\mathcal{N}_v = \{u \in V : (u, v) \in E\}$ denote the set of *neighbors* of $v \in V$.

Consider a subgraph $\kappa(G) = (V, \kappa(E))$ with $\kappa(E) \subseteq E$. For all $v \in V$, define the set $\kappa(\mathcal{N}_v) = \{u \in \mathcal{N}_v : (u, v) \in \kappa(E)\}$. If $u \in \kappa(\mathcal{N}_v)$, then we say that u is a *friend* of v in $\kappa(G)$. Next, the set of nodes V is partitioned into two groups: *tight* and *slack*. We denote the set of tight (resp. slack) nodes by $\kappa_T(V)$ (resp. $\kappa_S(V)$). Thus, we have $V = \kappa_T(V) \cup \kappa_S(V)$ and $\kappa_T(V) \cap \kappa_S(V) = \emptyset$.

DEFINITION 2. *Fix any $c \geq 1$ and any $\epsilon \in [0, 1/3)$. The subgraph $\kappa(G)$ is an (ϵ, c)-kernel of G with respect to the partition $(\kappa_T(V), \kappa_S(V))$ iff it satisfies Invariants 4- 6.*

INVARIANT 4. *$|\kappa(\mathcal{N}_v)| \leq (1 + \epsilon)c$ for all $v \in V$, i.e., every node has at most $(1 + \epsilon)c$ friends.*

INVARIANT 5. *$|\kappa(\mathcal{N}_v)| \geq (1 - \epsilon)c$ for all $v \in \kappa_T(V)$, i.e., every tight node has at least $(1 - \epsilon)c$ friends.*

INVARIANT 6. *For all $u, v \in \kappa_S(V)$, 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 $\kappa(G)$.*

By Invariant 4, the maximum degree of a node in an (ϵ, c)-kernel is $O(c)$. In Theorems 3.3 and 3.4, we show that an (ϵ, c)-kernel $\kappa(G)$ is basically a subgraph of G with maximum degree $O(c)$ that approximately preserves the size of the maximum matching.

For ease of exposition, we often refer to a kernel $\kappa(G)$ without explicitly mentioning the underlying partition $(\kappa_T(V), \kappa_S(V))$. The proofs of Theorems 3.3, 3.4 appear in Sections 3.2.1 and 3.2.2 respectively.

THEOREM 3.3. *Let M be a maximal matching in an (ϵ, c)-kernel $\kappa(G)$. Then M is a $(4+6\epsilon)$-approximation to the maximum matching in G.*

THEOREM 3.4. *Let M be a matching in an (ϵ, c)-kernel $\kappa(G)$ such that every augmenting path in $\kappa(G)$ (w.r.t. M) has length at least five. Then M is a $(3+3\epsilon)$-approximation to the maximum matching in G.*

3.2.1 Proof of Theorem 3.3.

Let $V(M) = \{v \in V : (u, v) \in M \text{ for some } u \in \mathcal{N}_v\}$ be the set of nodes that are matched in M, and let $F_T = \kappa_T(V) \setminus V(M)$ be the subset of tight nodes in $\kappa(G)$ that are free in M.

LEMMA 3.1. *We have $|F_T| \leq (2 + 6\epsilon) \cdot |M|$.*

Proof. We will show that $|F_T| \leq (1+3\epsilon) \cdot |V(M)|$. Since $|V(M)| = 2 \cdot |M|$, the lemma follows.

We design a charging scheme where each node in F_T contributes one dollar to a *global fund*. So the total amount of money in this fund is equal to $|F_T|$ dollars. Below, we demonstrate how to transfer this fund to the nodes in $V(M)$ so that each node $x \in V(M)$ receives at most $(1 + 3\epsilon)$ dollars. This implies that $|F_T| \leq (1 + 3\epsilon) \cdot |V(M)|$.

Since M is a maximal matching in $\kappa(G)$, we must have $\kappa(\mathcal{N}_v) \subseteq V(M)$ for all $v \in F_T$. For each node $v \in F_T$, we distribute its one dollar equally among its friends, i.e., each node $x \in \kappa(\mathcal{N}_v)$ gets $1/|\kappa(\mathcal{N}_v)|$ dollars from v. Since $F_T \subseteq \kappa_T(V)$, Invariant 5 implies that $1/|\kappa(\mathcal{N}_v)| \leq 1/((1 - \epsilon)c)$ for all $v \in F_T$. In other words, a node in $V(M)$ receives at most $1/((1 - \epsilon)c)$ dollars from each of its friends under this money-transfer scheme. But, by Invariant 4, a node can have at most $(1 + \epsilon)c$ friends. So the total amount of money received by a node in $V(M)$ is at most $(1 + \epsilon)c/((1 - \epsilon)c) \leq (1 + 3\epsilon)$, for $\epsilon \in [0, 1/3)$.

To continue with the proof of Theorem 3.3, let $M^o \subseteq E$ be a maximum-cardinality matching in $G = (V, E)$. Define $M_1^o \subseteq M^o$ to be the subset of edges whose both endpoints are unmatched in M, and let $M_2^o = M^o \setminus M_1^o$.

Consider any edge $(u, v) \in M_1^o$. By definition, both the nodes u, v are free in M. Since M is a maximal matching in $\kappa(G)$, it follows that the edge (u, v) is not part of the kernel, i.e., $(u, v) \neq \kappa(E)$. Hence, Invariant 6 implies that either $u \notin \kappa_S(V)$ or $v \notin \kappa_S(V)$. Without any loss of generality, suppose that $u \notin \kappa_S(V)$. This means that the node u is tight, and, furthermore, it is free in M. We infer that every edge in M_1^o is incident to at least one node from F_T, where $F_T \subseteq \kappa_T(V)$ is the subset of tight nodes that are free in M. Accordingly, we have $|M_1^o| \leq |F_T|$. Combining this inequality with Lemma 3.1, we get:

$$|M_1^o| \leq (2 + 6\epsilon) \cdot |M| \tag{3.21}$$

Next, every edge in M_2^o has at least one endpoint that is matched in M. Thus, M is a maximal matching in the graph $G' = (V, M_2^o \cup M)$. Since M_2^o is also a matching in G', we get:

$$|M_2^o| \leq 2 \cdot |M| \tag{3.22}$$

The theorem follows if we add equations 3.21 and 3.22.

