

# A Deamortization Approach for Dynamic Spanner and Dynamic Maximal Matching\*

Aaron Bernstein<sup>†</sup>

Sebastian Forster<sup>‡</sup>

Monika Henzinger<sup>§</sup>

## Abstract

Many dynamic graph algorithms have an *amortized* update time, rather than a stronger *worst-case* guarantee. But amortized data structures are not suitable for real-time systems, where each individual operation has to be executed quickly. For this reason, there exist many recent randomized results that aim to provide a guarantee stronger than amortized expected. The strongest possible guarantee for a randomized algorithm is that it is always correct (Las Vegas), and has *high-probability worst-case* update time, which gives a bound on the time for each individual operation that holds with high probability.

In this paper we present the first polylogarithmic high-probability worst-case time bounds for the dynamic spanner and the dynamic maximal matching problem.

1. For dynamic spanner, the only known  $o(n)$  worst-case bounds were  $O(n^{3/4})$  high-probability worst-case update time for maintaining a 3-spanner and  $O(n^{5/9})$  for maintaining a 5-spanner. We give a  $O(1)^k \log^3(n)$  high-probability worst-case time bound for maintaining a  $(2k - 1)$ -spanner, which yields the first worst-case polylog update time for all constant  $k$ . (All the results above maintain the optimal tradeoff of stretch  $2k - 1$  and  $\tilde{O}(n^{1+1/k})$  edges.)
2. For dynamic *maximal* matching, or dynamic 2-approximate maximum matching, no algorithm with  $o(n)$  worst-case time bound was known and we present an algorithm with  $O(\log^5(n))$  high-probability worst-case time; similar worst-case bounds existed only for maintaining a matching that was  $(2 + \epsilon)$ -approximate, and hence not maximal.

Our results are achieved using a new approach for converting amortized guarantees to worst-case ones for randomized data structures by going through a third type of guarantee, which is a middle ground between the two above: an algorithm is said to have *worst-case expected* update time  $\alpha$  if for *every* update  $\sigma$ , the expected time to process  $\sigma$  is at most  $\alpha$ . Although stronger than amortized expected, the worst-case expected guarantee does not resolve the fundamental problem of amortization: a worst-case expected update time of  $O(1)$  still allows for the possibility that every  $1/f(n)$  updates requires  $\Theta(f(n))$  time to process, for arbitrarily high  $f(n)$ . In this paper we present a *black-box* reduction that converts any data structure with worst-case expected update time into one with a high-probability worst-case update time: the query time remains the same, while the update time increases by a factor of  $O(\log^2(n))$ .

---

\*To be presented at the 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2019).

<sup>†</sup>Rutgers University, USA. Work done in part while at TU Berlin and while visiting University of Vienna.

<sup>‡</sup>University of Salzburg, Department of Computer Sciences, Austria. Work done in part while at University of Vienna. This author previously published under the name Sebastian Krinninger.

<sup>§</sup>University of Vienna, Faculty of Computer Science, Austria. The research leading to these results has received funding from the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement No. 340506.

Thus we achieve our results in two steps: (1) First we show how to convert existing dynamic graph algorithms with *amortized* expected polylogarithmic running times into algorithms with *worst-case expected* polylogarithmic running times. (2) Then we use our black-box reduction to achieve the polylogarithmic high-probability worst-case time bound. All our algorithms are Las-Vegas-type algorithms.

## 1 Introduction

A *dynamic graph algorithm* is a data structure that maintains information in a graph that is being modified by a sequence of edge insertion and deletion operations. For a variety of graph properties there exist dynamic graph algorithms for which amortized expected time bounds are known and the main challenge is to de-amortize and de-randomize these results. Our paper addresses the first challenge: de-amortizing dynamic data structures.

An amortized algorithm guarantees a small average update time for a “large enough” sequence of operations: dividing the total time for  $T$  operations by  $T$  leads to the *amortized time* per operation. If the dynamic graph algorithm is randomized, then the *expected total time* for a sequence of operations is analyzed, giving a bound on the *amortized expected time* per operation. But in real-time systems, where each individual operation has to be executed quickly, we need a stronger guarantee than amortized expected time for randomized algorithms. The strongest possible guarantee for a randomized algorithm is that it is always correct (Las Vegas), and has *high-probability worst-case* update time, which gives an upper bound on the time for *every* individual operation that holds with high probability. (The probability that the time bound is not achieved should be polynomially small in the problem size.) There are many recent results which provide randomized data structures with worst-case guarantees (see e.g. [San04, KKM13, Gib<sup>+</sup>15, Abr<sup>+</sup>16, BK16, ACK17, NSW17, CS18, Ara<sup>+</sup>18]), often via a complex “deamortization” of previous results.

In this paper we present the first algorithms with worst-case polylog update time for two classical problems in the dynamic setting: dynamic spanner, and dynamic maximal matching. In both cases, polylog *amortized* results were already known, but the best worst-case results required polynomial update time.

Both results are based on a new de-amortization approach for randomized dynamic graph algorithms. We bring attention to a third possible type of guarantee: an algorithm is said to have *worst-case expected* update time  $\alpha$  if for *every* update  $\sigma$ , the expected time to process  $\sigma$  is at most  $\alpha$ . On it’s own this guarantee does not resolve the fundamental problem of amortization, since a worst-case expected update time of  $O(1)$  still allows for the possibility that every  $1/f(n)$  updates requires  $\Theta(f(n))$  time to process, for arbitrarily high  $f(n)$ . But by using some relatively simple probabilistic bootstrapping techniques, we show a *black-box* reduction that converts any algorithm with a worst-case expected update time into one with a high-probability worst-case update time.

This leads to the following deamortization approach: rather than directly aiming for high-probability worst-case, first aim for the weaker worst-case expected guarantee, and then apply the black-box reduction. Achieving such a worst-case expected bound can involve serious technical challenges, in part because one cannot rely on the standard charging arguments used in amortized analysis. We nonetheless show how to achieve such a guarantee for both dynamic spanner and dynamic maximal matching, which leads to our improved results for both problems.

