Deterministically Maintaining a $(2+\epsilon)$-Approximate Minimum Vertex Cover in $O(1/\epsilon^2)$ Amortized Update Time

Sayan Bhattacharya* Janardhan Kulkarni†

July 11, 2019

Abstract

We consider the problem of maintaining an (approximately) minimum vertex cover in an n-node graph $G = (V, E)$ that is getting updated dynamically via a sequence of edge insertions/deletions. We show how to maintain a $(2+\epsilon)$-approximate minimum vertex cover, *deterministically*, in this setting in $O(1/\epsilon^2)$ amortized update time.

Prior to our work, the best known deterministic algorithm for maintaining a $(2+\epsilon)$-approximate minimum vertex cover was due to Bhattacharya, Henzinger and Italiano [SODA 2015]. Their algorithm has an update time of $O(\log n/\epsilon^2)$. Recently, Bhattacharya, Chakrabarty, Henzinger [IPCO 2017] and Gupta, Krishnaswamy, Kumar, Panigrahi [STOC 2017] showed how to maintain an $O(1)$-approximation in $O(1)$-amortized update time for the same problem. Our result gives an *exponential* improvement over the update time of Bhattacharya et al. [SODA 2015], and nearly matches the performance of the *randomized* algorithm of Solomon [FOCS 2016] who gets an approximation ratio of 2 and an expected amortized update time of $O(1)$.

We derive our result by analyzing, via a novel technique, a variant of the algorithm by Bhattacharya et al. We consider an idealized setting where the update time of an algorithm can take any arbitrary fractional value, and use insights from this setting to come up with an appropriate potential function. Conceptually, this framework mimics the idea of an LP-relaxation for an optimization problem. The difference is that instead of relaxing an integral objective function, we relax the update time of an algorithm itself. We believe that this technique will find further applications in the analysis of dynamic algorithms.

1 Introduction

Consider an undirected graph $G = (V, E)$ with $n = |V|$ nodes, and suppose that we have to compute an (approximately) minimum vertex cover[1] in G. This problem is well-understood in the static setting. There is a simple linear time greedy algorithm that returns a *maximal matching*[2] in G. Let $V(M) \subseteq V$ denote the set of nodes that are matched in M. Using the duality between maximum matching and minimum vertex cover, it is easy to show that the set $V(M)$ forms a 2-approximate minimum vertex cover in G. Accordingly, we can compute a 2-approximate minimum vertex cover in linear time. In contrast, under the Unique Games Conjecture [15], there is no polynomial time $(2-\epsilon)$-approximation algorithm for minimum vertex cover for any $\epsilon > 0$. In this paper, we consider the problem of maintaining an (approximately) minimum vertex cover in a *dynamic* graph, which gets *updated* via a sequence of edge insertions/deletions. The time taken to handle the insertion or deletion of an edge is called the *update time* of the algorithm. The goal is to design a dynamic algorithm with small approximation ratio whose update time is significantly faster than the trivial approach of recomputing the solution *from scratch* after every update.

A naive approach for this problem will be to maintain a maximal matching M and the set of matched nodes $V(M)$ as follows. When an edge (u, v) gets inserted into G, add the edge (u, v) to the matching iff both of its endpoints u, v are currently unmatched. In contrast, when an edge (u, v) gets deleted from G, first check if the edge (u, v) was matched in M just before this deletion. If yes, then remove the edge (u, v) from M and try to *rematch* its endpoints $x \in \{u, v\}$. Specifically, for every endpoint $x \in \{u, v\}$, scan through all the edges $(x, y) \in E$ incident on x till an edge (x, y) is found whose other endpoint y is currently unmatched, and at that point add the edge (x, y) to the matching M and stop the scan. Since a node x can have $\Theta(n)$ neighbors, this approach leads to an update time of $\Theta(n)$.

Our main result is stated in Theorem 1.1. Note that the amortized update time[3] of our algorithm is independent

*University of Warwick, Coventry, UK. Email: S.Bhattacharya@warwick.ac.uk

†Microsoft Research, Redmond, USA. Email: jakul@microsoft.com

[1] A vertex cover in G is a subset of nodes $S \subseteq V$ such that every edge $(u, v) \in E$ has at least one endpoint in S.

[2] A matching in G is a subset of edges $M \subseteq E$ such that no two edges in M share a common endpoint. A matching M is maximal if for every edge $(u, v) \in E \setminus M$, either u or v is matched in M.

[3] Following the standard convention in dynamic algorithms literature,

Approximation Ratio	Amortized Update Time	Algorithm	Reference
$(2+\epsilon)$	$O(\log n/\epsilon^2)$	deterministic	Bhattacharya et al. [8]
$O(1)$	$O(1)$	deterministic	Gupta et al. [13] and Bhattacharya et al. [6]
2	$O(1)$	randomized	Solomon [20]

Table 1: State of the art on dynamic algorithms with fast amortized update times for minimum vertex cover.

of n. As an aside, our algorithm also maintains a $(2+\epsilon)$-approximate maximum *fractional* matching as a dual certificate, deterministically, in $O(\epsilon^{-2})$ amortized update time.

THEOREM 1.1. *For any $0 < \epsilon < 1$, we can maintain a $(2+\epsilon)$-approximate minimum vertex cover in a dynamic graph, deterministically, in $O(\epsilon^{-2})$ amortized update time.*

1.1 Perspective The first major result on maintaining a small vertex cover and a large matching in a dynamic graph appeared in STOC 2010 [18]. By now, there is a large body of work devoted to this topic: both on general graphs [1, 2, 3, 4, 5, 7, 9, 10, 12, 13, 14, 16, 17, 18, 19] and on graphs with bounded arboricity [4, 5, 16, 19]. These results give interesting tradeoffs between various parameters such as (a) approximation ratio, (b) whether the algorithm is deterministic or randomized, and (c) whether the update time is amortized or worst case. In this paper, our focus is on aspects (a) and (b). *We want to design a dynamic algorithm for minimum vertex cover that is deterministic and has (near) optimal approximation ratio and amortized update time.* In particular, we are *not* concerned with the worst case update time of our algorithm. From this specific point of view, the literature on dynamic vertex cover can be summarized as follows.

Randomized algorithms. Onak and Rubinfeld [18] presented a randomized algorithm for maintaining a $O(1)$-approximate minimum vertex cover in $O(\log^2 n)$ expected update time. This bound was improved upon by Baswana, Gupta, Sen [3], and subsequently by Solomon [20], who obtained a 2-approximation in $O(1)$ expected update time.

Deterministic algorithms. Bhattacharya, Henzinger and Italiano [8] showed how to deterministically maintain a $(2+\epsilon)$-approximate minimum vertex cover in $O(\log n/\epsilon^2)$ update time. Subsequently, Bhattacharya, Chakrabarty and Henzinger [6] and Gupta, Krishnaswamy, Kumar and Panigrahy [13] gave deterministic dynamic algorithms for this problem with an approximation ratio of $O(1)$ and update time of $O(1)$. The algorithms designed in these two papers [6, 13] extend to the more general problem of dynamic set cover.

an algorithm has $O(\alpha)$ amortized update time if starting from a graph $G = (V, E)$ where $E = \emptyset$, it takes $O(t \cdot \alpha)$ time overall to handle any sequence of t edge insertions/deletions in G.

Thus, from our perspective the state of the art results on dynamic vertex cover prior to our paper are summarized in Table 1. Note that the results stated in Table 1 are mutually incomparable. Specifically, the algorithms in [8, 13, 6] are all deterministic, but the paper [8] gives near-optimal (under Unique Games Conjecture) approximation ratio whereas the papers [13, 6] give optimal update time. In contrast, the paper [20] gives optimal approximation ratio and update time, but the algorithm there is randomized. Our main result as stated in Theorem 1.1 combines the best of the both worlds, by showing that there is a dynamic algorithm for minimum vertex cover that is simultaneously (a) deterministic, and has (b) near-optimal approximation ratio and (c) optimal update time for constant $0 < \epsilon < 1$. In other words, we get an *exponential* improvement in the update time bound in [8], without increasing the approximation ratio or using randomization.

Most of the randomized dynamic algorithms in the literature, including the ones [1, 3, 18, 20] that are relevant to this paper, assume that the adversary is *oblivious*. Specifically, this means that the future edge insertions/deletions in the input graph do not depend on the current solution being maintained by the algorithm. A deterministic dynamic algorithm does not require this assumption, and hence designing deterministic dynamic algorithms for fundamental optimization problems such as minimum vertex cover is an important research agenda in itself. Our result should be seen as being part of this research agenda.

Our technique. A novel and interesting aspect of our techniques is that we *relax the notion of update time of an algorithm*. We consider an idealized, continuous world where the update time of an algorithm can take any fractional, arbitrarily small value. We first study the behavior of a natural dynamic algorithm for minimum vertex cover in this idealized world. Using insights from this study, we design an appropriate potential function for analyzing the update time of a minor variant of the algorithm from [8] in the *real-world*. Conceptually, this framework mimics the idea of an LP-relaxation for an optimization problem. The difference is that instead of relaxing an integral objective function, we relax the update time of an algorithm itself. We believe that this technique will find further applications in the analysis of dynamic algorithms.

Organization of the rest of the paper. In Section 2, we

present a summary of the algorithm from [8]. A reader already familiar with the algorithm will be able to quickly skim through this section. In Section 3, we analyze the update time of the algorithm in an idealized, continuous setting. This sets up the stage for the analysis of our actual algorithm in the real-world. We present an overview of our algorithm and analysis in the "real-world" in Section 4. The full version of our algorithm, along with a complete analysis of its update time, appears in [11].

2 The framework of Bhattacharya, Henzinger and Italiano [8]

We henceforth refer to the dynamic algorithm developed in [8] as the BHI15 algorithm. In this section, we give a brief overview of the BHI15 algorithm, which is based on a primal-dual approach that simultaneously maintains a fractional matching[4] and a vertex cover whose sizes are within a factor of $(2+\epsilon)$ each other.