3.2.2 Proof of Theorem 3.4.

Let $V(M) = \{v \in V : (u, v) \in M \text{ for some } u \in \mathcal{N}_v\}$ be the set of nodes that are matched in M. For all nodes $v \in V(M)$, let $e_M(v)$ denote the edge in M that is incident to v. Let $F_T = \kappa_T(V) \setminus V(M)$ be the subset of tight nodes in $\kappa(G)$ that are free in M.

LEMMA 3.2. *For every edge $(u,v) \in M$, we have $|\,(\kappa(\mathcal{N}_u) \cap F_T) \cup (\kappa(\mathcal{N}_v) \cap F_T)\,| \leq (1+\epsilon)c$.*

Proof. Suppose that the lemma is false and we have $|\,(\kappa(\mathcal{N}_u) \cap F_T) \cup (\kappa(\mathcal{N}_v) \cap F_T)\,| > (1+\epsilon)c$ for some edge $(u,v) \in M$. As Invariant 4 guarantees that $|\kappa(\mathcal{N}_u)| \leq (1+\epsilon)c$ and $|\kappa(\mathcal{N}_v)| \leq (1+\epsilon)c$, there has to be a pair of distinct nodes $u', v' \in F_T$ such that $u' \in \kappa(\mathcal{N}_u)$ and $v' \in \kappa(\mathcal{N}_v)$. This means that the path (u', u, v, v') is an augmenting path in $\kappa(G)$ (w.r.t. M) and has length three. We reach a contradiction.

LEMMA 3.3. *We have $|F_T| \leq (1+3\epsilon) \cdot |M|$.*

Proof. We design a charging scheme where each node in F_T contributes one dollar to a *global fund*. So the total amount of money in this fund is equal to $|F_T|$ dollars. Below, we demonstrate how to transfer this fund to the edges in M so that each edge $e \in M$ receives at most $(1+3\epsilon)$ dollars. This implies that $|F_T| \leq (1+3\epsilon) \cdot |M|$.

Since M is a maximal matching in $\kappa(G)$, we must have $\kappa(\mathcal{N}_v) \subseteq V(M)$ for all $v \in F_T$. For each node $v \in F_T$, we distribute its one dollar equally among the matched edges incident to its friends, i.e., for each node $x \in \kappa(\mathcal{N}_v)$, the edge $e_M(x)$ gets $1/|\kappa(\mathcal{N}_v)|$ dollars from v. Since $F_T \subseteq \kappa_T(V)$, Invariant 5 implies that $1/|\kappa(\mathcal{N}_v)| \leq 1/\left((1-\epsilon)c\right)$ for all $v \in F_T$. In other words, an edge $(x,y) \in M$ receives at most $1/\left((1-\epsilon)c\right)$ dollars from each of the nodes $v \in (\kappa(\mathcal{N}_x) \cap F_T) \cup (\kappa(\mathcal{N}_y) \cap F_T)$ under this money-transfer scheme. Hence, Lemma 3.2 implies that the total amount of money received by an edge $(x,y) \in M$ is at most $(1+\epsilon)c/((1-\epsilon)c) \leq (1+3\epsilon)$, for $\epsilon \in [0, 1/3)$.

To continue with the proof of Theorem 3.4, define $M^o \subseteq E$ to be a maximum-cardinality matching in $G = (V, E)$. Furthermore, let $M_1^o \subseteq M^o$ be the subset of edges whose both endpoints are unmatched in M, and let $M_2^o = M^o \setminus M_1^o$.

Consider any edge $(u,v) \in M_1^o$. By definition, both the nodes u, v are free in M. Since M is a maximal matching in $\kappa(G)$, it follows that the edge (u,v) is not part of the kernel, i.e., $(u,v) \neq \kappa(E)$. Hence, Invariant 6 implies that either $u \notin \kappa_S(V)$ or $v \notin \kappa_S(V)$. Without any loss of generality, suppose that $u \notin \kappa_S(V)$. This means that the node u is tight, and, furthermore, it is free in M. We infer that every edge in M_1^o is incident to at least one node from F_T. Accordingly, we have $|M_1^o| \leq |F_T|$. Combining this inequality with Lemma 3.3, we get:

$$|M_1^o| \leq (1+3\epsilon) \cdot |M| \tag{3.23}$$

Next, every edge in M_2^o has at least one endpoint that is matched in M. Thus, M is a maximal matching in the graph $G' = (V, M_2^o \cup M)$. Since M_2^o is also a matching in G', we get:

$$|M_2^o| \leq 2 \cdot |M| \tag{3.24}$$

The theorem follows if we add equations 3.23 and 3.24.

3.3 An algorithm for building a kernel in a static graph.

We present a linear-time algorithm for constructing a $(0,c)$-kernel $\kappa(G)$ of a static graph $G = (V, E)$.

THEOREM 3.5. *We have an algorithm for computing a $(0,c)$-kernel $\kappa(G) = (V, \kappa(E))$ of a graph $G = (V, E)$. The kernel returned by the algorithm has the added property that tight nodes have exactly c friends and slack nodes have less than c friends. For every $c \geq 1$, the algorithm runs in $O(|E|)$ time.*

Proof. Initially, each node in V has zero friends, and the edge-set $\kappa(E)$ is empty. We then execute the FOR loop stated below.

- FOR ALL $(u,v) \in E$
  - IF $|\kappa(\mathcal{N}_u)| < c$ and $|\kappa(\mathcal{N}_v)| < c$, THEN
    - Set $\kappa(\mathcal{N}_v) \leftarrow \kappa(\mathcal{N}_v) \cup \{u\}$ and $\kappa(\mathcal{N}_u) \leftarrow \kappa(\mathcal{N}_u) \cup \{v\}$.
    - Set $\kappa(E) \leftarrow \kappa(E) \cup \{(u,v)\}$.

Consider the $\kappa(G) = (V, \kappa(E))$ we get at the end of the FOR loop. Clearly, in $\kappa(G)$ every node has at most c friends, i.e., $|\kappa(\mathcal{N}_v)| \leq c$ for all $v \in V$. Furthermore, for all edges $(u,v) \in E$, if $|\kappa(\mathcal{N}_u)| < c$ and $|\kappa(\mathcal{N}_v)| < c$, then it is guaranteed that $(u,v) \in \kappa(E)$. Thus, if we define $\kappa_T(V) = \{v \in V : |\kappa(\mathcal{N}_v)| = c\}$ and $\kappa_S(V) = \{v \in V : |\kappa(\mathcal{N}_v)| < c\}$, then $\kappa(G)$ becomes a $(0,c)$-kernel of G. The whole procedure can be implemented in $O(|E|)$ time.