**Details of the New Reduction.** We show a black-box conversion of an algorithm with worst-case expected update time into one with worst-case high-probability update time: the worst-case

query time remains the same, while the update time increases by a  $\log^2(n)$  factor. Our reduction is quite general, but with our applications to dynamic graph algorithms in mind, we restrict ourselves to dynamic data structures that support only two types of operations: (1) *update* operations, which manipulate the internal state of the data structure, but do not return any information, and (2) *query* operations, which return information about the internal state of the data structure, but do not manipulate it. We say the data structure has *update time*  $\alpha$  if the maximum update time of any type of update (e.g. insertion or deletion) is  $\alpha$ .

**Theorem 1.1.** *Let  $A$  be an algorithm that maintains a dynamic data structure  $D$  with worst-case expected update time  $\alpha$  for each update operation and let  $n$  be a parameter such that the maximum number of items stored in the data structure at any point in time is polynomial in  $n$ . We assume that for any set of elements  $S$  such that  $|S|$  is polynomial in  $n$ , a new version of the data structure  $D$  containing exactly the elements of  $S$  can be constructed in polynomial time. If this assumption holds, then there exists an algorithm  $A'$  with the following properties:*

1. *For any sequence of updates  $\sigma_1, \sigma_2, \dots$ ,  $A'$  processes each update  $\sigma_i$  in  $O(\alpha \log^2(n))$  time with high probability. The amortized expected update time of  $A'$  is  $O(\alpha \log(n))$ .*
2.  *$A'$  maintains  $\Theta(\log(n))$  data structures  $D_1, D_2, \dots, D_{\Theta(\log(n))}$ , as well as a pointer to some  $D_i$  that is guaranteed to be correct at the current time. Query operations are answered with  $D_i$ .*

The theorem applies to any dynamic data structure, but we will apply it to dynamic graph algorithms. Due to its generality, however, we expect that the theorem will prove useful in other settings as well. When applied to a dynamic graph algorithm,  $n$  denotes the number of vertices, and at most  $n^2$  elements (the edges) are stored at any point in time. Note that our assumption about polynomial preprocessing time for any polynomial-size set of elements  $S$  is satisfied by the vast majority of data structures, and is in particular satisfied by all dynamic graph algorithms that we know of.

Observe that a high-probability worst-case update time bound of  $O(\alpha \log^2(n))$  allows us to stop the algorithm whenever its update time exceeds the  $O(\alpha \log^2(n))$  bound and in this way obtain an algorithm that is correct with high probability.

*Remark 1.2.* By Item 2, the converted algorithm  $A'$  stores a slightly different data structure than the original algorithm  $A$ , because it maintains  $O(\log(n))$  copies  $D_i$  of the data structure in  $A$ . The data structure in  $A'$  is equally powerful to that in  $A$  because it can answer all the same queries in the same asymptotic time:  $A'$  always has a pointer to some  $D_i$  that is guaranteed to be fixed, so it can use  $D_i$  to answer queries. The main difference is that the answers produced by  $A'$  may have less “continuity” than those produced by  $A$ : for example, in a dynamic maximal matching algorithm, if each query outputs the entire maximal matching, then a single update may change the pointer in  $A'$  from some  $D_i$  to some  $D_j$ , and  $A'$  will then output a completely different maximal matching before and after the update. Note that this issue does not arise in our dynamic spanner algorithm, as the spanner is formed by the union of the spanners of all copies.

**First Result: Dynamic Spanner Maintenance.** Given a graph  $G$ , a spanner  $H$  with stretch  $\alpha$  is a subgraph of  $G$  such that for any pair of vertices  $(u, v)$ , the distance between  $u$  and  $v$  in  $H$  is at most an  $\alpha$  factor larger than their distance in  $G$ . In the dynamic spanner problem the main goal is to maintain, for any given integer  $k \geq 2$ , a spanner of stretch  $2k - 1$  with  $\tilde{O}(n^{1+1/k})$  edges; we focus on these particular bounds because spanners of stretch  $2k - 1$  and  $O(n^{1+1/k})$  edges exist for every

graph [Awe85], and this trade-off is presumed tight under Erdős’s girth conjecture. The dynamic spanner problem was introduced by Ausiello, Franciosa, and Italiano [AFI06] and has been improved upon by [Elk11, BKS12, BK16]. There currently exist near-optimal amortized expected bounds: a  $(2k - 1)$ -spanners can be maintained with expected amortized update time  $O(1)^k$  [BKS12] or time  $O(k^2 \log^2(n))$  [GK18]. The state-of-the-art for high-probability worst-case lags far behind:  $O(n^{3/4})$  update time for maintaining a 3-spanner, and  $O(n^{5/9})$  for a 5-spanner [BK16]; no  $o(n)$  worst-case update time was known for larger  $k$ . All of these algorithms exhibit the stretch/space trade-off mentioned above, up to polylogarithmic factors in the size of the spanner<sup>1</sup>.

We give the first dynamic spanner algorithm with polylog worst-case update time for any constant  $k$ , which significantly improves upon the result of [BK16] both in update time and in range of  $k$ . Our starting point is the earlier result of Baswana, Khurana, and Sarkar [BKS12] that maintains a  $(2k - 1)$  spanner with  $O(n^{1+1/k} \log^2(n))$  edges with update time  $O(1)^k$ . We show that while their algorithm is amortized expected, it can be modified to yield worst-case expected bounds: this requires a few changes to the algorithm, as well as significant changes to the analysis. We then apply the reduction in Theorem 1.1.

**Theorem 1.3.** *There exists a fully dynamic (Las Vegas) algorithm for maintaining a  $(2k - 1)$  spanner with  $O(n^{1+1/k} \log^6(n) \log \log(n))$  edges that has worst-case expected update time  $O(1)^k \log(n)$ .*

**Theorem 1.4.** *There exists a fully dynamic (Las Vegas) algorithm for maintaining a  $(2k - 1)$  spanner with  $O(n^{1+1/k} \log^7(n) \log \log(n))$  edges that has high-probability worst-case update time  $O(1)^k \log^3(n)$ .*

The proof follows directly from Theorem 1.3 and Theorem 1.1. In the case of maintaining a spanner, the potential lack of continuity discussed in Remark 1.2 does not exist, as instead of switching between the  $O(\log(n))$  spanners maintained by the conversion in Theorem 1.1, we can just let the final spanner be the union of all of them. This incurs an extra  $\log(n)$  factor in the size of the spanner.