Notations. Let $0 \leq w(e) \leq 1$ be the weight assigned to an edge $e \in E$. Let $W_y = \sum_{x \in N_y} w(x,y)$ be the total weight received by a node $y \in V$ from its incident edges, where $N_y = \{x \in V : (x,y) \in E\}$ denotes the set of neighbors of y. The BHI15 algorithm maintains a partition of the node-set V into $L+1$ *levels*, where $L = \log_{(1+\epsilon)} n$. Let $\ell(y) \in \{0, \ldots, L\}$ denote the level of a node $y \in V$. For any two integers $i, j \in [0, L]$ and any node $x \in V$, let $N_y(i,j) = \{x \in N_y : i \leq \ell(x) \leq j\}$ denote the set of neighbors of y that lie in between level i and level j. The level of an edge $(x,y) \in E$ is denoted by $\ell(x,y)$, and it is defined to be equal to the maximum level among its two endpoints, that is, we have $\ell(x,y) = \max(\ell(x), \ell(y))$. In the BHI15 framework, the weight of an edge is completely determined by its level. In particular, we have $w(x,y) = (1+\epsilon)^{-\ell(x,y)}$, that is, the weight $w(x,y)$ decreases exponentially with the level $\ell(x,y)$.

A static primal-dual algorithm. To get some intuition behind the BHI15 framework, consider the following static primal-dual algorithm. The algorithm proceeds in rounds. Initially, before the first round, every node $y \in V$ is at level $\ell(y) = L$ and every edge $(x,y) \in E$ has weight $w(x,y) = (1+\epsilon)^{-\ell(x,y)} = (1+\epsilon)^{-L} = 1/n$. Since each node has at most $n-1$ neighbors in an n-node graph, it follows that $W_y = (1/n) \cdot |N_y| \leq (n-1)/n$ for all nodes $y \in V$ at this point in time. Thus, we have $0 \leq W_y < 1$ for all nodes $y \in V$, so that the edge-weights $\{w(e)\}$ form a valid fractional matching in G at this stage. We say that a node y is *tight* if $W_y \geq 1/(1+\epsilon)$ and *slack* otherwise. In each subsequent round, we identify the set of tight nodes $T = \{y \in V : W_y \geq 1/(1+\epsilon)\}$, set $\ell(y) = \ell(y) - 1$ for all $y \in V \setminus T$, and then raise the weights of the edges in the subgraph induced by $V \setminus T$ by a factor of $(1+\epsilon)$. As we only raise the weights of the edges whose both endpoints are slack, the edge-weights $\{w(e)\}$ continue to be a valid fractional matching in G. The algorithm stops when every edge has at least one tight endpoint, so that we are no longer left with any more edges whose weights can be raised.

Clearly, the above algorithm guarantees that the weight of an edge $(x,y) \in E$ is given by $w(x,y) = (1+\epsilon)^{-\ell(x,y)}$. It is also easy to check that the algorithm does not run for more than L rounds, for the following reason. If after starting from $1/n = (1+\epsilon)^{-L}$ we increase the weight of an edge (x,y) more than L times by a factor of $(1+\epsilon)$, then we would end up having $w(x,y) \geq (1+\epsilon) > 1$, and this would mean that the edge-weights $\{w(e)\}$ no longer form a valid fractional matching. Thus, we conclude that $\ell(y) \in \{0, \ldots, L\}$ for all $y \in V$ at the end of this algorithm. Furthermore, at that time every node $y \in V$ at a nonzero level $\ell(y) > 0$ is tight. The following invariant, therefore, is satisfied.

INVARIANT 1. *For every node y, we have $1/(1+\epsilon) \leq W_y \leq 1$ if $\ell(y) > 0$, and $0 \leq W_y \leq 1$ if $\ell(y) = 0$.*

Every edge $(x,y) \in E$ has at least one tight endpoint under Invariant 1. To see why this is true, note that if the edge has some endpoint $z \in \{x,y\}$ at level $\ell(z) > 0$, then Invariant 1 implies that the node z is tight. On the other hand, if $\ell(x) = \ell(y) = 0$, then the edge (x,y) has weight $w(x,y) = (1+\epsilon)^{-0} = 1$ and both its endpoints are tight, for we have $W_x, W_y \geq w(x,y) = 1$. In other words, the set of tight nodes constitute a valid vertex cover of the graph G. Since every tight node y has weight $1 \geq W_y \geq 1/(1+\epsilon)$, and since every edge (x,y) contributes its own weight $w(x,y)$ towards both W_x and W_y, a simple counting argument implies that the number of tight nodes is within a factor $2(1+\epsilon)$ of the sum of the weights of the edges in G. Hence, we have a valid vertex cover and a valid fractional matching whose sizes are within a factor $2(1+\epsilon)$ of each other. It follows that the set of tight nodes form a $2(1+\epsilon)$-approximate minimum vertex cover and that the edge-weights $\{w(e)\}$ form a $2(1+\epsilon)$-approximate maximum fractional matching.

Making the algorithm dynamic. In the dynamic setting, all we need to ensure is that we maintain a partition of the node-set V into levels $\{0, \ldots, L\}$ that satisfies Invariant 1. By induction hypothesis, suppose that Invariant 1 is satisfied by every node until this point in time. Now, an edge (u,v) is either inserted into or deleted from the graph. The former event increases the weight W_x of each node $x \in \{u,v\}$ by $(1+\epsilon)^{-\max(\ell(u),\ell(v))}$, whereas the latter

[4] A fractional matching in $G = (V,E)$ assigns a weight $0 \leq w(e) \leq 1$ to each edge $e \in E$, subject to the constraint that the total weight received by any node from its incident edges is at most one. The size of the fractional matching is given by $\sum_{e \in E} w(e)$. It is known that the maximum matching problem is the dual of the minimum vertex cover problem. Specifically, it is known that the size of the maximum fractional matching is at most the size of the minimum vertex cover.

event decreases the weight W_x of each node $x \in \{u, v\}$ by $(1+\epsilon)^{-\max(\ell(u),\ell(v))}$. As a result, one or both of the endpoints $\{u, v\}$ might now violate Invariant 1. For ease of presentation, we say that a node is *dirty* if it violates Invariant 1. To be more specific, a node v is *dirty* if either (a) $W_v > 1$ or (b) $\{\ell(v) > 0$ and $W_v < 1/(1+\epsilon)\}$. In case (a), we say that the node v is *up-dirty*, whereas in case (b) we say that the node v is *down-dirty*. To continue with our discussion, we noted that the insertion or deletion of an edge might make one or both of its endpoints dirty. In such a scenario, we call the subroutine described in Figure 1. Intuitively, this subroutine keeps changing the levels of the dirty nodes in a greedy manner till there is no dirty node (equivalently, till Invariant 1 is satisfied).

In a bit more details, suppose that a node x at level $\ell(x) = i$ is up-dirty. If our goal is to make this node satisfy Invariant 1, then we have to decrease its weight W_x. A greedy way to achieve this outcome is to increase its level $\ell(x)$ by one, by setting $\ell(x) = i + 1$, without changing the level of any other node. This decreases the weights of all the edges $(x, y) \in E$ incident on x whose other endpoints y lie at levels $\ell(y) \leq i$. The weight of every other edge remains unchanged. Hence, this decreases the weight W_x. Note that this step changes the weights of the neighbors $y \in N_x(0, i)$ of x that lie at level i or below. These neighbors, therefore, might now become dirty. Such neighbors will be handled in some future iteration of the WHILE loop. Furthermore, it might be the case that the node x itself remains dirty even after this step, since the weight W_x has not decreased by a sufficient amount. In such an event, the node x itself will be handled again in a future iteration of the WHILE loop. Next, suppose that the node x is down-dirty. By an analogous argument, we need to increase the weight W_x if we want to make the node x satisfy Invariant 1. Accordingly, we decrease its level $\ell(x)$ in step 5 of Figure 1. As in the previous case, this step might lead to some neighbors of x becoming dirty, who will be handled in future iterations of the WHILE loop. If the node x itself remains dirty after this step, it will also be handled in some future iteration of the WHILE loop.

To summarize, there is no dirty node when the WHILE loop terminates, and hence Invariant 1 *is* satisfied. But due to the *cascading effect* (whereby a given iteration of the WHILE loop might create additional dirty nodes), it is not clear why this simple algorithm will have a small update time. In fact, it is by no means obvious that the WHILE loop in Figure 1 is even guaranteed to terminate. The main result in [8] was that (a slight variant of) this algorithm actually has an amortized update time of $O(\log n/\epsilon^2)$. Before proceeding any further, however, we ought to highlight the data structures used to implement this algorithm.

Data structures. Each node $x \in V$ maintains its weight W_x and level $\ell(x)$. This information is sufficient for a node to detect when it becomes dirty. In addition, each node $x \in V$ maintains the following doubly linked lists: For every level $i > \ell(x)$, it maintains the list $E_x(i) = \{(x, y) \in E : \ell(y) = i\}$ of edges incident on x whose other endpoints lie at level i. Thus, every edge $(x, y) \in E_x(i)$ has a weight $w(x, y) = (1+\epsilon)^{-i}$. The node x also maintains the list $E_x^- = \{(x, y) \in E : \ell(y) \leq \ell(x)\}$ of edges whose other endpoints are at a level that is at most the level of x. Thus, every edge $(x, y) \in E_x^-$ has a weight of $w(x, y) = (1+\epsilon)^{-\ell(x)}$. We refer to these lists as the *neighborhood lists* of x. Intuitively, there is one neighborhood list for each nonempty subset of edges incident on x that have the same weight. For each edge $(x, y) \in E$, the node x maintains a pointer to its own position in the neighborhood list of y it appears in, and vice versa. Using these pointers, a node can be inserted into or deleted from a neighborhood list in $O(1)$ time. We now bound the time required to update these data structures during one iteration of the WHILE loop.