3.4 Data structures for representing a kernel in a dynamic graph.

In the graph G, the set of neighbors of a node v is stored in the form of a linked list NEIGHBORS(v), which is part of an adjacency-list data structure. Further, each node v maintains the following information.

- A bit TYPE(v) indicating if the node is tight or slack.

$$\text{TYPE}(v) = \begin{cases} \text{tight} & \text{if } v \in \kappa_T(V); \\ \text{slack} & \text{if } v \in \kappa_S(V). \end{cases}$$

- The set of nodes $\kappa(\mathcal{N}_v)$, in the form of a doubly linked list FRIENDS(v).

- A counter #FRIENDS$(v) = |\kappa(\mathcal{N}_v)|$ that keeps track of the number of friends of the node.

Furthermore, we store at each edge $(u, v) \in \kappa(E)$ two pointers, corresponding to the two occurrences of edge (u, v) in the linked lists FRIENDS(u) and FRIENDS(v). In particular, we denote by POINTER$[u, v]$ (resp. POINTER$[v, u]$) the pointer to the position of u (resp. v) in the list FRIENDS(v) (resp. FRIENDS(u)). Using those pointers, we can insert an edge into $\kappa(G)$ or delete an edge from $\kappa(G)$ in $O(1)$ time.

4 $(3 + \epsilon)$-approximate matching in $O(m^{1/3}/\epsilon^2)$ amortized update time

Fix any constant $\epsilon \in (0, 1/3)$. In this section, we present an algorithm for maintaining a $(3 + \epsilon)$-approximate matching M in a graph $G = (V, E)$ undergoing a sequence of edge insertions/deletions. It requires $O(m^{1/3}/\epsilon^2)$ amortized update time.

4.1 Overview of our approach.

The main idea is to partition the sequence of updates (edge insertions/deletions) in G into *phases*. Each phase lasts for $\epsilon^2 c^2/2$ consecutive updates in G. Let $G_{i,t}$ denote the state of G just after the t^{th} update in phase i. The initial state of the graph, before it starts changing, is given by $G_{1,0}$. We reach the graph $G_{i,t}$ from $G_{i,0}$ after a sequence of $(i-1) \cdot (\epsilon^2 c^2/2) + t$ updates in G.

For the rest of this section, we focus on describing our algorithm for any given phase $i \geq 1$. We define $m \leftarrow |E_{i,0}|$ to be the number of edges in the input graph in the beginning of the phase, and set $c \leftarrow m^{1/3}$. Since the phase lasts for only $O(\epsilon^2 m^{2/3})$ updates in G, it follows that $|E_{i,t}| = O(m)$ for all $0 \leq t \leq \epsilon^2 c^2/2$. During the phase, we maintain a $(3 + \epsilon)$-approximate matching in G as described below.

Just before phase i begins, we build a $(0, c)$-kernel $\kappa(G)$ on $G = G_{i,0}$ as per Theorem 3.5. This takes $O(m)$ time. Next, we compute a 3/2-approximate maximum matching $M = M_{i,0}$ on $\kappa(G_{i,0})$ by ensuring that every augmenting path in $\kappa(G_{i,0})$ (w.r.t. $M_{i,0}$) has length at least five. This also takes $O(m)$ time, and concludes the preprocessing step.

After each update in G during phase i, we first modify the kernel $\kappa(G)$ using the algorithm in Section 4.2, and subsequently we modify the matching M in $\kappa(G)$ using Theorem 3.2. Theorem 4.2 guarantees that the graph $\kappa(G)$ remains an (ϵ, c)-kernel of G throughout the phase. Hence, by Theorem 3.4, the matching M remains a $(3 + \epsilon)$-approximation to the maximum matching in G.

The idea behind the algorithm in Section 4.2 is simple. In the beginning of the phase, Theorem 3.5 guarantees that every tight node has exactly c friends. During the phase, whenever the number of friends of a tight node v drops below $(1 - \epsilon)c$, it scans through its first c neighbors in $\mathcal{N}_v$, and keeps making friends out of them until $|\kappa(\mathcal{N}_v)|$ becomes equal to c. This procedure is implemented in the subroutine REFILL(v) (see Figure 2). A potential problem with this approach is that after a while a node v may have more than $(1+\epsilon)c$ friends due to the repeated invocations of the subroutine REFILL(u), for $u \in \mathcal{N}_v$. We show that this event can be ruled out by ending the current phase after $\epsilon^2 c^2/2$ updates in G.

Bounding the update time.

- *Preprocessing.*

  The preprocessing in the beginning of the phase takes $O(m)$ time. Since the phase lasts for $\epsilon^2 c^2/2$ updates, we get an amortized bound of $O(m/(\epsilon^2 c^2)) = O(m^{1/3}/\epsilon^2)$.

- *Maintaining the kernel $\kappa(G)$.*

  By Theorem 4.2, the kernel $\kappa(G)$ can be modified after each update in G in $O(c) = O(m^{1/3})$ time.

- *Maintaining the matching M in $\kappa(G)$.*

  By Theorem 4.2, the number of updates made into $\kappa(G)$ during the entire phase is $O(\epsilon^2 c^2)$. Since the maximum degree of a node in $\kappa(G)$ is $O(c)$, modifying the matching M after each update in $\kappa(G)$ requires $O(c)$ time (see Theorem 3.2). Thus, the total time spent during the phase in maintaining the matching M is $O(\epsilon^2 c^2) \cdot O(c) = O(\epsilon^2 c^3)$. Since the phase lasts for $\epsilon^2 c^2/2$ updates in G, we get an amortized bound of $O(c) = O(m^{1/3})$.

We summarize the main result of this section in the theorem below.

THEOREM 4.1. *In a dynamic graph $G = (V, E)$, we can maintain a $(3 + \epsilon)$-approximate matching $M \subseteq E$ in $O(m^{1/3}/\epsilon^2)$ amortized update time.*

4.2 Algorithms for maintaining the kernel during a phase.

We present our algorithm for maintaining an (ϵ, c)-kernel of the graph during a phase.

THEOREM 4.2. *Suppose that we are given a $(0, c)$-kernel $\kappa(G)$ of G in the beginning of a phase. Then we have an algorithm for modifying $\kappa(G)$ after each update (edge insertion/deletion) in G such that:*

1. *Each update in G is handled in $O(c)$ worst-case time.*

2. *At most $O(\epsilon^2 c^2)$ updates are made into $\kappa(G)$ during the entire phase.*

3. *Throughout the duration of the phase, $\kappa(G)$ remains an (ϵ, c)-kernel of G.*

The rest of Section 4.2 is organized as follows. In Section 4.2.1 (resp. 4.2.2), we present our algorithm for handling an edge insertion (resp. edge deletion) in G during the phase. In Section 4.2.3, we prove that the algorithm satisfies the properties stated in Theorem 4.2.