**Second Result: Dynamic Maximal Matching.** A maximum cardinality matching can be maintained dynamically in  $O(n^{1.495})$  amortized expected time per operation [San07]. Due to conditional lower bounds of  $\Omega(\sqrt{m})$  on the time per operation for this problem [AW14, Hen<sup>+</sup>15], there is a large body of work on the dynamic *approximate* matching problem [OR10, BGS18, NS16, GP13, BHI18, BHN16, Sol16, BHN17, BCH17, Gup<sup>+</sup>17, CS18, Ara<sup>+</sup>18]. Still the only algorithms with polylogarithmic (amortized or worst-case) time per operation require a 2 or larger approximation ratio.

A matching is said to be *maximal* if the graph contains no edges between unmatched vertices. A maximal matching is guaranteed to be a 2-approximation of the maximum matching, and is also a well-studied object in its own right (see e.g. [HKP01, GKP08, Lat<sup>+</sup>11, BGS18, NS16, Sol16, Fis17]). The groundbreaking result of Baswana, Gupta, and Sen [BGS18] showed how to maintain a maximal matching (and so 2-approximation) with  $O(\log(n))$  expected amortized update time. Solomon [Sol16] improved the update time to  $O(1)$  expected amortized. There has been recent work on either deamortizing or derandomizing this result [BHN16, BCH17, BHN17, CS18, Ara<sup>+</sup>18]. Most notably, the two independent results in [CS18] and [Ara<sup>+</sup>18] both present algorithms with

---

<sup>1</sup>A standard assumption for the analysis of randomized dynamic graph algorithms is that the “adversary” who supplies the sequence of updates is assumed to have fixed this sequence  $\sigma_1, \sigma_2, \dots$  *before* the dynamic algorithm starts to operate, and the random choices of the algorithm then define a distribution on the time to process each  $\sigma_i$ . This is called an *oblivious* adversary. Our dynamic algorithms for spanners and matching share this assumption, as does all prior work.

polylog high-probability worst-case update time that maintain a  $(2 + \epsilon)$ -approximate matching. Unfortunately, all these results comes at the price of increasing the approximation factor from 2 to  $(2 + \epsilon)$ , and in particular no longer ensure that the matching is maximal. One of the central questions in this line of work is thus whether it is possible to maintain a maximal matching without having to use both randomization *and* amortization.

We present the first affirmative answer to this question by removing the amortization requirement, thus resolving an open question of [Ara<sup>+</sup>18]. Much like for dynamic spanner, we use an existing amortized algorithm as our starting point: namely, the  $O(\log(n))$  amortized algorithm of [BGS18]. We then show how the algorithm and analysis can be modified to achieve a worst-case expected guarantee, and then we apply our reduction.

**Theorem 1.5.** *There exists a fully dynamic (Las Vegas) algorithm for maintaining a maximal matching with worst-case expected update time  $O(\log^3(n))$ .*

**Theorem 1.6.** *There exists a fully dynamic (Las Vegas) algorithm that maintains a maximal matching with high-probability worst-case update time  $O(\log^5(n))$ .*

The proof follows directly from Theorem 1.5 and Theorem 1.1. As in Remark 1.2 above, we note that our worst-case algorithm in Theorem 1.6 stores the matching in a different data structure than the original amortized algorithm of Baswana et al. [BGS18]: while the latter stores the edges of the maximal matching in a single list  $D$ , our algorithm stores  $O(\log(n))$  lists  $D_i$ , along with a pointer to some specific list  $D_j$  that is guaranteed to contain the edges of a maximal matching. In particular, the algorithm always knows which  $D_j$  is correct. The pointer to  $D_j$  allows our algorithm to answer queries about the maximal matching in optimal time.

**Discussion of Our Contribution.** We present the first dynamic algorithms with worst-case polylog update times for two classical graph problems: dynamic spanner, and dynamic maximal matching. Both results are achieved with a new de-amortization approach, which shows how the concept of worst-case expected time can be a very fruitful way of thinking about dynamic graph algorithms. From a technical perspective, the conversion from worst-case expected to high-probability worst-case (Theorem 1.1) is relatively simple. The main technical challenge lies in showing how the existing amortized algorithms for dynamic spanner and maximal matching can be modified to be worst-case expected. The changes to the algorithms themselves are not too major, but a very different analysis is required, because we can no longer rely on charging arguments and potential functions. We hope that our tools for proving worst-case expected guarantees can be used to de-amortize other existing dynamic algorithms. For example, the dynamic coloring algorithm of [Bha<sup>+</sup>18], the dynamic spectral sparsifier algorithm of [Abr<sup>+</sup>16], the dynamic distributed maximal independent set algorithm of [CHK16], and the dynamic distributed spanner algorithm of [BKS12] (all amortized) seem like natural candidates for our approach.

Section 2 provides a proof of the back-box reduction in Theorem 1.1. Section 4 presents our dynamic matching algorithm, and Section 3 presents our dynamic spanner algorithm.

## 2 Converting Worst-Case Expected to High-Probability Worst-Case

In this section we give the proof of Theorem 1.1. To do so, we first prove the following theorem that restricts the length of the update sequence and then show how to extend it.

**Theorem 2.1.** *Let  $A$  be an algorithm that maintains a dynamic data structure  $D$  with worst-case expected update time  $\alpha$ , let  $n$  be a parameter such that the maximum number of items stored in the data structure at any point in time is polynomial in  $n$ , and let  $\ell$  be a parameter for the length of the update sequence to be considered. Then there exists an algorithm  $A'$  with the following properties:*

1. *For any sequence of updates  $\sigma_1, \sigma_2, \dots$ ,  $A'$  processes each update  $\sigma_i$  in  $O(\alpha \log^2(n))$  time with high probability. The amortized expected update time of  $A'$  is  $O(\alpha \log(n))$ .*
2.  *$A'$  maintains  $\Theta(\log(n))$  data structures  $D_1, D_2, \dots, D_{\Theta(\log(n))}$ , as well as a pointer to some  $D_i$  that is guaranteed to be correct at the current time. Query operations are answered with  $D_i$ .*

*Proof.* Let  $q = c \log(n)$  for a sufficiently large constant  $c$ . The algorithm runs  $q = \Theta(\log(n))$  versions of the algorithm  $A$ , denoted  $A_1, \dots, A_q$ , each with their own independently chosen random bits. This results in  $q$  data structures  $D_i$ . Each  $D_i$  maintains a possibly empty buffer  $L_i$  of uncompleted updates. If  $L_i$  is empty,  $D_i$  is marked as *fixed*, otherwise as *broken*. The algorithm maintains a list of all the fixed data structures, and a pointer to the  $D_i$  of smallest index that is fixed.