CLAIM 2. *Consider a node x that moves from a level i to level $i+1$ during an iteration of the* WHILE *loop in Figure 1. Then it takes $O(|N_x(0, i)|)$ time to update the relevant data structures during that iteration, where $N_x(0, i) = \{y \in N_x : 0 \leq \ell(y) \leq i\}$ is the set of neighbors of x that lie on or below level i.*

Proof. (Sketch) Consider the event where the node x moves up from level i to level $i + 1$. The key observation is this. If the node x has to change its own position in the neighborhood list of another node y due to this event, then we must have $y \in N_x(0, i)$. And as far as changing the neighborhood lists of x itself is concerned, all we need to do is to merge the list E_x^- with the list $E_x(i + 1)$, which takes $O(1)$ time.

CLAIM 3. *Consider a node x that moves from a level i to level $i-1$ during an iteration of the* WHILE *loop in Figure 1. Then it takes $O(|N_x(0, i)|)$ time to update the relevant data structures during that iteration, where $N_x(0, i) = \{y \in N_x : 0 \leq \ell(y) \leq i\}$ is the set of neighbors of x that lie on or below level i.*

Proof. (Sketch) Consider the event where the node x moves down from level i to level $i - 1$. If the node x has to change its own position in the neighborhood list of another node y due to this event, then we must have $y \in N_x(0, i - 1)$. On the other hand, in order to update the neighborhood lists of x itself, we have to visit all the nodes $y \in E_x^-$ one after the other and check their levels. For each such node y, if we find that $\ell(y) = i$, then we have to move y from the list E_x^- to the list $E_x(i)$. Thus, the total time spent during this iteration is $O(|N_x(0, i - 1)| + |N_x(0, i)|) = O(|N_x(0, 1)|)$. The last equality holds since $N_x(0, i - 1) \subseteq N_x(0, i)$.

1. WHILE there exists some dirty node x:
2. IF the node x is up-dirty, THEN // We have $W_x > 1$.
3. Move it up by one level by setting $\ell(x) \leftarrow \ell(x) + 1$.
4. ELSE IF the node x is down-dirty, THEN // We have $\ell(x) > 0$ and $W_x < 1/(1+\epsilon)$.
5. Move it down one level by setting $\ell(x) \leftarrow \ell(x) - 1$.

Figure 1: Subroutine: FIX(.) is called after the insertion/deletion of an edge.

2.1 The main technical challenge: Can we bring down the update time to $O(1)$? As we mentioned previously, it was shown in [8] that the dynamic algorithm described above has an amortized update time of $O(\log n/\epsilon^2)$. In order to prove this bound, the authors in [8] had to use a complicated potential function. Can we show that (a slight variant of) the same algorithm actually has an update time of $O(1)$ for every fixed ϵ? This seems to be quite a challenging goal, for the following reasons. For now, assume that ϵ is some small constant.

In the potential function developed in [8], whenever an edge (u, v) is inserted into the graph, we create $O(1) \cdot (L - \max(\ell(u), \ell(v)))$ many *tokens*. For each endpoint $x \in \{u, v\}$ and each level $\max(\ell(u), \ell(v)) < i \leq L$, we store $O(1)$ tokens for the node x at level i. These tokens are used to account for the time spent on updating the data structures when a node x moves up from a lower level to a higher level, that is, in dealing with up-dirty nodes. It immediately follows that if we only restrict ourselves to the time spent in dealing with up-dirty nodes, then we get an amortized update time of $O(\log n)$. This is because of the following simple accounting: Insertion of an edge (u, v) creates at most $O(L - \max(\ell(u), \ell(v))) = O(\log n)$ many tokens, and each of these tokens is used to pay for one unit of computation performed by our algorithm while dealing with the up-dirty nodes. Next, it is also shown in [8] that, roughly speaking, over a sufficiently long time horizon the time spent in dealing with the down-dirty nodes is dominated by the time spent in dealing with the up-dirty nodes. This gives us an overall amortized update time of $O(\log n)$. From this very high level description of the potential function based analysis in [8], it seems intrinsically challenging to overcome the $O(\log n)$ barrier. This is because nothing is preventing an edge (u, v) from moving up $\Omega(\log n)$ levels after getting inserted, and according to [8] the only way we can bound this type of *work* performed by the algorithm is by *charging* it to the insertion of the edge (u, v) itself. In recent years, attempts were made to overcome this $O(\log n)$ barrier. The papers [6, 13], for example, managed to improve the amortized update time to $O(1)$, but only at the cost of increasing the approximation ratio from $(2 + \epsilon)$ to some unspecified constant $\Theta(1)$. The question of getting $(2 + \epsilon)$-approximation in $O(1)$ time, however, remained wide open.

It seems unlikely that we will just stumble upon a suitable potential function that proves the amortized bound of $O(1)$ by trial and error: There are way too many options to choose from! What we instead need to look for is a systematic *meta-method* for finding the suitable potential function – something that will allow us to prove the optimal possible bound for the given algorithm. This is elaborated upon in Section 3.

3 Our technique: A thought experiment with a continuous setting

In order to search for a suitable potential function, we consider an idealized setting where the level of a node or an edge can take *any* (not necessarily integral) value in the *continuous* interval $[0, L]$, where $L = \log_e n$. To ease notations, here we assume that the weight of an edge (x, y) is given by $w(x, y) = e^{-\ell(x,y)}$, instead of being equal to $(1 + \epsilon)^{-\ell(x,y)}$. This makes it possible to assign each node to a (possibly fractional) level in such a way that the edge-weights $\{w(e)\}$ form a *maximal* fractional matching, and the nodes $y \in V$ with weights $W_y = 1$ form a 2-approximate minimum vertex cover. We use the notations introduced in the beginning of Section 2. In the idealized setting, we ensure that the following invariant is satisfied.

INVARIANT 4. *For every node $y \in V$, we have $W_y = 1$ if $\ell(y) > 0$, and $W_y \leq 1$ if $\ell(y) = 0$.*

A static primal-dual algorithm. As in Section 2, under Invariant 4 the levels of the nodes have a natural primal-dual interpretation. To see this, consider the following static algorithm. We initiate a continuous process at time $t = -\log n$. At this stage, we set $w(e) = e^t = 1/n$ for every edge $e \in E$. We say that a node y is tight iff $W_y = 1$. Since the maximum degree of a node is at most $n - 1$, no node is tight at time $t = -\log n$. With the passage of time, the edge-weights keep increasing exponentially with t. During this process, whenever a node y becomes tight we *freeze* (i.e., stop raising) the weights of all its incident edges. The process stops at time $t = 0$. The level of a node y is defined as $\ell(y) = -t_y$, where t_y is the time when it becomes tight during this process. If the node does not become tight till the very end, then we define $\ell(y) = t_y = 0$. When the process ends at time $t = 0$, it is easy to check that $w(x, y) = e^{-\ell(x,y)}$ for every edge $(x, y) \in E$ and that Invariant 4 is satisfied.

We claim that under Invariant 4 the set of tight nodes form a 2-approximate minimum vertex cover in G. To see why this is true, suppose that there is an edge (x, y) between two nodes x and y with $W_x, W_y < 1$. According to Invariant 4, both the nodes x, y are at level 0. But this implies that $w(x, y) = e^{-0} = 1$ and hence $W_x, W_y \geq w(x, y) = 1$, which leads to a contradiction. Thus, the set of tight nodes must be a vertex cover in G. Since $W_v = 1$ for every tight node $v \in V$, and since every edge $(u, v) \in E$ contributes the weight $w(x, y)$ to both W_x and W_y, a simple counting argument implies that the edge-weights $\{w(e)\}$ forms a fractional matching in G whose size is at least $(1/2)$ times the number of tight nodes. The claim now follows from the duality between maximum fractional matching and minimum vertex cover.

We will now describe how Invariant 4 can be maintained in a dynamic setting – when edges are getting inserted into or deleted from the graph. For ease of exposition, we will use Assumption 5. Since the level of a node can take *any* value in the *continuous* interval $[0, L]$, this does not appear to be too restrictive.

ASSUMPTION 5. *For any two nodes $x \neq y$, if $\ell(y), \ell(y) > 0$, then we have $\ell(x) \neq \ell(y)$.*

Notations. Let $N_x^+ = \{y \in V : (x, y) \in E \text{ and } \ell(y) > \ell(x)\}$ denote the set of *up-neighbors* of a node $x \in V$, and let $N_x^- = \{y \in V : (x, y) \in E \text{ and } \ell(y) < \ell(x)\}$ denote the set of *down-neighbors* of x. Assumption 5 implies that $N_x = N_x^+ \cup N_x^-$. Finally, let W_x^+ (resp. W_x^-) denote the total weight of the edges incident on x whose other endpoints are in N_x^+ (resp. N_x^-). We thus have $W_x = W_x^+ + W_x^-$ and $W_x^- = |N_x^-| \cdot e^{-\ell(x)}$. We will use these notations throughout the rest of this section.

Insertion or deletion of an edge (u, v). We focus only on the case of edge insertion, as the case of edge deletion can be handled in an analogous manner. Consider the *event* where an edge (u, v) is inserted into the graph. By induction hypothesis Invariant 4 is satisfied just before this event, and without any loss of generality suppose that $\ell(v) = i \geq \ell(u) = j$ at that time. For ease of exposition, we assume that $i, j > 0$: the other case can be dealt with using similar ideas. Then, we have $W_u = W_v = 1$ just before the event (by Invariant 4) and $W_u = W_v = 1 + e^{-i}$ just after the event (since $i \geq j$). So the nodes u and v violate Invariant 4 just after the event. We now explain the process by which the nodes change their levels so as to ensure that Invariant 4 becomes satisfied again. This process consists of two *phases* – one for each endpoint. $x \in \{u, v\}$. We now describe each of these phases.