4.2.1 Handling an edge insertion in G during the phase.

Suppose that an edge (u, v) is inserted into the graph $G = (V, E)$. To handle this edge insertion, we first update the lists NEIGHBORS(u) and NEIGHBORS(v), and then process the edge as follows.

- **Case 1:** Either TYPE(u) = tight or TYPE(v) = tight.

  We do nothing and conclude the procedure.

- **Case 2:** Both TYPE(u) = slack and TYPE(v) = slack.

  - **Case 2a:** Either #FRIENDS$(u) \geq c$ or #FRIENDS$(v) \geq c$.

    If #FRIENDS$(u) \geq c$, then we set TYPE$(u) \leftarrow$ tight. Next, if #FRIENDS$(v) \geq c$, then we set TYPE$(v) \leftarrow$ tight.

  - **Case 2b:** Both #FRIENDS$(u) < c$ and #FRIENDS$(v) < c$.

    We add the edge (u, v) to the kernel $\kappa(G)$ and make u, v friends of each other. Specifically, we add the node u to the list FRIENDS(v) and v to the list FRIENDS(u), update the pointers POINTER$[u, v]$ and POINTER$[v, u]$ accordingly, and increment each of the counters #FRIENDS(u), #FRIENDS(v) by one unit.

LEMMA 4.1. *Suppose that an edge is inserted into the graph G and we run the procedure described above.*

1. *This causes at most one edge insertion into $\kappa(G)$ and at most one edge deletion from $\kappa(G)$.*

2. *The procedure runs in $O(1)$ time in the worst case.*

4.2.2 Handling an edge deletion in G during the phase.

Suppose that an edge (u, v) is deleted from the graph $G = (V, E)$. To handle this edge deletion, we first update the lists NEIGHBORS(u) and NEIGHBORS(v), and then proceed as follows.

We check if the edge (u, v) was part of the kernel $\kappa(G)$, and, if yes, then we delete (u, v) from $\kappa(G)$. Specifically, we delete u from FRIENDS(v), v from FRIENDS(u) (using POINTER$[u, v]$ and POINTER$[v, u]$), and decrement each of the counters #FRIENDS(u), #FRIENDS(v) by one unit.

We then process the nodes u and v one after another. Below, we describe only the procedure that runs on the node u. The procedure for the node v is exactly the same.

- **Case 1:** TYPE(u) = tight. Here, we check if the number of friends of u has dropped below the prescribed limit due to the edge deletion, and, accordingly, we consider two possible sub-cases.

  - **Case 1a:** #FRIENDS$(u) < (1 - \epsilon)c$. Here, we call the subroutine REFILL(u) as described in Figure 2.

  - **Case 1b:** #FRIENDS$(u) \geq (1 - \epsilon)c$. In this case, we do nothing and conclude the procedure.

- **Case 2:** TYPE(u) = slack. In this case, we do nothing and conclude the procedure.

LEMMA 4.2. *Suppose that an edge is deleted from the graph G and we run the procedure described above.*

1. *This causes at most $O(\epsilon c)$ edge insertions into $\kappa(G)$ and at most one edge deletion from $\kappa(G)$.*

2. *The procedure runs in $O(c)$ time in the worst case.*

4.2.3 Proof of Theorem 4.2.

The first part of the theorem immediately follows from Lemmas 4.1, 4.2. We now focus on the second part. Note that at most one edge is deleted from $\kappa(G)$ after an update (edge insertion or deletion) in G. Since the phase lasts for $\epsilon^2 c^2/2$ updates in G, at most $\epsilon^2 c^2/2$ edge deletions occur in $\kappa(G)$ during the phase. To complete the proof, we will show that the corresponding number of edge insertions in $\kappa(G)$ is also $O(\epsilon^2 c^2)$.

For an edge insertion in G, at most one edge is inserted into $\kappa(G)$. For an edge deletion in G, there can be $O(\epsilon c)$ edge insertions in $\kappa(G)$, *but only if* the subroutine REFILL(.) is called. Lemma 4.3 shows that at most ϵc calls are made to the subroutine REFILL(.) during the phase. This implies that at most $O(\epsilon^2 c^2)$ edge insertions occur in $\kappa(G)$ during the phase, which proves the second part of the theorem.

LEMMA 4.3. *The subroutine* REFILL(.) *is called at most ϵc times during the phase.*

IF the list NEIGHBORS(u) is empty, THEN
  Set TYPE(u) $\leftarrow$ slack.
  RETURN.
Let x be the first node in the list NEIGHBORS(u).
WHILE (#FRIENDS(u) $< c$)
  IF $x \notin$ FRIENDS(u), THEN
    Add x to FRIENDS(u) and u to FRIENDS(x), so that the edge (u, x) becomes part of the kernel $\kappa(G) = (V, \kappa(E))$.
    Increment each of the counters #FRIENDS(u), #FRIENDS(v) by one unit.
    Update the pointers POINTER$[u, x]$ and POINTER$[x, u]$.
  IF x is the last node in NEIGHBORS(u), THEN
    exit the WHILE loop.
  ELSE
    Let y be the node that succeeds x in the list NEIGHBORS(u).
    Set $x \leftarrow y$.
IF #FRIENDS(u) $< c$, THEN
  Set TYPE(u) $\leftarrow$ slack.

Figure 2: REFILL(u).

Proof. When the phase begins, every tight node has exactly c friends (see Theorem 3.5), and during the phase, the status of a node u is changed from slack to tight only when $|\kappa(\mathcal{N}_u)|$ becomes (weakly) greater than c. On the other hand, the subroutine REFILL(u) is called only when u is tight and $|\kappa(\mathcal{N}_u)|$ falls below $(1-\epsilon)c$. Thus, each call to REFILL(.) corresponds to a scenario where a tight node has lost at least ϵc friends. Each edge deletion in G leads to at most two such losses (one for each of the endpoints), whereas an edge insertion in G leads to no such event. Since the phase lasts for $\epsilon^2 c^2/2$ edge insertions/deletions in G, a counting argument shows that it can lead to at most $(\epsilon^2 c^2/2) \cdot 2/(\epsilon c) = \epsilon c$ calls to REFILL(.).

It remains to prove the final part of the theorem, which states that the algorithm maintains an (ϵ, c)-kernel. Specifically, we will show that throughout the duration of the phase, the subgraph $\kappa(G)$ maintained by the algorithm satisfies Invariants 4–6.