Let  $r = 4\alpha \log(\ell) = O(\alpha \log(\ell))$ . Given an update  $\sigma_j$  the algorithm adds  $\sigma_j$  to the end of each  $L_i$  and then allows each  $A_i$  to run for  $r$  steps. Each  $A_i$  will work on the uncompleted updates in  $L_i$ , continuing where it left off after the last update, and completing the first uncompleted update before starting the next one in the order in which they appear in  $L_i$ . If within these  $r$  steps all uncompleted updates in  $L_i$  have been completed,  $A_i$  marks itself as fixed; otherwise it marks itself as broken. If at the end of update  $\sigma_j$  all of the  $q$  data structures  $D_i$  are broken then the algorithm performs a FLUSH, which simply processes all the updates in all the versions  $A_i$ : this could take much more than  $r$  work, but our analysis will show that this event happens with extremely small probability. The FLUSH ensures Property 2 of Theorem 2.1: at the end of every update, some  $D_i$  is fixed.

By linearity of expectation, the expected amortized update time is  $O(\alpha q) = O(\alpha \log(n))$ , and the worst-case update time is  $r q = O(\alpha \log^2(n))$  unless a FLUSH occurs. All we have left to show is that after every update the probability of a FLUSH is at most  $(1/2)^q = 1/n^c$ . We use the following counter analysis:

**Definition 2.2.** *We define the dynamic counter problem with positive integer parameters  $\alpha$  (for average),  $r$  (for reduction), and  $\ell$  (for length) as follows. Given a finite sequence of possibly dependent random variables  $X_1, X_2, \dots, X_\ell$  such that for each  $t$ ,  $E[X_t] \leq \alpha$ , we define a sequence of counters  $C_t$  which changes over a finite sequence of time steps. Let  $C_0 = 0$  and let  $C_t = \max(X_t + C_{t-1} - r, 0)$ .*

**Lemma 2.3.** *Given a dynamic counter problem with parameters  $\alpha$ ,  $r$ , and  $\ell$ , if  $r \geq 4\alpha \log(\ell)$  and  $\alpha \geq 1$  then for every  $t$  we have  $\Pr[C_t = 0] \geq 1/2$ .*

Lemma 2.3 implies that for any version  $A_i$  and any time  $t$ ,  $\Pr[D_i \text{ is fixed after time } t] \geq 1/2$ . To see this, note that each  $D_i$  exactly mimics the dynamic counter of Definition 2.2:  $X_j$  corresponds to the time it takes for  $A_i$  to process update  $\sigma_j$ ; by the assumed properties of  $A$ , we have  $E[X_j] = \alpha$ . The counter  $C_j$  then corresponds to the amount of work that  $A_i$  has left to do after the  $j$ -th update phase; in particular,  $C_j = 0$  corresponds to  $D_i$  being fixed after time  $j$ , which by Lemma 2.3 occurs with probability at least  $1/2$ . Since all the  $q$  versions  $A_i$  have independent randomness, the probability that all the  $D_i$  are broken and a FLUSH occurs is at most  $(1/2)^q = 1/n^c$ .  $\square$

*Proof of Lemma 2.3.* Let us focus on some  $C_t$ , and say that  $k$  is the *critical moment* if it is the smallest index such that  $C_j > 0$  for all  $k \leq j \leq t$ . Note that there is exactly one critical moment if  $C_t > 0$

(possibly  $k = t$ ) and none otherwise. Define  $B_i$  ( $B$  for bad) for  $0 \leq i \leq \log(t)$  to be the event that the critical moment occurs in interval  $(t + 1 - 2^{i+1}, t + 1 - 2^i]$ . Thus,

$$\Pr[C_t > 0] = \Pr[B_0 \vee B_1 \vee B_2 \dots \vee B_{\log(t)}] \leq \sum_{0 \leq i \leq \log(t)} \Pr[B_i]. \quad (1)$$

We now need to bound  $\Pr[B_i]$ . Note that if  $B_i$  occurs, then  $C_j > 0$  for  $t + 1 - 2^i \leq j \leq t$ . Thus the counter reduces by  $r$  at least  $2^i$  times between the critical moment and time  $t$  ( $2^i$  and not  $2^i - 1$  because the counter reduces at time  $t$  as well). Furthermore, the counter is always non-negative. Thus,

$$B_i \rightarrow \sum_{t+1-2^{i+1} \leq j \leq t} X_j \geq r2^i,$$

meaning that the event  $B_i$  implies the event  $\sum_{t+1-2^{i+1} \leq j \leq t} X_j \geq r2^i$ . Plugging in for  $r = 4\alpha \log(\ell)$  and recalling that if event  $E_1$  implies  $E_2$  then  $\Pr[E_1] \leq \Pr[E_2]$  we have that

$$\Pr[B_i] \leq \Pr \left[ \sum_{t+1-2^{i+1} \leq j \leq t} X_j \geq 2 \cdot \log(\ell) \cdot \alpha \cdot 2^{i+1} \right]. \quad (2)$$

Now observe that, by linearity of expectation,

$$E \left[ \sum_{t+1-2^{i+1} \leq j \leq t} X_j \right] = \sum_{t+1-2^{i+1} \leq j \leq t} E[X_j] = \alpha \cdot 2^{i+1}. \quad (3)$$

Combining the Markov inequality with Equations 2 and 3 yields  $\Pr[B_i] \leq 1/(2 \log(\ell))$  for any  $i$ . Plugging that into Equation 1, and recalling that  $t \leq \ell$ , we get  $\Pr[C_t > 0] \leq \sum_{0 \leq i \leq \log(t)} 1/(2 \log(\ell)) \leq 1/2$ .  $\square$

Note that the  $\log(\ell)$  factor is necessary, even though intuitively  $r = O(\alpha)$  should be enough, since at each step the counter only goes up by  $\alpha$  (in expectation) and goes down by  $r > \alpha$ , so we would expect it to be zero most of the time. And that is in fact true: with  $r = 4\alpha$  one could show that for any  $\ell$ , the probability that  $C_t = 0$  for at least half the values of  $t \in [0, \ell]$  is at least  $1/2$ . But this claim is not strong enough because it still leaves open the possibility that even if the counter is usually zero, there is some particular time  $t$  at which  $\Pr[C_t = 0]$  is very small.