Phase I: *This phase is defined by a continuous process which is driven by the node v.* Specifically, in this phase the node v continuously increases its level so as to decrease its weight W_v. The process stops when the weight W_v becomes equal to 1. During the same process, every other node $x \neq v$ continuously changes its level so as to ensure that its weight W_x remains fixed.[5] This creates a cascading effect which leads to a long chain of interdependent movements of nodes. To see this, consider an infinitesimal time-interval $[t, t + dt]$ during which the node v increases its level from $\ell(v)$ to $\ell(v) + d\ell(v)$. The weight of every edge $(v, x) \in E$ with $x \in N_v^-$ *decreases* during this interval, whereas the weight of every other edge remains unchanged. Thus, during this interval, the upward movement of the node v leads to a *decrease* in the weight W_x of every neighbor $x \in N_v^-$. Each such node $x \in N_v^-$ wants to *nullify* this effect and ensure that W_x remains fixed. Accordingly, each such node $x \in N_v^-$ *decreases* its level during the same infinitesimal time-interval $[t, t + dt]$ from $\ell(x)$ to $\ell(x) + d\ell(x)$, where $d\ell(x) < 0$. The value of $d\ell(x)$ is such that W_x actually remains unchanged during the time-interval $[t, t + dt]$. Now, the weights of the neighbors $y \in N_x^-$ of x also get affected as x changes its level, and as a result each such node y also changes its level so as to preserve its own weight W_y, and so on and so forth. We emphasize that all these movements of different nodes occur *simultaneously*, and in a continuous fashion. Intuitively, the set of nodes form a self-adjusting system – akin to a spring. Each node moves in a way which ensure that its weight becomes (or, remains equal to) a "critical value". For the node u this critical value is $1 + e^{-i}$, and for every other node (at a nonzero level) this critical value is equal to 1. Thus, every node other than u satisfies Invariant 4 when Phase I ends. At this point, we initiate Phase II described below.

Phase II: *This phase is defined by a continuous process which is driven by the node u.* Specifically, in this phase the node u continuously increases its level so as to decrease its weight W_u. The process stops when W_u becomes equal to 1. As in Phase I, during the same process every other node $x \neq u$ continuously changes its level so as to ensure that W_x remains fixed. Clearly, Invariant 4 is satisfied when Phase II ends.

"Work": A proxy for update time. We cannot implement the above continuous process using any data structure, and hence we cannot meaningfully talk about the *update time*

[5] To be precise, this statement does not apply to the nodes at level 0. A node x with $\ell(x) = 0$ remains at level 0 as long as $W_x < 1$, and starts moving upward only when its weight W_x is about to exceed 1. But, morally speaking, this does not add any new perspective to our discussion, and henceforth we ignore this case. Furthermore, one other corner case arises when a node x is at a positive level $\ell(x) > 0$ and $N^-(x) = \emptyset$. Such a node x might make a discrete jump to the minimum level k above where $N^-(x)$ becomes nonempty again. Our algorithm, however, is not performing any "work" during such a jump (as no edge changes its level during such a jump). Thus, for ease of exposition, we ignore such a scenario.

of our algorithm in the idealized, continuous setting. To address this issue, we introduce the notion of *work*, which is defined as follows. We say that our algorithm performs $\delta \geq 0$ work whenever it changes the level $\ell(x, y)$ of an edge (x, y) by δ. Note that δ can take any arbitrary fractional value. To see how the notion of work relates to the notion of update time from Section 2, recall Claim 2 and Claim 3. They state that whenever a node x at level $\ell(x) = k$ moves up or down one level, it takes $O(|N_x(0, k)|)$ time to update the relevant data structures. A moment's thought will reveal that in the former case (when the node moves up) the total work done is equal to $|N_x(0, k)|$, and in the latter case (when the node moves down) the work done is equal to $|N_x(0, k-1)|$. Since $N_x(0, k-1) \subseteq N_x(0, k)$, we have $|N_x(0, k-1)| \leq |N_x(0, k)|$. Thus, the work done by the algorithm is upper bounded by (and, closely related to) the time spent to update the date structures. In light of this observation, we now focus on analyzing the work done by our algorithm in the continuous setting.

3.1 Work done in handling the insertion or deletion of an edge (u, v)

We focus on the case of an edge-insertion. The case of an edge-deletion can be analyzed using similar ideas. Accordingly, suppose that an edge (u, v), where $\ell(v) \geq \ell(u)$, gets inserted into the graph. We first analyze the work done in Phase I, which is driven by the movement of v. Without any loss of generality, we assume that v is changing its level in such a way that its weight W_v is decreasing at unit-rate. Every other node x at a nonzero level wants to preserve its weight W_x at its current value. Thus, we have:

$$\frac{dW_v}{dt} = -1 \tag{3.1}$$

$$\frac{dW_x}{dt} = 0 \text{ for all nodes } x \neq v \text{ with } \ell(x) > 0. \tag{3.2}$$

A note on how the sets N_x^- and N_x^+ and the weights W_x^- and W_x^+ change with time: We will soon write down a few differential equations, which capture the behavior of the continuous process unfolding in Phase I during an infinitesimally small time-interval $[t, t + dt]$. Before embarking on this task, however, we need to clarify the following important issue. Under Assumption 5, at time t the (nonzero) levels of the nodes take distinct, finite values. Thus, we have $\ell_t(x) \neq \ell_t(y)$ for any two nodes $x \neq y$ with $\ell_t(x), \ell_t(y) > 0$, where $\ell_t(z)$ denotes the level of a node z at time t. The level of a node can only change by an infinitesimally small amount during the time-interval $[t + dt]$. This implies that if $\ell_t(x) > \ell_t(y)$ for any two nodes x, y, then we also have $\ell_{t+dt}(x) > \ell_{t+dt}(y)$. In words, while writing down a differential equation we can assume that the sets N_x^+ and N_x^- remain unchanged throughout the infinitesimally small time-interval $[t, t + dt]$.[6] But, this observation does not apply to the weights W_x^- and W_x^+, for the weight $w(x, y) = e^{-\max(\ell(x), \ell(y))}$ of an edge will change if we move the level of its higher endpoint by an infinitesimally small amount.

Let $s_x = \frac{d\ell(x)}{dt}$ denote the *speed* of a node $x \in V$. It is the rate at which the node x is changing its level. Let $f(x, x')$ denote the rate at which the weight $w(x, x')$ of an edge $(x, x') \in E$ is changing. Note that:

$$\text{If } \ell(x) > \ell(x'), \text{ then } f(x, x') = \frac{dw(x,x')}{dt} = \frac{de^{-\ell(x)}}{dt} = \frac{d\ell(x)}{dt} \cdot \frac{de^{-\ell(x)}}{d\ell(x)} = -s_x \cdot w(x, x'). \tag{3.3}$$

Consider a node $x \neq v$ with $\ell(x) > 0$. By (3.2), we have $\frac{dW_x}{dt} = 0$. Hence, we derive that:

$$0 = \frac{dW_x}{dt} = \sum_{x' \in N_x^-} \frac{dw(x,x')}{dt} + \sum_{x' \in N_x^+} \frac{dw(x,x')}{dt} = \sum_{x' \in N_x^-} f(x, x') + \sum_{x' \in N_x^+} f(x, x').$$

Rearranging the terms in the above equality, we get:

$$\text{for every node } x \in V \setminus \{v\} \text{ with } \ell(x) > 0, \quad \sum_{x' \in N_x^-} f(x, x') = -\sum_{x' \in N_x^+} f(x, x') \tag{3.4}$$

Now, consider the node v. By (3.1), we have $\frac{dW_v}{dt} = \sum_{x' \in N_v^-} \frac{dw(x',v)}{dt} = -1$. Hence, we get:

$$\sum_{x' \in N_v^-} f(x', v) = \frac{dW_v}{dt} = -1. \tag{3.5}$$

Conditions (3.4) and (3.5) are reminiscent of a flow constraint. Indeed, the entire process can be visualized as follows. Let $|f(x, y)|$ be the *flow* passing through an edge $(x, y) \in E$. We *pump* 1 unit of flow *into* the node v (follows from (3.1)). This flow then splits up evenly among all the edges $(x, v) \in E$ with $x \in N_v^-$ (follows from (3.5) and (3.3)). As we sweep across the system down to lower and lower levels, we see the same phenomenon: The flow coming into a node x from its neighbors $y \in N_x^+$ splits up evenly among its neighbors $y \in N_x^-$ (follows from (3.4) and (3.3)). Our goal is to analyze the work done by our algorithm. Towards this end, let $P_{(x,x')}$ denote the *power* of an edge $(x, x') \in E$. This is the amount of work being done by the algorithm on the edge (x, x') per time unit. Thus, from (3.3), we get:

$$\text{If } \ell(x) > \ell(x'), \text{ then } P_{(x,x')} = |s_x| = \frac{|f(x, x')|}{w(x, x')} = |f(x, x')| \cdot e^{\ell(x)}. \tag{3.6}$$

[6] The sets N_x^+ and N_x^- will indeed change over a sufficiently long, *finite* time-interval. The key observation is that we can ignore this change while writing down a differential equation for an infinitesimally small time-interval.