LEMMA 4.4. *Suppose that $\kappa(G)$ satisfies Invariants 5, 6 before an edge update in G. Then these invariants continue to hold even after we modify $\kappa(G)$ as per the procedure in Section 4.2.1 (resp. Section 4.2.2).*

Proof. Follows from the descriptions of the procedures in Sections 4.2.1 and 4.2.2.

Recall that in the beginning of the phase, the graph $\kappa(G)$ is a $(0, c)$-kernel of G. Since every $(0, c)$-kernel is also an (ϵ, c)-kernel, we repeatedly invoke Lemma 4.4 after each update in G, and conclude that $\kappa(G)$ satisfies Invariants 5, 6 throughout the duration of the phase. For the rest of this section, we focus on proving the remaining Invariant 4.

Fix any node $v \in V$. When the phase begins, the subgraph $\kappa(G)$ is a $(0, c)$-kernel of G, so that we have $|\kappa(\mathcal{N}_v)| \leq c$. During the phase, the node v can get new friends under two possible situations.

1. An edge incident to v has just been inserted into (resp. deleted from) the graph G, and the procedure in Section 5.2.3 (resp. the subroutine REFILL(v)) is going to be called.

2. The subroutine REFILL(u) is going to be called for some $u \in \mathcal{N}_v$.

If we are in situation (1) and the node v already has at least c friends, then the procedure under consideration will not end up adding any more node to $\kappa(\mathcal{N}_v)$. Thus, it suffices to show that the node v can get at most ϵc new friends during the phase under situation (2). Note that each call to REFILL(u), $u \neq v$, creates at most one new friend for v. Accordingly, if we show that the subroutine REFILL(.) is called at most ϵc times during the entire phase, then this will suffice to conclude the proof of Theorem 4.2. But this has already been done in Lemma 4.3.

5 $(4+\epsilon)$-approximate matching in $O(m^{1/3}/\epsilon^2)$ worst-case update time

Fix any $\epsilon \in (0, 1/6)$. We present an algorithm for maintaining a $(4+\epsilon)$-approximate matching M in a dynamic graph $G = (V, E)$ with $O(m^{1/3}/\epsilon^2)$ worst-case update time.

5.1 Overview of our approach.

As in Section 4.1, we partition the sequence of updates (edge insertions/deletions) in G into *phases*. Each phase lasts for $\epsilon^2 c^2/2$ consecutive updates in G. Let $G_{i,t}$ denote the state of G just after the t^{th} update in phase i. The initial state of the graph, before it starts changing, is given by $G_{1,0}$. Thus, we reach the graph $G_{i,t}$ from $G_{i,0}$ after a sequence of $(i-1) \cdot (\epsilon^2 c^2/2) + t$ updates in G.

For the rest of this section, we focus on describing our algorithm for any given phase $i \geq 1$. We define $m \leftarrow |E_{i,0}|$ to be the number of edges in the input graph in the beginning of the phase, and set $c \leftarrow m^{1/3}$. Since the phase lasts for only $O(\epsilon^2 m^{2/3})$ updates in G, it follows that $|E_{i,t}| = O(m)$ for all $0 \leq t \leq \epsilon^2 c^2/2$.

Till now, the description has been the same as in Section 4.1. From this point onwards, however, we need to modify the framework in Section 4.1 to get a *worst-case bound* on the update time.

Preprocessing. If $i = 1$, then we build a $(0, c)$-kernel $\kappa(G_{1,0})$ in the beginning of the phase as per Theorem 3.5. Next, we build a maximal matching $M_{1,0}$ in $\kappa(G_{1,0})$. Overall, this takes $O(|V|+|E_{1,0}|)$ time.

Algorithm for each phase. The algorithm of the previous section rebuilds the kernel and a corresponding matching from scratch at the beginning of each phase and we amortize the cost of the rebuild over the phase. To achieve a worst-case running time we do this rebuilding "in the background" during the phase. This means that at the beginning of a phase we start with an empty graph G^* and an empty kernel and insert $O(m/(\epsilon^2 c^2))$ edges into G^* and its kernel during each update in G. Thus, edge insertions into G^* need to be handled in constant time. We can handle "bunch updates" into G^* to build the kernel efficiently, but we cannot efficiently update a 3/2-approximate matching, in the kernel as each edge insertion takes $O(c)$ time (Theorem 3.2). However, we can update a *maximal* matching, i.e., a 2-approximate matching in constant time per edge insertions (Theorem 3.1). Thus, we run the 2-approximation algorithm on the kernel instead of the 3/2-approximation algorithm, leading to a $(4 + \epsilon)$ overall approximation. Additionally, we need to perform the updates of the current phase in G^*. For that we use basically the same algorithm as in Section 4.2. As a result at the end of a phase $G^* = G$ and the kernel that we built for G^* is only an (ϵ, c) kernel of G, i.e., we have to start the next phase with a (ϵ, c)-kernel instead of a $(0, c)$-kernel. This, however, degrades the approximation ratio only by an additional factor of ϵ.

To summarize our algorithm for phase i has two *components*.

- *Dealing with the current phase.* Just before the start of phase i, an (ϵ, c)-kernel $\kappa(G_{i,0})$ and a maximal matching $M_{i,0}$ in $\kappa(G_{i,0})$ are handed over to us. Then, as G keeps changing, we keep modifying the subgraph $\kappa(G)$ and the matching M. Till the end of phase i, we ensure that $\kappa(G)$ remains a $(2\epsilon, c)$-kernel of G and that M remains a maximal matching in $\kappa(G)$. Hence, by Theorem 3.3, the matching M gives a $(4 + \epsilon)$-approximation[2] to the maximum matching in G throughout the duration of phase i.
- *Preparing for the next phase.* We build a new (ϵ, c)-kernel (and a maximal matching in it) for $G_{i+1,0}$ in the background. They are handed over to the algorithm for phase $(i + 1)$ at the start of phase $(i + 1)$.

We elaborate upon each of these components in more details in Section 5.2 (see Lemmas 5.1, 5.2). For maintaining a kernel in a dynamic graph, both these components use a procedure that is essentially the one described in Section 4.2. However, a more fine tuned analysis of the procedure is necessary in order get worst-case update time.

Finally, Theorem 3.3 and Lemmas 5.1, 5.2 give us the desired guarantee.