To exhibit this bad case, consider the following sequence  $X_1, X_2, \dots, X_\ell$ , where each  $X_t$  is chosen independently and is set to  $2r(\ell + 1 - t)$  with probability  $\frac{\alpha}{2r(\ell+1-t)}$  and to 0 otherwise. It is easy to see that for each  $t \leq \ell$  we have  $E[X_t] = \alpha$ . Now, what is  $\Pr[C_\ell = 0]$ ? For each  $t \leq \ell$  if  $X_t \neq 0$ , the counter will reduce by  $r(\ell + 1 - t)$  from time  $t$  to time  $\ell$ , which still leaves us with  $C_\ell \geq 2r(\ell + 1 - t) - r(\ell + 1 - t) > 0$ . Let  $Y_t$  be the indicator variable for the event that  $X_t \neq 0$ . Then,  $\Pr[C_\ell > 0] = \Pr[Y_1 \vee Y_2 \dots \vee Y_\ell]$ . This probability is hard to bound exactly, but note that since the  $Y_t$  are independent random variables between 0 and 1 and we can apply the following Chernoff bound.

**Lemma 2.4** (Chernoff Bound). *Let  $Y_1, Y_2, \dots, Y_k$  be a sequence of independent random variables such that  $0 \leq Y_t \leq U$  for all  $t$ . Let  $Y = \sum_{1 \leq t \leq k} Y_t$  and  $\mu = E[Y]$ . Then the following two properties hold for all  $\delta > 0$ :*

$$\Pr[Y \leq (1 - \delta)\mu] \leq e^{-\frac{\delta^2 \mu}{2U}} \quad (4)$$

$$\Pr[Y \geq (1 + \delta)\mu] \leq e^{-\frac{\delta \mu}{3U}}. \quad (5)$$

Formulation 1 with  $\delta = .74$  yields that if  $\sum_{1 \leq t \leq \ell} E[Y_t] \geq 4$ , then

$$\Pr[C_\ell = 0] = \Pr\left[\sum_{1 \leq t \leq \ell} Y_t < 1\right] < .34 < 1/2.$$

Thus, to have  $\Pr[C_\ell = 0] \geq 1/2$  we certainly need  $\sum_{1 \leq t \leq \ell} E[Y_t] < 4$ . Now observe that

$$\sum_{1 \leq t \leq \ell} E[Y_t] = \frac{\alpha}{2r} \sum_{1 \leq t \leq \ell} \frac{1}{\ell + 1 - t} = \frac{\alpha \cdot \Omega(\log \ell)}{r}$$

Thus, to have  $\sum_{1 \leq t \leq \ell} E[Y_t] \leq 4$ , we indeed need  $r = \Omega(\alpha \log(\ell))$ .

Finally, we observe the restriction to an update sequence of finite length is mainly a technical constraint, which can easily be eliminated. This completes the proof of Theorem 1.1.

*Proof of Theorem 1.1.* Note that if the data structure does not allow any updates then Theorem 2.1 gives the desired bound. Otherwise the data structure allows either insertions or deletions or both. In this case we use a standard technique to enhance the algorithm  $A'$  from Theorem 2.1 providing worst-case high probability update time for a *finite* number of updates to an algorithm  $A''$  providing worst-case high probability update time for an *infinite* number of updates. Recall that we assume that the maximum number of items that are stored in the data structure at any point in time as well as the preprocessing time to build the data structure for any set  $S$  of size polynomial in  $n$  is polynomial in  $n$ . Let this polynomial be upper bounded by  $n^c$  for some constant  $c$ . We break the infinite sequence of updates into non-overlapping *phases*, such that phase  $i$  consists of all updates between update  $i \times n^c$  to update  $(i + 1) \times n^c - 1$ .

During each phase the algorithm uses two instances of algorithm  $A'$ , one of them being called *active* and one being called *inactive*. For each instance the algorithm has a pointer that points to the corresponding data structure. Our new algorithm  $A''$  always points to the data structures  $D_1, D_2, \dots, D_{\log(1/p)}$  of the active instance. In particular it also points to the  $D_i$  for which the active instance ensures correctness. At the end of a phase the inactive data structure of the current phase becomes the active data structure for the next phase and the active one becomes the inactive one.

Additionally,  $A'$  keeps a list  $L$  of all items (e.g. edges in the graph) that are currently stored in the data structure, stored in a balanced binary search tree, such that adding and removing an item takes time  $O(\log n)$  and the set of items that are currently in the data structure can be listed in time linear in their number.

We now describe how each of the two instances is modified during a phase. In the following when we use the term *update* we mean an update in the (main) data structure.

(1) *Active instance.* All updates are executed in the active instance and these are the only modifications performed on the active data structure.

(2) *Inactive instance.* During the first  $n^c/2$  updates in a phase, we do not allow any changes to  $L$ , but record all these updates. Additionally during the first  $n^c/4$  updates in the phase, we enumerate all items in  $L$  and store them in an array by performing a constant amount of work of the enumeration and copy algorithm for each update. Let  $S$  denote this set of items. During the next  $n^c/4$  updates we run the preprocessing algorithm for  $S$  to build the corresponding data structure, again by performing a constant amount of work per update. This data structure becomes our current version of the inactive instance.

We also record all updates of the second half of the phase. During the third  $n^c/4$  updates in the phase, we forward to the inactive instance and to  $L$  all  $n^c/2$  updates of the *first* half of the current



phase, by performing two recorded updates to the inactive instance and to  $L$  per update in the second half of the phase. Finally, during the final  $n^c/4$  updates, we forward to the inactive instance and to  $L$  all  $n^c/2$  updates of the *second* half of the current phase, again performing two recorded update per update. This process guarantees that at the end of a phase the items stored in the active and the inactive instance are identical.