Let P_x denote the *power* of a node $x \in V$. We define it to be the amount of work being done by the algorithm for changing the level of x per time unit. This is the sum of the powers of the edges whose levels change due to the node x changing its own level. Let $f^-(x) = \sum_{x' \in N_x^-} f(x, x')$. From (3.3), it follows that either $f(x', x) \geq 0$ for all $x' \in N_x^-$, or $f(x', x) \leq 0$ for all $x' \in N_x^-$. In other words, every term in the sum $\sum_{x' \in N_x^-} f(x, x')$ has the same sign. So the quantity $|f^-(x)|$ denotes the total flow moving from the node x to its neighbors $x' \in N_x^-$, and we derive that:

$$\begin{aligned} (3.7) \quad &= \sum_{x' \in N_x^-} P_{(x,x')} = e^{\ell(x)} \cdot \sum_{x' \in N_x^-} |f(x, x')| \\ &= e^{\ell(x)} \cdot |f^-(x)| \end{aligned}$$

The total work done by the algorithm per time unit in Phase I is equal to $\sum_{(x,x') \in E} P_{(x,x')} = \sum_{x \in V} P_x$. We will like to upper bound this sum. We now make the following important observations. First, since the flow only moves downward after getting pumped into the node v at unit rate, conditions (3.4) and (3.5) imply that:

$$(3.8) \qquad \sum_{x : \ell(x) = k} \left| f^-(x) \right| \leq 1 \text{ at every level } k \leq \ell(v).$$

Now, suppose that we get extremely lucky, and we end up in a situation where the levels of all the nodes are integers (this was the case in Section 2). In this situation, as the flow moves down the system to lower and lower levels, the powers of the nodes decrease geometrically as per condition (3.7). Hence, applying (3.8) we can upper bound the sum $\sum_x P_x$ by the geometric series $\sum_{k=0}^{\ell(v)} e^k$. This holds since:

$$(3.9) \quad \sum_{x \in V : \ell(x) = k} P_x = \sum_{x \in V : \ell(x) = k} |f^-(x)| \cdot e^k \leq e^k \text{ at every level } k \leq \ell(v).$$

Thus, in Phase I the algorithm performs $\sum_{k=0}^{\ell(v)} e^k = O\left(e^{\ell(v)}\right)$ units of work per time unit. Recall that Phase I was initiated after the insertion of the edge (u, v), which increased the weight W_v by (say) η_v. During this phase the node v decreases its weight W_v at unit rate, and the process stops when W_v becomes equal to 1. Thus, from the discussion so far we expect Phase I to last for η_v time-units. Accordingly, we also expect the total work done by the algorithm in Phase I to be at most $O\left(e^{\ell(v)}\right) \cdot \eta_v$. Since $\eta_v = e^{-\max(\ell(u), \ell(v))} \leq e^{-\ell(v)}$, we expect that the algorithm will do at most $O\left(e^{\ell(v)}\right) \cdot e^{-\ell(v)} = O(1)$ units of work in Phase I. A similar reasoning applies for Phase II as well. This gives an intuitive explanation as to why an appropriately chosen variant of the BHI15 algorithm [8] should have $O(1)$ update time for every constant $\epsilon > 0$.

3.2 Towards analyzing the "real-world", discretized setting Towards the end of Section 3.1 we made a crucial assumption, namely, that the levels of the nodes are integers. It turns out that if we want to enforce this condition, then we can no longer maintain an *exact* maximal fractional matching and get an approximation ratio of 2. Instead, we will have to satisfied with a fractional matching that is approximately maximal, and the corresponding vertex cover we get will be a $(2 + \epsilon)$-approximate minimum vertex cover. Furthermore, in the idealized continuous setting we could get away with moving the level of a node x, whose weight W_x has only slightly deviated from 1, by any arbitrarily small amount and thereby doing arbitrarily small amount of work on the node at any given time. This is why the intuition we got out of the above discussion also suggests that the overall update time should be $O(1)$ in the *worst case*. This will no longer be possible in the *real-world*, where the levels of the nodes need to be integers. In the real-world, a node x can move to a different integral level only after its weight W_x has changed by a sufficiently large amount, and the work done to move the node to a different level can be quite significant. This is why our analysis in the discretized, real-world gets an *amortized* (instead of worst-case) upper bound of $O(\epsilon^{-2})$ on the update time of the algorithm.

Coming back to the continuous world, suppose that we pump in an infinitesimally small δ amount of weight into a node x at a level $\ell(x) = k > 0$ at unit rate. The process, therefore, lasts for δ time units. During this process, the level of the node x increases by an infinitesimally small amount so as to ensure that its weight W_x remains equal to 1. The work done per time unit on the node x is equal to P_x. Hence, the total work done on the node x during this event is given by:

$$\begin{aligned} \delta \cdot P_x = \delta \cdot \sum_{y \in N_x^-} P_y &= \delta \cdot \sum_{y \in N_x^-} |f(x, y)| \cdot e^{\ell(x)} \\ &= \delta \cdot e^{\ell(x)} \cdot \left| \sum_{y \in N_x^-} f(x, y) \right| = \delta \cdot e^{\ell(x)}. \end{aligned}$$

In this derivation, the first two steps follow from (3.6) and (3.7). The third step holds since $f(x, y) < 0$ for all $y \in N_x^-$, as the node x moves up to a higher level. The fourth step follows from (3.5). Thus, we note that:

OBSERVATION 6. *To change the weight W_x by δ, we need to perform $\delta \cdot e^{\ell(x)}$ units of work on the node x.*

The intuition derived from Observation 6 will guide us while we design a potential function for bounding the amortized update time in the "real-world". This is shown in Section 4.

4 An overview of our algorithm and the analysis in the "real-world"

To keep the presentation as modular as possible, we describe the algorithm itself in Section 4.1, which happens to be almost the same as the BHI15 algorithm from Section 2, with one crucial twist. Accordingly, our main goal in Section 4.1 is to point out the difference between the new algorithm and the old one. We also explain why this difference does not

impact in any significant manner the approximation ratio of $(2+\epsilon)$ derived in Section 2. Moving forward, in Section 4.2 we present a very high level overview of our new potential function based analysis of the algorithm from Section 4.1, which gives the desired bound of $O(1/\epsilon^2)$ on the amortized update time. See the arXiv version [11] for the complete description the algorithm and its analysis.

4.1 The algorithm We start by setting up the necessary notations. We use all the notations introduced in the beginning of Section 2. In addition, for every node $x \in V$ and every level $0 \leq i \leq L$, we let $W_{x\to i} = \sum_{y\in N_x}(1+\epsilon)^{-\max(\ell(y),i)}$ denote what the weight of x would have been if we were to place x at level i, without changing the level of any other node. Note that $W_{x\to i}$ is a monotonically (weakly) decreasing function of i, for the following reason. As we increase the level of x (say) from i to $(i+1)$, all its incident edges $(x,y) \in E$ with $y \in N_x(0,i)$ decrease their weights, and the weights of all its other incident edges remain unchanged.

Up-dirty and down-dirty nodes. We use the same definition of a *down-dirty* node as in Section 2 (see the second paragraph after Invariant 1) – a node x is *down-dirty* iff $\ell(x) > 0$ and $W_x < 1/(1+\epsilon)$. But we slightly change the definition of an *up-dirty* node. Specifically, here we say that a node $x \in V$ is *up-dirty* iff $W_x > 1$ *and* $W_{x\to\ell(x)+1} > 1$. As before, we say that a node is *dirty* iff it is either up-dirty or down-dirty.

Handling the insertion or deletion of an edge. The pseudocode for handling the insertion or deletion of an edge (u,v) remains the same as in Figure 1 – although the conditions which specify when a node is up-dirty have changed. As far as the time spent in implementing the subroutine in Figure 1 is concerned, it is not difficult to come up with suitable data structures so that Claim 2 and Claim 3 continue to hold.

Approximation ratio. Clearly, this new algorithm ensures that there is no dirty node when it is done with handling the insertion or deletion of an edge. We can no longer claim, however, that Invariant 1 is satisfied. This is because we have changed the definition of an up-dirty node. To address this issue, we make the following key observation: If a node x with $W_x > 1$ is *not* up-dirty according to the new definition, then we must have $W_x \leq (1+\epsilon)$. To see why this true, suppose that we have a node x with $W_x > (1+\epsilon)$ that is *not* up-dirty. If we move this node up by one level, then every edge incident on x will decrease its weight by at most a factor of $(1+\epsilon)$, and hence the weight W_x will also decrease by a factor of at most $(1+\epsilon)$. Therefore, we infer that $W_{x\to\ell(x)+1} \geq W_x/(1+\epsilon) > 1$, and the node x *is* in fact up-dirty. This leads to a contradiction. Hence, it must be the case that if a node x with $W_x > 1$ is up-dirty, then $W_x \leq (1+\epsilon)$. This observation implies that if there is no dirty node, then the following conditions are satisfied. (1) $W_x \leq (1+\epsilon)$ for all nodes $x \in V$. (2) $W_x \geq 1/(1+\epsilon)$ for all nodes $x \in V$ at levels $\ell(x) > 0$. Accordingly, we get a valid fractional matching if we scale down the edge-weights by factor of $(1+\epsilon)$. As before, the set of nodes x with $W_x \geq 1/(1+\epsilon)$ forms a valid vertex cover. A simple counting argument (see the paragraph after Invariant 1) implies that the size of this fractional matching is within a $2(1+\epsilon)^2$ factor of the size of this vertex cover. Hence, we get an approximation ratio of $2(1+\epsilon)^2$. Basically, the approximation ratio degrades only by a factor of $(1+\epsilon)$ compared to the analysis in Section 2.

4.2 Bounding the amortized update time Our first task is to find a discrete, real-world analogue of Observation 6 (which holds only in the continuous setting). This is done in Claim 7 below. This relates the time required to move up a node x from level k to level $k+1$ with the change in its weight W_x due to the same event.

CLAIM 7. *Suppose that a node x is moving up from level k to level $k+1$ during an iteration of the* WHILE *loop in Figure 1. Then it takes $O\left((W_{x\to k} - W_{x\to k+1})\cdot\epsilon^{-1}\cdot(1+\epsilon)^k\right)$ time to update the relevant data structures during this iteration.*

Proof. As the node x moves up from level k to level $k+1$, the weight of every edge $(x,y) \in E$ with $y \in N_x(0,k)$ decreases from $(1+\epsilon)^{-k}$ to $(1+\epsilon)^{-(k+1)}$, whereas the weight of every other edge remains unchanged. Hence, it follows that:

$$\begin{aligned} W_{x\to k} - W_{x\to k+1} &= |N_x(0,k)|\cdot\left((1+\epsilon)^{-k} - (1+\epsilon)^{-(k+1)}\right) \\ &= |N_x(0,k)|\cdot\epsilon\cdot(1+\epsilon)^{-(k+1)}. \end{aligned}$$

Rearranging the terms in the above equality, we get:

$$\begin{aligned} |N_x(0,k)| &= (W_{x\to k} - W_{x\to k+1})\cdot\epsilon^{-1}\cdot(1+\epsilon)^{k+1} \\ &= O\left((W_{x\to k} - W_{x\to k+1})\cdot\epsilon^{-1}\cdot(1+\epsilon)^k\right). \end{aligned}$$

The desired proof now follows from Claim 2.