THEOREM 5.1. *We can maintain a $(4+\epsilon)$-approximate matching in a dynamic graph $G = (V, E)$ in $O(m^{1/3}/\epsilon^2)$ worst-case update time.*

5.2 Algorithm for maintaining and building the kernel during each phase.

In Subsections 5.2.3 and 5.2.4 we present an algorithm for maintaining a kernel in a fully dynamic graph. It is basically a relaxation of the algorithm from Subsection 4.2 and is based on the concept of an "epoch".

DEFINITION 3. *For $c \geq 1$, $\lambda \in (0, 1)$, a sequence of updates in a graph is called a* (λ, c)-epoch *iff it contains at most $\lambda^2 c^2/2$ edge deletions (which can be arbitrarily interspersed with any number of edge insertions).*

The resulting algorithm is summarized in the theorem below.

THEOREM 5.2. *Fix any $c \geq 1$ and $\epsilon, \lambda > 0$ with $(\epsilon + \lambda) < 1/3$. Consider any (λ, c)-epoch in the input graph $G = (V, E)$, and suppose that we are given an (ϵ, c)-kernel $\kappa(G)$ in the beginning of the epoch. Then we have an algorithm for updating $\kappa(G)$ after each update in G. The algorithm satisfies three properties.*

1. *An edge insertion in G is handled in $O(1)$ worst-case time, and this leads to at most one edge insertion in $\kappa(G)$ and zero edge deletion in $\kappa(G)$.*
2. *An edge deletion in G is handled in $O(c)$ worst-case time, and this leads to at most $O(\lambda c)$ edge insertions in $\kappa(G)$ and at most one edge deletion in $\kappa(G)$.*
3. *Throughout the duration of the epoch, $\kappa(G)$ remains a $(\lambda + \epsilon, c)$-kernel of G.*

[2]To be precise, the theorem shows a $(4 + 12\epsilon)$-approximation but running the algorithm with $\epsilon' = \epsilon/12$ results in a $(4 + \epsilon)$-approximation.

We use the kernel update algorithm in the two components of a phase, which we describe next in detail. Recall that $G_{i,t}$ denotes the state of G just after the t^{th} update in phase $i \geq 1$, $m \leftarrow |E_{i,0}|$, and $c \leftarrow m^{1/3}$.

5.2.1 Dealing with the current phase In the beginning of phase i, we are given an (ϵ, c)-kernel $\kappa(G_{i,0})$ and a maximal matching $M_{i,0}$ in $\kappa(G_{i,0})$. Since a phase lasts for $\epsilon^2 c^2/2$ edge updates in G, we treat phase i as an (ϵ, c)-epoch (see Definition 3). Thus, after each edge insertion/deletion in G during phase i, we modify $\kappa(G)$ as per Theorem 5.2. This ensures that $\kappa(G)$ remains an $(2\epsilon, c)$-kernel till the end of phase i. After each update in $\kappa(G)$, we modify the matching M using Theorem 3.1 so as to ensure that M remains a maximal matching in $\kappa(G)$.

We show that this procedure requires $O(c)$ time in the worst case to handle an update in G.

- *Case 1: An edge is inserted into the graph G.* By Theorem 5.2, in this case updating the kernel $\kappa(G)$ requires $O(c)$ time, and this results in at most one edge insertion into $\kappa(G)$ and zero edge deletion from $\kappa(G)$. By Theorem 3.1, updating the matching M requires $O(1)$ time.
- *Case 2: An edge is deleted from the graph G.* By Theorem 5.2, in this case updating the kernel $\kappa(G)$ requires $O(c)$ time, and this results in $O(\epsilon c)$ edge insertions into $\kappa(G)$ and at most one edge deletion from $\kappa(G)$. Since the maximum degree of a node in $\kappa(G)$ is $O(c)$, updating the matching M requires $O(\epsilon c) + O(c) = O(c)$ time (see Theorem 3.1).

LEMMA 5.1. *Suppose that when phase i begins, we are given an (ϵ, c)-kernel $\kappa(G)$ of G and a maximal matching M in $\kappa(G)$. After each update in G during phase i, we can modify $\kappa(G)$ and M in $O(c)$ time. Till the end of phase i, $\kappa(G)$ remains an $(2\epsilon, c)$-kernel of G and M remains a maximal matching in $\kappa(G)$.*

5.2.2 Preparing for the next phase.

Before phase i begins, in $O(1)$ time we initialize a graph $G^* = (V, E^*)$ with $E^* = \emptyset$, a $(0, c)$-kernel $\kappa(G^*) = (V, \kappa(E^*))$, and a maximal matching $M^* = \emptyset$ in $\kappa(G^*)$. To achieve constant time we use an uninitialized array for the TYPE-bit and assume that every node for which the bit is not initialized is of type slack.

After each update in G in phase i, we call UPDATE-G^* *(Figure 3). This ensures the following properties.*

- (P1) Each edge insertion in G leads to at most $O(m/(\epsilon^2 c^2))$ edge insertions in G^*.
- (P2) Each edge deletion in G leads to at most one edge deletion in G^* and at most $O(m/(\epsilon^2 c^2))$ edge insertions in G^*.
- (P3) After each edge insertion/deletion in G, changing the graph G^* requires $O(m/(\epsilon^2 c^2))$ time.
- (P4) We always have $E^* \subseteq E$. Furthermore, at the end of phase i, we have $E^* = E$. This holds since m is the size of E in the beginning of phase i, and since the phase lasts for $\epsilon^2 c^2/2$ updates in G.

IF an edge (u, v) is inserted into G, THEN
    $E^* \leftarrow E^* \cup \{(u, v)\}$.
ELSE IF an edge (u, v) is deleted from G, THEN
    $E^* \leftarrow E^* \setminus \{(u, v)\}$.
FOR $i = 1$ to $2m/(\epsilon^2 c^2)$
    IF $E \setminus E^* \neq \emptyset$, THEN
    Pick any edge $e \in E \setminus E^*$.
    $E^* \leftarrow E^* \cup \{e\}$.

Figure 3: UPDATE-G^*.

Now, consider the sequence of updates in G^* that take place during phase i. (P1) and (P2) guarantee that this sequence contains at most $\epsilon^2 c^2/2$ edge deletions. So this sequence can be treated as an (ϵ, c)-epoch in G^* (see Definition 3).

Hence, after each update in G^*, we modify the subgraph $\kappa(G^*)$ using Theorem 5.2, and subsequently, we modify the maximal matching M^* in $\kappa(G^*)$ using Theorem 3.1.

Since $\kappa(G^*)$ was a $(0, c)$-kernel of G^* in the beginning of the epoch, we get the following guarantee.