The correctness of this approach is straightforward. To analyze the running time, observe that each update to the data structure will result in one update being processed by the active instance and at most two updates being processed in the inactive instance. Additionally maintaining  $L$  increases the time per update by an additive amount of  $O(\log n)$ . By the union bound, our new algorithm  $A''$  spends worst-case time  $2 \cdot O(\alpha \log(n) \log(1/p))$  with probability  $1 - 2/p$ . By linearity of expectation,  $A''$  has amortized expected update time  $2 \cdot O(\alpha \log(1/p))$ . By initializing the instance in preparation with the modified probability parameter  $p' = p/2$  we obtain the desired formal guarantees.  $\square$

### 3 Dynamic Spanner with Worst-Case Expected Update Time

In this section, we give a dynamic spanner algorithm with with worst-case expected update time that, by our main reduction, can be converted to a dynamic spanner algorithm with high-probability worst-case update time with polylogarithmic overheads. We heavily build upon prior work of Baswana et al. [BKS12] and replace a crucial subroutine requiring deterministic amortization by a randomized counterpart with worst-case expected update time guarantee. In Subsection 3.1, we first give a high-level overview explaining where the approach of Baswana et al. [BKS12] requires (deterministic) amortization and how we circumvent it. We then, in Subsection 3.2, give a more formal review of the algorithm of Baswana et al. together with its guarantees and isolate the dynamic subproblem we improve upon. Finally, in Subsection 3.3, we give our new algorithm for this subproblem and work out its guarantees.

#### 3.1 High-Level Overview

Recall that in the dynamic spanner problem, the goal is to maintain, for a graph  $G = (V, E)$  with  $n = |V|$  vertices that undergoes edge insertions and deletions, and a given integer  $k \geq 2$ , a subgraph  $H = (V, F)$  of size  $|F| = \tilde{O}(n^{1+1/k})$  such that for every edge  $(u, v) \in E$  there is a path from  $u$  to  $v$  in  $H$  of length at most  $2k - 1$ . If the latter condition holds, we also say that the spanner has stretch  $2k - 1$ .

The algorithm of Baswana et al. emulates a “ball-growing” approach for maintaining hierarchical clusterings. In each “level” of the construction, we are given some clustering of the vertices and each cluster is sampled with probability  $p = 1/n^{1/k}$ . The sampled clusters are grown as follows: Each vertex in a non-sampled cluster that is incident on at least one sampled cluster, joins one of these neighboring sampled cluster. Thus, for each unclustered vertex, there might be a choice as to which of its neighboring sampled clusters to join. Furthermore, the algorithm distinguishes the edge that a non-sampled vertex uses to “hook” onto the sampled cluster it joins. All sampled clusters together with the edges between them move to the next level of the hierarchy and in this way the growing of clusters is repeated  $k - 1$  times. With the help of sophisticated data structures this hierarchy is more or less maintained in a straightforward way with some crucial applications of randomization to keep the expected update time low. In such a hierarchical approach, this in particular needs to take into account the potentially exponentially growing blow-up in the propagation of updates: updates

to the input graph might lead to changes in the clustering of the first level of the hierarchy, which might have to be propagated as *induced updates* to the second level of the hierarchy, and so on. Baswana et al. show that the amortized expected number of induced updates at level  $i$  per update to the input graph is at most  $O(1)^i$ . Our contribution in this section is to remove the amortization argument, i.e., to give a bound of  $O(1)^i$  with worst-case expected guarantee

In the first level of the hierarchy, each vertex is a singleton cluster and each non-sampled vertex picks, among all edges going to neighboring sampled vertices, one edge uniformly at random as its hook. Now consider the deletion of some edge  $e = (u, v)$ . If  $e$  was not the hook of  $u$ , then the clustering does not need to be fixed. However, if  $e$  was the hook, then the algorithm spends time up to  $O(\deg(u))$  for picking a new hook, possibly joining a different cluster, and if so informing all neighbors about the cluster change. If the adversary deleting  $e$  is oblivious to the random choices of the algorithm, then one can argue that the probability of  $e$  being the hook of  $u$  is  $\frac{1}{\deg(u)}$ . Thus, the expected update time is  $\frac{1}{\deg(u)} \cdot \deg(u) = O(1)$ .

The situation is more complex at higher levels, when the clusters are not singleton anymore. While the time spent upon deleting the hook is still  $O(\deg_i(u))$ , where  $\deg_i(u)$  is the degree of  $u$  at level  $i$ , one cannot argue that the probability of the deleted edge being the hook is  $O(\frac{1}{\deg_i(u)})$ . To see why this could be the case, Baswana et al. provide the following example of a “skewed” distribution of edges to neighboring clusters: Suppose  $u$  has  $\ell = \Theta(\frac{1}{p} \log(n))$  neighboring clusters such that there are  $\Theta(n)$  edges from  $u$  into the first neighboring cluster and each remaining neighboring cluster has only one edge incident on  $u$ . Now there is a quite high probability (namely  $1 - p \approx 1$ ) that the first cluster is not sampled and with high probability  $O(\log(n))$  of the remaining clusters will be sampled, as follows from the Chernoff bound. Thus, if  $u$  picked the hook uniformly at random from all edges into neighboring sampled clusters, it would join one of the single-edge clusters with high-probability. As there are  $\ell$  edges incident on  $u$  from these single-edge clusters, this gives a probability of approximately  $1/\ell$  for some deleted edge  $(u, v)$  being the hook, which is much larger than  $\frac{1}{\deg_i(u)}$ . This problem would not appear if among all edges going to neighboring clusters a  $p$ -th fraction would be sampled ones. Then, intuitively speaking, one could argue that the probability of some edge  $e = (u, v)$  being the hook edge of  $u$  is at most  $p \cdot \frac{1}{p \deg_i(u)}$ , the probability that the cluster of  $u$  is a sampled one times the probability that a particular edge among all edges to sampled clusters was selected.

This is why Baswana et al. introduce an edge *filtering* step to their algorithm. By making a sophisticated selection of edges going to the next level of the hierarchy, they can ensure that (a) among all such selected edges going to neighboring clusters a  $p$ -th fraction go to sampled clusters and (b) to compensate for edges not being selected for going to the next level, each vertex only needs to add  $O(\frac{1}{p} \log^2(n)) = O(n^{1/k} \log^2(n))$  edges to neighboring clusters to the spanner. The filtering boils down to the following idea: For each vertex  $u$ , group the neighboring non-sampled clusters into  $O(\log(n))$  buckets such that clusters in the same bucket have approximately the same number of edges incident on  $u$ . For buckets that are large enough (containing  $\Theta(\frac{1}{p} \log(n))$  edges), a standard Chernoff bound for binary random variables guarantees that a  $p$ -th fraction of *all* clusters in the respective range for the number of edges incident on  $u$  go to sampled clusters. As all these clusters have roughly the same number of edges incident on  $u$ , a Chernoff bound for positive random variables with bounded aspect ratio also guarantees that a  $p$ -th fraction of the edges of these clusters will go to sampled clusters. Therefore, one gets the desired guarantee if all edges incident on clusters of small buckets are prevented from going to the next level in the hierarchy. To compensate for this filtering, it is sufficient to add one edge – picked arbitrarily – from  $u$  to each

cluster in a small bucket to the spanner. As there are at most  $O(\log(n))$  small buckets containing  $O(\frac{1}{p} \log(n))$  clusters each, this step is affordable without blowing up the asymptotic size of the spanner too much.