Next, consider a node x that is moving *down* from level k to level $k-1$. We use a different accounting scheme to bound the time spent during this event. This is because the work done on the node x during such an event is equal to $|N_x(0,k-1)|$, but it takes $O\left(|N_x(0,k)|\right)$ time to update the relevant data structures (see Claim 3 and the last paragraph before Section 3.1). Note that $N_x(0,k-1) \subseteq N_x(0,k)$. Thus, although it is possible to bound the work done during this event in a manner analogous to Claim 7, the bound obtained in that manner might be significantly less than the

actual time spent in updating the data structures during this event. Instead, we bound the time spent during this event as specified in Claim 8 below.

CLAIM 8. *Consider a node x moving down from level k to level $k-1$ during an iteration of the* WHILE *loop in Figure 1. Then it takes $O\left((1+\epsilon)^{k-1}\right)$ time to update the relevant data structures during this iteration.*

Proof. By Claim 3, it takes $O\left(|N_x(0,k)|\right)$ time to update the relevant data structures when the node x moves down from level k to level $k-1$. We will now show that $|N_x(0,k)| = O\left((1+\epsilon)^k\right)$. To see why this is true, first note that the node x moves down from level k only if it is down-dirty at that level (see step 4 in Figure 1). Hence, we get: $W_{x\to k} < (1+\epsilon)^{-1}$. When the node is at level k, every edge $(x,y) \in E$ with $y \in N_x(0,k)$ has a weight $w(x,y) = (1+\epsilon)^{-k}$. It follows that $(1+\epsilon)^{-k} \cdot |N_x(0,k)| \leq W_{x\to k} < (1+\epsilon)^{-1}$. Rearranging the terms in the resulting inequality, we get: $|N_x(0,k)| < (1+\epsilon)^{k-1}$.

Node potentials and energy. In order to bound the amortized update time, we introduce the notions of potentials and energy of nodes. Every node $x \in V$ stores nonnegative potentials $\Phi^{\uparrow}(x,k)$, $\Phi^{\downarrow}(x,k)$ and energies $\mathcal{E}^{\uparrow}(x,k)$, $\mathcal{E}^{\downarrow}(x,k)$ at every level $0 \leq k \leq L$. The potential $\Phi^{\uparrow}(x,k)$ and the energy $\mathcal{E}^{\uparrow}(x,k)$ are used to account for the time spent in moving the node x *up* from level k to level $k+1$. Similarly, the potential $\Phi^{\downarrow}(x,k)$ and the energy $\mathcal{E}^{\downarrow}(x,k)$ are used to account for the time spent in moving the node x *down* from level k to level $k-1$. Each unit of potential at level k results in $(1+\epsilon)^k \cdot \epsilon^{-1}$ units of energy. Accordingly, we refer to this quantity $(1+\epsilon)^k \cdot \epsilon^{-1}$ as the *conversion rate* between potential and energy at level k.

For all $x \in V$, all $k \in [0,L]$, and all $\gamma \in \{\uparrow,\downarrow\}$ we have,

$$\mathcal{E}^{\gamma}(x,k) = \Phi^{\gamma}(x,k) \cdot (1+\epsilon)^k \cdot \epsilon^{-1} \tag{4.10}$$

The potentials stored by a node x across different levels depends on its weight W_x. To be more specific, it depends on whether $W_x > 1$ or $W_x \leq 1$. We first define the potentials stored by a node x with weight $W_x > 1$. Throughout the following discussion, we crucially rely upon the fact that $W_{x\to k}$ is a monotonically (weakly) decreasing function of k. For any node $x \in V$ with $W_x > 1$, let $\ell^{\uparrow}(x)$ be the maximum level $k \in \{\ell(x),\ldots,L\}$ where $W_{x\to k} \geq 1$. The potentials $\Phi^{\uparrow}(x,k), \Phi^{\downarrow}(x,k)$ are then defined as follows.

If a node x has $W_x > 1$, then

$$\Phi^{\uparrow}(x,k) = \begin{cases} 0, & \text{for all } \ell^{\uparrow}(x) < k \leq L; \\ W_{x\to k} - 1, & \text{for } k = \ell^{\uparrow}(x); \\ W_{x\to k} - W_{x\to k+1}, & \text{for all } \ell(x) \leq k < \ell^{\uparrow}(x); \\ 0, & \text{for all } 0 \leq k < \ell(x). \end{cases} \tag{4.11}$$

$$\Phi^{\downarrow}(x,k) = 0 \text{ for all } 0 \leq k \leq L. \tag{4.12}$$

It is easy to check that under (4.11) we have $\Phi^{\uparrow}(x,k) \geq 0$ at every level k. Summing over all the levels, the total potential associated with this node x is given by: $\sum_{k=1}^{L} \Phi^{\uparrow}(x,k) = W_{x\to\ell(x)} - 1 = W_x - 1$.

We now give some intuitions behind (4.11) and (4.12). Although these equations might seem daunting at first glance, they in fact follow quite naturally from Claim 7. To see this, first note that as long as $W_x > 1$, the node x never has to decrease its level by moving downward. Hence, if $W_x > 1$, then it is natural to set $\Phi^{\downarrow}(x,k) = 0$ at evert level k. Next, consider the "interesting" scenario when the node x is moving up from its current level $\ell(x) = i$ to level $i+1$. According to step 2 in Figure 1, this means that the node x is up-dirty at level i. From the new definition of a up-dirty node introduced in this section, it follows that $W_{x\to i} \geq W_{x\to i+1} > 1$. Just *before* the node x moves up from level i, we have $\Phi^{\uparrow}(x,i) = W_{x\to i} - W_{x\to i+1}$, and just *after* the node x moves to level $i+1$ we have $\Phi^{\uparrow}(x,i) = 0$. Accordingly, we say that the node x *releases* $(W_{x\to i} - W_{x\to i+1})$ units of potential at level i during this event. As per our conversion ratio between potential and energy defined in (4.10), the node x also releases $(W_{x\to i} - W_{x\to i+1}) \cdot (1+\epsilon)^i \cdot \epsilon^{-1}$ units of energy during this event. Claim 7 now implies that the time spent in updating the data structures during this event is at most the energy released by the node x during the same event. Note that this event does not affect the potentials of the node x at any other level $j \neq i$.

Below, we define the potentials of a node x with weight $W_x \leq 1$.

$$\Phi^{\downarrow}(x,k) = \begin{cases} 0 & \text{for all } \ell(x) < k \leq L; \\ 1 - W_{x\to k} & \text{for } k = \ell(x); \\ 0 & \text{for all } 0 \leq k < \ell(x). \end{cases} \tag{4.13}$$

$$\Phi^{\uparrow}(x,k) = 0 \quad \text{for all } 0 \leq k \leq L. \tag{4.14}$$

As one might expect, the above equations should be seen as naturally following from Claim 8. To see this, first note that the node x does not need to move up from its current level as long as $W_x \leq 1$. Hence, if $W_x \leq 1$, then it makes sense

to define $\Phi^{\uparrow}(x,k) = 0$ at every level $0 \leq k \leq L$. Next, consider the event where the node x is moving down from level i to level $i-1$. Then step 4 in Figure 1 ensures that the node x is down-dirty at level i, so that $W_{x\to i} < (1+\epsilon)^{-1}$. Thus, we have $\Phi^{\downarrow}(x,i) = 1 - W_{x\to i} > \epsilon \cdot (1+\epsilon)^{-1}$ just *before* the event during which the node x moves down from level i to level $i-1$, whereas we have $\Phi^{\downarrow}(x,i) = 0$ just *after* the same event. Accordingly, we say that the node x *releases* at least $\epsilon \cdot (1+\epsilon)^{-1}$ units of potential at level k during this event. As per the conversion ratio between potential and energy defined in (4.10), the node x also releases at least $\epsilon \cdot (1+\epsilon)^{-1} \cdot (1+\epsilon)^{i} \cdot \epsilon^{-1} = (1+\epsilon)^{i-1}$ units of energy at level i during this event. On the other hand, Claim 8 states that the time spent in updating the data structures during this event is at most $O\left((1+\epsilon)^{i-1}\right)$. So the time spent during this event is at most the energy released by the node x at level i. However, in contrast with the discussion following (4.11) and (4.12), here $1 - W_{x\to i-1}$ units of potential and $(1 - W_{x\to i-1}) \cdot (1+\epsilon)^{i-1} \cdot \epsilon^{-1}$ are *created* when the node x moves down to level $i-1$. The energy released by the node at level i only accounts for the time spent in updating the data structures. We need to delve into a deeper analysis of the entire framework in order to bound this new energy that gets created *as a result of* moving the node x down to a lower level, without which our proof for the bound on the amortized update time will remain incomplete. Before embarking on this task, however, we formally clarify the way we are going to use two phrases: *potential (resp. energy) absorbed by a node*, and *potential (resp. energy) released by a node*. This is explained below.

Fix a node $x \in V$ and a level $k \in [0, L]$. Consider an event which (possibly) changes the potentials $\Phi^{\uparrow}(x,k)$ and $\Phi^{\downarrow}(x,k)$. Let $\Phi^{\uparrow}_0(x,k)$ and $\Phi^{\uparrow}_1(x,k)$ respectively denote the value of $\Phi^{\uparrow}(x,k)$ before and after this event. Let $\Delta^{\uparrow} = \Phi^{\uparrow}_1(x,k) - \Phi^{\uparrow}_0(x,k)$. Similarly, let $\Phi^{\downarrow}_0(x,k)$ and $\Phi^{\downarrow}_1(x,k)$ respectively denote the value of $\Phi^{\downarrow}(x,k)$ before and after this event. Let $\Delta^{\downarrow} = \Phi^{\downarrow}_1(x,k) - \Phi^{\downarrow}_0(x,k)$. We now consider four cases.