- (P5) Throughout the duration of phase i, the graph $\kappa(G^*)$ is an (ϵ, c)-kernel of $G^* = (V, E^*)$ and M^* is a maximal matching in $\kappa(G^*)$.

By (P1) and (P2), an update in G leads to $O(m/(\epsilon^2 c^2))$ edge insertions and at most one edge deletion in G^*. Each of these edge insertions in G^* further leads to at most one edge insertion in $\kappa(G^*)$, while the potential edge deletion in G^* leads to at most one edge deletion and $O(\epsilon c)$ edge insertions in $\kappa(G^*)$ (see Theorem 5.2). To summarize, an update in G leads to the following updates in G^* and $\kappa(G^*)$.

- At most one edge deletion and $O(m/(\epsilon^2 c^2))$ edge insertions in G^*.
- At most one edge deletion and $O(\epsilon c + m/(\epsilon^2 c^2))$ edge insertions in $\kappa(G^*)$.

Thus, by Theorems 5.2 and 3.1, we get the following bound on the update time

- (P6) After each edge update in G, the graph $\kappa(G^*)$ and the maximal matching M^* in $\kappa(G^*)$ can be modified in $O(c) + O(\epsilon c + m/(\epsilon^2 c^2)) = O(c + m/(\epsilon^2 c^2))$ time.

Using (P3) and (P6), we derive that after each update in G, the time required to modify the structures $G^*, \kappa(G^*)$ and M^* is $O(c + m/(\epsilon^2 c^2)) = O(m^{1/3}/\epsilon^2)$ in the worst case. By (P4) and (P5), the graph G^* becomes identical with G at the end of phase i, and at this point $\kappa(G^*)$ becomes an (ϵ, c)-kernel of $G_{i+1,0}$.

LEMMA 5.2. *Starting from the beginning of phase i, we can run an algorithm with the following properties.*

- *After each update in G during phase i, it performs $O(m^{1/3}/\epsilon^2)$ units of computation.*
- *At the end of phase i, it returns an (ϵ, c)-kernel of $G_{i+1,0}$ and a maximal matching in this kernel.*

Theorem 5.1 now follows from Theorem 3.3 and Lemmas 5.1, 5.2.

5.2.3 Handling an edge insertion into G during the epoch.

Suppose that an edge (u, v) is inserted into the graph $G = (V, E)$. To handle this edge insertion, we first update the lists NEIGHBORS(u) and NEIGHBORS(v), and then process the edge as follows.

- **Case 1:** Either TYPE(u) = tight or TYPE(v) = tight.

  We do nothing and conclude the procedure.

- **Case 2:** Both TYPE(u) = slack and TYPE(v) = slack.

  - **Case 2a:** Either #FRIENDS(u) $\geq (1-\epsilon)c$ or #FRIENDS(v) $\geq (1-\epsilon)c$.

    If #FRIENDS(u) $\geq (1-\epsilon)c$, then we set TYPE(u) $\leftarrow$ tight. Next, if #FRIENDS(v) $\geq (1-\epsilon)c$, then we set TYPE(v) $\leftarrow$ tight.

  - **Case 2b:** Both #FRIENDS(u) $< (1-\epsilon)c$ and #FRIENDS(v) $< (1-\epsilon)c$.

    We add the edge (u, v) to the kernel $\kappa(G)$. Specifically, we add u to the list FRIENDS(v) and v to the list FRIENDS(u), update the pointers POINTER$[u, v]$ and POINTER$[v, u]$ accordingly, and increment each of the counters #FRIENDS(u), #FRIENDS(v) by one unit.

LEMMA 5.3. *Suppose that an edge is inserted into the graph G and we run the procedure described above.*

1. *This causes at most one edge insertion into $\kappa(G)$ and at most one edge deletion from $\kappa(G)$.*
2. *The procedure runs in $O(1)$ time in the worst-case.*

5.2.4 Handling an edge deletion in G during the epoch.

Suppose that an edge (u, v) gets deleted from the graph $G = (V, E)$. To handle this edge deletion, we first consider the adjacency-list data structure for G, and update the lists NEIGHBORS(u) and NEIGHBORS(v).

We check if the edge (u, v) was part of the kernel $\kappa(G)$, and, if yes, then we delete (u, v) from $\kappa(G)$. Specifically, we delete u from FRIENDS(v) and v from FRIENDS(u) (using the pointers POINTER$[u, v]$, POINTER$[v, u]$), and decrement each of the counters #FRIENDS(u), #FRIENDS(v) by one unit.

We then process the nodes u and v one after another. Below, we describe only the procedure that runs on the node u. The procedure for the node v is exactly the same.

- **Case 1:** TYPE(u) = tight. Here, we check if the number of friends of u has dropped below the prescribed limit due to the edge deletion, and, accordingly, we consider two possible sub-cases.
  - **Case 1a:** #FRIENDS(u) $< (1-\lambda-\epsilon)c$. Here, we call the subroutine REFILL-NOW(u) (see Figure 4).
  - **Case 1b:** #FRIENDS(u) $\geq (1-\lambda-\epsilon)c$. In this case, we do nothing and conclude the procedure.
- **Case 2:** TYPE(u) = slack. In this case, we do nothing and conclude the procedure.

LEMMA 5.4. *Suppose that an edge is deleted from the graph G and we run the procedure described above.*

1. *This causes at most $O(\lambda c)$ edge insertions into $\kappa(G)$ and at most one edge deletion from $\kappa(G)$.*
2. *The procedure runs in $O(c)$ time in the worst-case.*

5.2.5 Proof of Theorem 5.2

The first and the second parts of the theorem follows from Lemma 5.3 and Lemma 5.4 respectively. It remains to show that the algorithm maintains a $(\lambda+\epsilon, c)$-kernel. Specifically, we will show that throughout the duration of the (λ, c)-epoch, the subgraph $\kappa(G)$ maintained by the algorithm continues to satisfy Invariants 4–6 (replacing ϵ by $\lambda + \epsilon$ in Invariants 4, 5).

IF the list NEIGHBORS(u) is empty, THEN
  Set TYPE(u) $\leftarrow$ slack.
  RETURN.
Let x be the first node in the list NEIGHBORS(u).
WHILE (#FRIENDS(u) $< (1-\epsilon)c$)
  IF $x \notin$ FRIENDS(u), THEN
    Add x to FRIENDS(u) and u to FRIENDS(x), so that the edge (u,x) becomes part of the kernel $\kappa(G) = (V, \kappa(E))$.
    Increment each of the counters #FRIENDS(u), #FRIENDS(x) by one unit.
    Update the pointers POINTER$[u,x]$, POINTER$[x,u]$.
  IF x is the last node in NEIGHBORS(u), THEN
    exit the WHILE loop.
  ELSE
    Let y be the node that succeeds x in the list NEIGHBORS(u).
    Set $x \leftarrow y$.
IF #FRIENDS(u) $< (1-\epsilon)c$, THEN
  Set TYPE(u) $\leftarrow$ slack.