Maintaining the bucketing is not trivial because whenever a cluster moves from one bucket to the other it might find itself in a small bucket coming from a large bucket, or vice versa. In order to enforce the filtering constraint, this might cause updates to the next level of the hierarchy. One way of controlling the number of induced updates is amortization: Baswana et al. use soft thresholds for the upper and lower bounds on the number of edges incident on  $u$  for each bucket. This ensures that updates introduced to the next level can be charged to updates in the current level, and leads to an amortized bound of  $O(1)$  on the number of induced updates. Note that the filtering step is the only part in the spanner algorithm of Baswana et al. where this deterministic amortization technique is used. If it were not for this specific sub-problem, the dynamic spanner algorithm would have worst-case expected update time.

Our contribution is a new dynamic filtering algorithm with worst-case expected update time, which then gives a dynamic spanner algorithm with worst-case expected update time. Roughly speaking, we achieve this as follows: whenever the number of edges incident on  $u$  for a cluster  $c$  in some bucket  $j$  (with  $0 \leq j \leq O(\log(n))$ ) exceeds a bucket-specific threshold of  $\alpha_j$ , we move  $c$  up to the appropriate bucket with probability  $\Theta(\frac{1}{\alpha_j})$  after each insertion of an edge between  $u$  and  $c$ . This ensures that, with high probability, the number of edges to  $u$  for clusters in bucket  $j$  is at most  $O(\alpha_j \log(n))$ . Such a bound immediately implies that the expected number of induced updates to the next level per update to the current level is  $O(\frac{1}{\alpha_j} \cdot \alpha_j \log(n)) = O(\log(n))$ , which is already non-trivial but also unsatisfactory because it would lead to an overall update time of  $O(\log(n))^{k/2}$  for a  $(2k - 1)$ -spanner, instead of  $O(1)$  as in the case of Baswana et al. By a more careful analysis we do actually obtain the  $O(1)$ -bound. By taking into account the diminishing probability of not having moved up previously, we argue that the probability to exceed the threshold by a factor of  $2^t$  is proportional to  $1/e^{(2^t)}$ . This bounds the expected number of induced updates by  $\sum_{t \geq 1} 2^t / e^{(2^t)}$ , which converges to a constant. A similar, but slightly more sophisticated approach, is applied for clusters moving down to a lower-order bucket. Here we essentially need to adapt the sampling probability to the amount of deviation from the threshold because in the analysis we have fewer previous updates available for which the cluster has not moved, compared to the case of moving up.

## 3.2 The Algorithm of Baswana et al. at a Glance

In the following, we review the algorithm of Baswana et al. [BKS12] for completeness and isolate the filtering procedure we want to modify. We deviate from the original notation only when it is helpful for our purposes.

### 3.2.1 Spanner Construction

Given an integer parameter  $k \geq 2$ , their algorithm maintains clusterings  $C_0, C_1, \dots, C_{k-1}$  of subgraphs  $G_0 = (V_0, E_0), G_1 = (V_1, E_1), \dots, G_{k-1} = (V_{k-1}, E_{k-1})$ , both to be specified below, where  $G_0 = G$  and, for each  $0 \leq i \leq k - 2$ ,  $G_{i+1}$  is a subgraph of  $G_i$  (i.e.,  $V_{i+1} \subseteq V_i$  and  $E_{i+1} \subseteq E_i$ ). For each  $0 \leq i \leq k - 1$ , a *cluster* of  $G_i$  is a connected subset of vertices of  $G_i$  and the *clustering*  $C_i$  is a partition of  $G_i$  into disjoint clusters. To control the size of the resulting spanner, the clusterings are guided by a hierarchy of randomly sampled subsets of vertices  $S_0 \supseteq S_1 \supseteq \dots \supseteq S_k$  in the sense that each cluster  $c$  in  $C_i$  contains a designated vertex of  $S_i$  called the *center* of  $c$ . The sampling is performed a priori

at the initialization of the algorithm by setting  $S_0 = V$ ,  $S_k = \emptyset$ , and, for  $1 \leq i \leq k-1$ , forming  $S_i$  by selecting each vertex from  $S_{i-1}$  independently with probability  $n^{-1/k}$ . The sets  $S_0, \dots, S_k$  remain the same during the course of the algorithm. In addition to the clusterings, the algorithm will maintain for each cluster of  $C_i$  a forest  $(V, F_i)$  consisting of a spanning tree for each cluster of  $C_i$  rooted at its center such that each vertex in the cluster has a path to the root of length at most  $i$ . Additionally, the algorithm maintains edge sets  $X_i$  and  $Y_i$  for every  $0 \leq i \leq k-1$  to be specified below. The spanner  $H$  will consist of the edge set  $\bigcup_{0 \leq i \leq k-1} (F_i \cup X_i \cup Y_i)$ .<sup>2</sup> Informally, level  $i$  of this hierarchy denotes all algorithms and data structures used to maintain the sets indexed with  $i$  defined above. Initially,  $G_0 = G$  and the clustering  $C_0$  consists of singleton clusters  $\{v\}$  for all vertices  $v \in S_0 = V$ .