Case 1. $\Delta^{\uparrow} \geq 0$ *and* $\Delta^{\downarrow} \geq 0$. In this case, we say that during this event the node x *absorbs* $(\Delta^{\uparrow} + \Delta^{\downarrow})$ units of potentials at level k.

Case 2. $\Delta^{\uparrow} < 0$ *and* $\Delta^{\downarrow} < 0$. In this case, we say that during this event the node x *releases* $-(\Delta^{\uparrow} + \Delta^{\downarrow})$ units of potential at level k.

Case 3. $\Delta^{\uparrow} \geq 0$ *and* $\Delta^{\downarrow} < 0$. In this case, we say that during this event the node x *absorbs* $\Delta^{\uparrow}$ units of potential at level k and *releases* $-\Delta^{\downarrow}$ units of potential at level k.

Case 4. $\Delta^{\uparrow} < 0$ *and* $\Delta^{\downarrow} \geq 0$. In this case, we say that during this event the node x *releases* $-\Delta^{\uparrow}$ units of potential at level k and *absorbs* $\Delta^{\downarrow}$ units of potential at level k.

In all the above four cases, the *energy* absorbed (resp. released) by the node x at level k is equal to $(1+\epsilon)^{k} \cdot \epsilon^{-1}$ times the potential absorbed (resp. released) by the node x at level k. Thus, as a matter of convention, we never allow the potential (resp. energy) released or absorbed by a node to be negative. Furthermore, during any given event, we define the potential (resp. energy) absorbed by a node x to be the sum of the potentials (resp. energies) absorbed by x at all the levels $0 \leq k \leq L$. Similarly, the potential (resp. energy) released by x is defined to be the sum of the potentials (resp. energies) released by x at all levels $0 \leq k \leq L$. From the discussion following (4.11) – (4.12) and (4.13) – (4.14), we get the following lemma.

LEMMA 4.1. *Consider an iteration of the* WHILE *loop in Figure 1 where a node x changes its level. During this iteration, the time spent in updating the data structures is at most the energy released by the node x.*

Our main result is summarized in the theorem below.

THEOREM 4.1. *Starting from an empty graph, our algorithm spends $O(\tau/\epsilon^2)$ total time to handle any sequence of τ updates (edge insertions/deletions) in G. This implies an amortized update time of $O(1/\epsilon^2)$.*

We devote the rest of this section towarding giving a high level overview of the proof of the above theorem. We begin with the crucial observation that according to Lemma 4.1, the energy released by the nodes is an upper bound on the total update time of our algorithm. Hence, in order to prove Theorem 4.1, it suffices to upper bound the total energy released by the nodes during the sequence of τ updates. Note that the total energy stored at the nodes is zero when the input graph is empty. Furthermore, the node-potentials (and energies) are always nonnegative. Thus, during the course of our algorithm the total energy released by the nodes is at most the total energy absorbed by the nodes, and hence it suffices to upper bound the latter quantity. We will show that overall the nodes absorb $O(\tau/\epsilon^2)$ units of energy while our algorithm handles τ updates starting from an empty graph. This implies Theorem 4.1.

Note that the nodes might absorb energy under two possible scenarios:

- (a) An edge (u,v) gets inserted into or deleted from the graph. Under this scenario, one or both the endpoints $\{u,v\}$ might absorb some energy. No node, however, changes its level under this scenario.
- (b) The subroutine in Figure 1 is called after the insertion or deletion of an edge (scenario (a)), and a node x moves up or down one level during an iteration of the WHILE loop in Figure 1. Under this scenario (b), one or more nodes in $N_x \cup \{x\}$ might absorb some energy.

Theorem 4.1 now follows from Claim 9 and Claim 10 stated below. In the remainder of this section, we will present high-level, intuitive justifications for each of these claims.

CLAIM 9. *The total energy absorbed by the nodes under scenario (a) is at most $O(\tau/\epsilon)$.*

CLAIM 10. *The total energy absorbed by the nodes under scenario (b) is at most $O(\tau/\epsilon^2)$.*

4.2.1 Justifying Claim 9 Consider an *event* where an edge (u, v) gets inserted into or deleted from the graph. This can change the potentials of only the endpoints u, v, and hence only u and v can absorb energy during such an event. Below, we show that the total energy absorbed by the two endpoints is at most $O(1/\epsilon)$. Since τ is the total number of edge insertions or deletions that take place in the graph G, this implies Claim 9.

Edge-Deletion. First, we focus on analyzing an edge-deletion. Specifically, suppose that an edge (u, v) with $\ell(u) = i \geq \ell(v) = j$ gets deleted from the graph. Consider the endpoint u. Due to this event (where the edge (u, v) gets deleted), the weight W_u decreases by $(1+\epsilon)^{-i}$. From (4.13) and (4.12) we infer that the value of $\Phi^{\downarrow}(u, i)$ can increase by at most $(1 + \epsilon)^{-i}$ during this event, whereas the value of $\Phi^{\downarrow}(u, k)$ remains equal to 0 for all $k \neq i$. In contrast, from (4.11) and (4.14) we infer that for all $k \in [0, L]$, the value of $\Phi^{\uparrow}(u, k)$ can never increase during this event. This is because for each level $k \in [i, L]$, the weight $W_{u\to k}$ decreases by $(1 + \epsilon)^{-i}$ due to this event. In other words, the node u can only absorb at most $(1 + \epsilon)^{-i}$ units of potential during this event, and that too only at level i. Hence, the energy absorbed by the node u during this event is at most $(1 + \epsilon)^{-i} \cdot (1 + \epsilon)^{i} \cdot \epsilon^{-1} = \epsilon^{-1}$. Applying a similar argument for the other endpoint v, we conclude that at most $2\epsilon^{-1} = O(1/\epsilon)$ units of energy can get absorbed due to the deletion of an edge.

Edge-Insertion. Next, we focus on the scenario where an edge (u, v) gets inserted into the graph. A formal proof for this scenario is a bit involved. To highlight the main idea, we only consider one representative scenario in this section, as described below.

Suppose that $\ell(u) = i \geq \ell(v) = j$ and $W_u > 1$ just before the insertion of the edge (u, v). We want to show that the node u absorbs at most $O(1/\epsilon)$ units of energy during this event (insertion of the edge (u, v)).

The key observation here is that just before the event the node u was not up-dirty. To be more specific, just before the event we had $W_{u\to i} > 1$ and $W_{u\to i+1} \leq 1$. This follows from the discussion on "approximation ratio" in Section 4.1. This implies that there was at least one edge $(u, x) \in E$ with $\ell(x) \leq i$ just before the event, for otherwise we would have $W_{u\to i+1} = W_{u\to i} > 1$. Let i' be the value of $\ell^{\uparrow}(u)$ just after the event. Now, note that for every level $i \leq k \leq i'$, the value of $W_{u\to k}$ increases by $(1 + \epsilon)^{-k} \leq (1 + \epsilon)^{-i}$ during this event. Hence, from (4.11) we conclude that the node u absorbs at most $(1 + \epsilon)^{-i}$ units of potential at each level $k \in [i, i']$. Thus, the total energy absorbed by the node u is at most $\sum_{k=i}^{i'} (1+\epsilon)^{-i} \cdot (1+\epsilon)^{k} \cdot \epsilon^{-1} \leq \epsilon^{-1} \cdot (1+\epsilon)^{i'-i+1}$. To complete the proof, below we show that $(1+\epsilon)^{i'-i} = O(1)$, which implies that the node u absorbs at most $O(1/\epsilon)$ units of energy during this event.

Just before the event, we had $W_{u\to i+1} \leq 1$. At that time, consider a thought experiment where we move up the node u to level i'. During that process, as we move up from level $i+1$ to level i', the weight of the edge (u, x) decreases by $(1 + \epsilon)^{-(i+1)} - (1 + \epsilon)^{-i'}$. Hence, we get: $W_{u\to i'} \leq 1-(1+\epsilon)^{-(i+1)}+(1+\epsilon)^{-i'}$ just before the event. Insertion of the edge (u, v) increases the weight $W_{u\to i'}$ by $(1 + \epsilon)^{-i'}$. Hence, we infer that $W_{u\to i'} \leq 1-(1+\epsilon)^{-(i+1)}+2\cdot(1+\epsilon)^{-i'}$ just after the event. Recall that i' is the value of $\ell^{\uparrow}(u)$ just after the event. Thus, by definition, we have: $W_{u\to i'} > 1$. Combining the last two inequalities, we get:

$$1 < 1 - (1 + \epsilon)^{-(i+1)} + 2 \cdot (1 + \epsilon)^{-i'},$$

which implies that

$$(1 + \epsilon)^{-(i+1)} < 2 \cdot (1 + \epsilon)^{-i'}.$$

Rearranging the terms in the last inequality, we get: $(1 + \epsilon)^{i'-i} < 2(1 + \epsilon) = O(1)$, as promised.

A note on the gap between $\ell(u)$ and $\ell^{\uparrow}(u)$. The above argument relies upon the following property: Since the node u was not up-dirty before the insertion of the edge (u, v), the level $\ell^{\uparrow}(u)$ cannot be too far away from the level $\ell(u)$ just after the insertion of the edge (u, v). This property, however, might no longer be true once we call the subroutine in Figure 1. This is because by the time we deal with a specific up-dirty node x, a lot of its neighbors might have changed their levels (thereby significantly changing the weight W_x).