Figure 4: REFILL-NOW(u).

LEMMA 5.5. *Suppose that $\kappa(G)$ satisfies two conditions before an edge insertion (resp. deletion) in G.*

1. *$|\kappa(\mathcal{N}_v)| \geq (1-\lambda-\epsilon)c$ for all nodes $v \in \kappa_T(V)$.*
2. *For all nodes $u, v \in \kappa_S(V)$, if $(u,v) \in E$, then either $u \in \kappa(\mathcal{N}_v)$ or $v \in \kappa(\mathcal{N}_u)$.*

Then these two conditions continue to hold even after we modify $\kappa(G)$ using the procedure in Section 5.2.3 (resp. Section 5.2.4).

Proof. Follows from the descriptions of the procedures in Sections 5.2.3 and 5.2.4.

Recall that before the (λ, c)-epoch begins, the graph $\kappa(G)$ is an (ϵ, c)-kernel of G. Since every (ϵ, c)-kernel is also a $(\lambda+\epsilon, c)$-kernel, we repeatedly invoke Lemma 5.5 after each update in G, and conclude that $\kappa(G)$ satisfies Invariants 5, 6 (replacing ϵ by $\lambda+\epsilon$) throughout the duration of the epoch. For the rest of this section, we focus on proving the remaining Invariant 4. Specifically, we will show that at any point in time during the epoch, we have $|\kappa(\mathcal{N}_v)| \leq (1+\lambda+\epsilon)c$ for all $v \in V$.

Fix any node $v \in V$. When the (λ, c)-epoch begins, the subgraph $\kappa(G)$ is an (ϵ, c)-kernel of G, so that we have $|\kappa(\mathcal{N}_v)| \leq (1+\epsilon)c$. During the epoch, the node v can get new friends under two possible situations.

1. An edge incident to v has just been inserted into (resp. deleted from) the graph G, and the procedure in Section 5.2.3 (resp. the subroutine REFILL-NOW(v)) is going to be called.
2. The subroutine REFILL-NOW(u) is going to be called for some $u \in \mathcal{N}_v$.

If we are in situation (1) and the node v already has more than $(1+\epsilon)c$ friends, then the procedure under consideration will not end up adding any more node to $\kappa(\mathcal{N}_v)$. Thus, it suffices to show that the node v can get at most λc new friends during the epoch under situation (2). Note that each call to REFILL-NOW(u), $u \neq v$, creates at most one new friend for v. Accordingly, if we show that the subroutine REFILL-NOW(.) is called at most λc times during the entire epoch, then this will suffice to conclude the proof of Theorem 5.2.

LEMMA 5.6. *The subroutine* REFILL-NOW(.) *is called at most λc times during a (λ, c)-epoch.*

Proof. When the epoch begins, every tight node has at least $(1-\epsilon)c$ friends, and during the epoch, the status of a node u is changed from slack to tight only when $|\kappa(\mathcal{N}_u)|$ exceeds $(1-\epsilon)c$. On the other hand, the subroutine REFILL-NOW(u) is called only when u is tight and $|\kappa(\mathcal{N}_u)|$ falls below $(1-\lambda-\epsilon)c$. Thus, each call to REFILL-NOW(.) corresponds to a scenario where a tight node has lost at least λc friends. Each edge deletion in G leads to at most two such losses (one for each of the endpoints), whereas an edge insertion in G leads to no such event. Since a (λ, c)-epoch contains at most $\lambda^2 c^2/2$ edge deletions (Definition 3), a counting argument shows that such an epoch can result in at most $(\lambda^2 c^2/2) \cdot 2/(\lambda c) = \lambda c$ calls to REFILL-NOW(.).

References

[1] Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *55th IEEE Symposium on Foundations of Computer Science*, 2014.

[2] Surender Baswana, Manoj Gupta, and Sandeep Sen. Fully dynamic maximal matching in $O(\log n)$ update time. In *52nd IEEE Symposium on Foundations of Computer Science*, pages 383–392, 2011.

[3] Ran Duan and Seth Pettie. Approximating maximum weight matching in near-linear time. In *51st IEEE Symposium on Foundations of Computer Science*, pages 673–682, 2010.

[4] Manoj Gupta and Richard Peng. Fully dynamic $(1+\epsilon)$-approximate matchings. In *54th IEEE Symposium on Foundations of Computer Science*, pages 548–557, 2013.

[5] John E. Hopcroft and Richard M. Karp. A $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. In *12th Symposium on Switching and Automata Theory*, pages 122–125, 1971.

[6] Zoran Ivkovic and Errol L. Lloyd. Fully dynamic maintenance of vertex cover. In *Graph-Theoretic Concepts in Computer Science*, pages 99–111, 1993.

[7] Subhash Khot and Oded Regev. Vertex cover might be hard to approximate to within $2 - \epsilon$. *Journal of Computer and System Sciences*, 74(3):335–349, 2008.

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

[9] Aleksander Madry. Navigating central path with electrical flows: From flows to matchings, and back. In *54th IEEE Symposium on Foundations of Computer Science*, pages 253–262, 2013.

[10] Silvio Micali and Vijay V. Vazirani. An $O(\sqrt{|V|}\,|E|)$ algorithm for finding maximum matching in general graphs. In *21st IEEE Symposium on Foundations of Computer Science*, pages 17–27, 1980.

[11] Marcin Mucha and Piotr Sankowski. Maximum matchings via Gaussian elimination. In *45th IEEE Symposium on Foundations of Computer Science*, pages 248–255, 2004.

[12] Ofer Neiman and Shay Solomon. Simple deterministic algorithms for fully dynamic maximal matching. In *45th ACM Symposium on Theory of Computing*, pages 745–754, 2013.

[13] Krzysztof Onak and Ronitt Rubinfeld. Maintaining a large matching and a small vertex cover. In *42nd ACM Symposium on Theory of Computing*, pages 457–464, 2010.

[14] Piotr Sankowski. Faster dynamic matchings and vertex connectivity. In *18th ACM-SIAM Symposium on Discrete Algorithms*, pages 118–126, 2007.