4.2.2 Justifying Claim 10 We now give a high-level, intuitive justification for Claim 10, which bounds the total energy absorbed by all the nodes under scenario (b). See the discussion following the statement of Theorem 4.1. We first classify the node-potentials into certain *types*, depending on the level the potential is stored at, *and* whether the potential will be used to account for the time spent in moving the node up or down from that level. Accordingly, for every level $k \in [0, L]$, we define:

$$\Phi^{\uparrow}(k) = \sum_{x \in V} \Phi^{\uparrow}(x, k) \text{ and } \Phi^{\downarrow}(k) = \sum_{x \in V} \Phi^{\downarrow}(x, k).$$

We say that there are $\Phi^{\uparrow}(k)$ units of potential in the system that are of *type* $(k, \uparrow)$, and there are $\Phi^{\downarrow}(k)$ units of potential

in the system that are of *type* $(k, \downarrow)$. Overall, there are $2(L+1)$ different types of potentials, since we can construct $2(L+1)$ many ordered pairs of the form (k, α), with $0 \leq k \leq L$ and $\alpha \in \{\uparrow, \downarrow\}$.

Let $\Gamma = \{(k, \alpha) : \alpha \in \{\uparrow, \downarrow\} \text{ and } 0 \leq k \leq L\}$ denote the set of all possible types of potentials. We define a total order $\succ$ on the elements of the set Γ as follows. For any two types of potentials $(k, \alpha), (k', \alpha') \in \Gamma$, we have $(k, \alpha) \succ (k', \alpha)$ iff either $\{k > k'\}$ or $\{k = k', \alpha = \uparrow, \text{ and } \alpha' = \downarrow\}$. Next, from (**??**) and (**??**), we recall that the *conversion rate* between energy and potential is $(1+\epsilon)^k \cdot \epsilon^{-1}$ at level k. In other words, from δ units of potential stored at level k we get $\delta \cdot (1+\epsilon)^k \cdot \epsilon^{-1}$ units of energy. Keeping this in mind, we define the *conversion rate* associated with both the types $(k, \uparrow)$ and $(k, \downarrow)$ to be $(1+\epsilon)^k \cdot \epsilon^{-1}$. Specifically, we write $c_{(k,\uparrow)} = c_{(k,\downarrow)} = (1+\epsilon)^k \cdot \epsilon^{-1}$. Thus, from δ units of potential of any type $\gamma \in \Gamma$, we get $\delta \cdot c_\gamma$ units of energy. The total order $\succ$ we defined on the set Γ has the following properties.

PROPERTY 4.1. *Consider any three types of potentials $\gamma_1, \gamma_2, \gamma_3 \in \Gamma$ such that $\gamma_1 \succ \gamma_2 \succ \gamma_3$. Then we must have $c_{\gamma_1} \geq (1+\epsilon) \cdot c_{\gamma_3}$. In words, the conversion rate between energy and potentials drops by at least a factor of $(1+\epsilon)$ as we move two hops down the total order Γ.*

Proof. Any $\gamma \in \Gamma$ is of the form (k, α) where $k \in \{0, \ldots, L\}$ and $\alpha \in \{\uparrow, \downarrow\}$. From the way we have defined the total order $\succ$, it follows that if $(k_1, \alpha_1) \succ (k_2, \alpha_2) \succ (k_3, \alpha_3)$, then $k_1 \geq k_3 + 1$. The property holds since $c_{(k_1,\alpha_1)} = (1+\epsilon)^{k_1}$ and $c_{(k_3,\alpha_3)} = (1+\epsilon)^{k_3}$.

PROPERTY 4.2. *Consider an event where some nonzero units of potential are absorbed by some nodes, under scenario (b). Such an event occurs only if some node x moves up or down one level from its current level k (say). In the former case let $\gamma^* = (k, \uparrow)$, and in the latter case let $\gamma^* = (k, \downarrow)$. Let $\delta^* \geq 0$ denote the potential of type γ^** released *by x during this event. For every type $\gamma \in \Gamma$, let $\delta_\gamma \geq 0$ denote the total potential of type γ* absorbed *by all the nodes during this event. Then: (1) For every $\gamma \in \Gamma$, we have $\delta_\gamma > 0$ only if $\gamma^* \succ \gamma$. (2) We also have $\sum_{\gamma \in \Gamma} \delta_\gamma \leq \delta^*$.*

Proof. (Sketch) A formal proof of Property 4.2 is quite involved. Instead, here we present a very high-level intuition behind the proof. Consider an event where an up-dirty node x with weight $W_x > 1$ moves up (say) from level k to level $k+1$. From (4.11), we infer that the node x releases $\delta^* = W_{x \to k} - W_{x \to k+1}$ units of potential, and the released potential is of type $(k, \uparrow)$.

Next, we observe that the weight W_x also decreases exactly by δ^* during this event. Now, during the same event, some neighbors y of x might decrease their weights W_y, and these are also the nodes that might absorb some nonzero units of potentials. We note that the weight of an edge (x, y) decreases only if $y \in N_x(0, k)$, and the sum of these weight-decreases is equal to δ^*. Thus, a neighbor y of x absorbs some potential only if $\ell(y) \leq k$, and the sum of these absorbed potentials is at most δ^*. From (4.11), (4.12), (4.13) and (4.14), we also conclude that if a node y absorbs potential when its weight decreases, then the absorbed potential must be of type $(\ell(y), \downarrow)$ where $\ell(y) \leq k$. To summarize, the node x releases δ^* units of potential of type $(k, \uparrow)$, and at most δ^* units of potential are overall absorbed by all the nodes during this event. Furthermore, if a nonzero amount of potential of some type γ gets absorbed, then we must have $\gamma = (k', \downarrow)$ for some $k' \leq k$, and hence $(k, \uparrow) \succ \gamma$.

A similar argument applies when the node x moves down from level k to level $k-1$.

Properties 4.1 and 4.2 together give us a complete picture of the way the potential stored by the nodes *flows within the system*. Specifically, there are two scenarios in which potential can be pumped into (i.e., absorbed by) the nodes. In scenario (a), potential gets pumped into the nodes *exogenously* by an adversary, due to the insertion or deletion of an edge in the graph. But, according to Claim 9, the total energy absorbed by the nodes under this scenario is already upper bounded by $O(\tau/\epsilon)$. On the other hand, in scenario (b), a node releases some $\delta^* \geq 0$ units of potential of type $\gamma^* \in \Gamma$ (say), and at the same time some $0 \leq \delta \leq \delta^*$ units of potential get created. This newly created δ units of potential are then *split up* in some *chunks*, and these chunks in turn get absorbed as potentials of (one or more) different types γ. We now note three key points about this process: (1) $\delta \leq \delta^*$. (2) If a chunk of this newly created δ units of potential gets absorbed as potential of type γ, then we must have $\gamma^* \succ \gamma$. (3) By Property 4.1, as we move down two hops in the total order $\succ$, the conversion rate between energy and potential *drops* at least by a factor of $(1+\epsilon)$. These three points together imply that the energy absorbed under scenario (b) is at most β times the energy absorbed under scenario (a), where $\beta = 2 + 2 \cdot (1+\epsilon)^{-1} + 2 \cdot (1+\epsilon)^{-2} + \cdots = O(1/\epsilon)$. Hence, from Claim 9 we infer that the total energy absorbed by the nodes under scenario (b) is at most $O(1/\epsilon) \cdot O(\tau/\epsilon) = O(\tau/\epsilon^2)$. This concludes the proof of Claim 10.

5 Bibliography

References

[1] Raghavendra Addanki and Barna Saha. Fully dynamic set cover - improved and simple. *CoRR*, abs/1804.03197, 2018.

[2] Moab Arar, Shiri Chechik, Sarel Cohen, Cliff Stein, and David Wajc. Dynamic matching: Reducing integral algorithms to approximately-maximal fractional algorithms. In *ICALP 2018*.

[3] S. Baswana, M. Gupta, and S. Sen. Fully dynamic maximal matching in $O(\log n)$ update time. In *FOCS 2011*.

[4] Aaron Bernstein and Cliff Stein. Faster fully dynamic matchings with small approximation ratios. In *SODA 2016*.

[5] Aaron Bernstein and Cliff Stein. Fully dynamic matching in bipartite graphs. In *ICALP 2015*.

[6] Sayan Bhattacharya, Deeparnab Chakrabarty, and Monika Henzinger. Deterministic fully dynamic approximate vertex cover and fractional matching in O(1) amortized update time. In *IPCO 2017*.

[7] Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. Design of dynamic algorithms via primal-dual method. In *ICALP 2015*.

[8] Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. Deterministic fully dynamic data structures for vertex cover and matching. In *SODA 2015*.

[9] Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. Fully dynamic approximate maximum matching and minimum vertex cover in $O(\log^3 n)$ worst case update time. In *SODA 2017*.

[10] Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. New deterministic approximation algorithms for fully dynamic matching. In *STOC 2016*.

[11] Sayan Bhattacharya and Janardhan Kulkarni. Deterministically maintaining a $(2+\epsilon)$-approximate minimum vertex cover in $o(1/\epsilon^2)$ amortized update time. In *arXiv link: https://arxiv.org/abs/1805.03498*.

[12] Moses Charikar and Shay Solomon. Fully dynamic almost-maximal matching: Breaking the polynomial barrier for worst-case time bounds. In *ICALP 2018*.

[13] Anupam Gupta, Ravishankar Krishnaswamy, Amit Kumar, and Debmalya Panigrahi. Online and dynamic algorithms for set cover. In *STOC 2017*.

[14] Manoj Gupta and Richard Peng. Fully dynamic $(1+\epsilon)$-approximate matchings. In *FOCS 2013*.

[15] Subhash Khot and Oded Regev. Vertex cover might be hard to approximate to within $2-\epsilon$. In *CCC 2003*.

[16] Tsvi Kopelowitz, Robert Krauthgamer, Ely Porat, and Shay Solomon. Orienting fully dynamic graphs with worst-case time bounds. In *ICALP 2014*.

[17] Ofer Neiman and Shay Solomon. Simple deterministic algorithms for fully dynamic maximal matching. In *STOC 2013*.

[18] Krzysztof Onak and Ronitt Rubinfeld. Maintaining a large matching and a small vertex cover. In *STOC 2010*.

[19] David Peleg and Shay Solomon. Dynamic (1+ϵ)-approximate matchings: A density-sensitive approach. In *SODA 2016*.

[20] Shay Solomon. Fully dynamic maximal matching in constant update time. In *FOCS 2016*.