We now review how to obtain, for every  $0 \leq i \leq k-1$ , the graph  $G_{i+1} = (V_{i+1}, E_{i+1})$ , the clustering  $C_{i+1}$  of  $G_{i+1}$ , and the edges sets  $F_{i+1}$ ,  $X_{i+1}$ , and  $Y_{i+1}$ , based on the graph  $G_i = (V_i, E_i)$ , the clustering  $C_i$ , the edge set  $F_i$ , and the sets  $S_i$  and  $S_{i+1}$ . Let  $R_i$  be the set of all “sampled” clusters in the clustering  $C_i$  i.e., all clusters in  $C_i$  whose cluster center is contained in  $S_{i+1}$ . Furthermore, let  $V_{i+1}$  be the set consisting of all vertices of  $V_i$  that belong to or are adjacent to clusters in  $R_i$  and let  $\mathcal{N}_i$  be the set consisting all vertices of  $V_i$  that are adjacent to, but do not belong to, clusters in  $R_i$ . Finally, for every  $u \in V_i$ , let  $E_i(u)$  denote the set of edges of  $E_i$  incident on  $u$  and some other vertex in  $V_i$ , and, for every  $u \in V_i$  and every  $c \in C_i$ , let  $E_i(u, c)$  denote the set of edges of  $E_i$  incident on  $u$  and any vertex of  $c$ .

For each vertex  $u \in \mathcal{N}_i$ , the algorithm maintains some edge  $(u, v) \in E_i(u)$  as the *hook* of  $u$  at level  $i$ , called  $\text{hook}(u, i)$ , guaranteeing the following “hook invariant”:

(HI) For every edge  $(u, v) \in E_i(u)$  such that  $v$  is contained in a cluster of  $R_i$ ,  $\Pr[(u, v) = \text{hook}(u, i)] = \frac{1}{|R_i(u)|}$ , where  $R_i(u)$  is the set of edges of  $E_i$  incident on  $u$  and any vertex in a cluster of  $R_i$ .

Now, the clustering  $C_{i+1}$  is obtained by adding each vertex  $u \in \mathcal{N}_i$  to the cluster of the other endpoint of its hook and the forest  $F_{i+1}$  is obtained from  $F_i$  by extending the spanning trees of the clusters by the respective hooks. To compensate for vertices that cannot hook onto any cluster in  $R_i$ , let  $X_i$  be a set of edges containing for each vertex  $v \in V_i \setminus V_{i+1}$  exactly *one* edge of  $E_i(u, c)$  – picked arbitrarily – for each non-sampled neighboring cluster  $c \in C_i \setminus R_i$ .

Additionally, for each  $u \in \mathcal{N}_i$ , the algorithm maintains, for certain parameters  $\lambda \geq g > 1$ ,  $0 < \epsilon < 1$  and  $a > 1$ , a partition of the non-sampled neighboring clusters of  $u$  into  $\lceil \log_g(n) \rceil$  subsets called “buckets”, a set of edges  $\mathcal{F}_i(u) \subseteq \bigcup_{c \in C_i \setminus R_i} E_i(u, c)$  and a set of clusters  $\mathcal{I}_i(u) \subseteq C_i \setminus R_i$  such that:<sup>3</sup>

(F1) For every  $0 \leq j \leq \lceil \log_g(n) \rceil$  and every cluster  $c$  in bucket  $j$ ,  $\frac{g^j}{\lambda} \leq |E_i(u, c)| \leq \lambda g^j$ .

(F2) For every edge  $(u, v) \in \mathcal{F}_i(u)$ , the bucket containing the cluster of  $v$  contains at least  $\ell := 4\gamma a \lambda^2 \frac{1}{\epsilon^3} n^{1/k} \ln(n) \ln(\lambda)$  clusters (where  $\gamma \leq 80$  is a given constant).

(F3) For every edge  $(u, v) \in \bigcup_{c \in C_i \setminus R_i} E_i(u, c) \setminus \mathcal{F}_i(u)$ , the (unique) cluster of  $v$  in  $C_i$  is contained in  $\mathcal{I}_i(u)$ .

Intuitively, the set  $\mathcal{F}_i(u)$  is a *filter* on the edges from  $u$  to non-sampled neighboring clusters and only edges in  $\mathcal{F}_i(u)$  will be passed on to the next level in the hierarchy. The clusters in  $\mathcal{I}_i(u)$  are those for which not all edges incident on  $u$  are contained in  $\mathcal{F}_i(u)$  and thus the algorithm has

<sup>2</sup>In [BKS12], the set  $\bigcup_{0 \leq i \leq k-1} X_i$  was called  $E_S$  and the sets  $Y_i$  did not have an explicit name.

<sup>3</sup>Here we slightly deviate from the original presentation of Baswana et al. by making the filtering process more explicit and also by giving the set  $\mathcal{I}_i(u)$  a name.

to compensate for these missing edges to keep the spanner intact. In the following, we call an algorithm maintaining  $\mathcal{F}_i(u)$  and  $\mathcal{I}_i(u)$  satisfying (F1), (F2), and (F3) for a given vertex  $u$  a *dynamic filtering algorithm* with parameters  $\epsilon$  and  $a$ . To compensate for the filtering of edges, let  $Y_i$  be a set of edges containing, for each vertex  $u \in V_{i+1}$  and each cluster  $c \in \mathcal{I}_i(u)$ , exactly *one* edge from  $E_i(u, c)$  – picked arbitrarily.

For every vertex  $u$ , let  $\mathcal{E}_i(u) = \mathcal{F}_i(u) \cup \bigcup_{c \in R_i} E_i(u, c)$  (where the latter is the set of edges incident on  $u$  from sampled clusters). Now, the edge set  $E_{i+1}$  is defined as follows. Every edge  $(u, v) \in E_i$  with  $u, v \in V_{i+1}$  belongs to  $E_{i+1}$  if and only if  $u$  and  $v$  belong to different clusters in  $C_{i+1}$  and one of the following conditions holds:

- At least one of  $u$  and  $v$  belongs to a sampled cluster (in  $R_i$ ) at level  $i$ , or
- $(u, v)$  belongs to  $\mathcal{E}_i(u)$  as well as  $\mathcal{E}_i(v)$ .

### 3.2.2 Analysis

As mentioned above, the spanner of Baswana et al. is the graph  $H = (V, \bigcup_{0 \leq i \leq k-1} (F_i \cup X_i \cup Y_i))$ . It follows from standard arguments that  $|F_i \cup X_i| \leq O(n^{1+1/k})$  for each  $0 \leq i \leq k-1$ . The stretch bound of  $2k-1$  follows from the clusters having radius at most  $k-1$  together with an argument that for each edge  $e = (u, v)$  not moving to the next level  $u$  has an edge to the cluster of  $v$  (or vice versa) in one of the  $X_i$ 's or one of the  $Y_i$ 's. Finally, the fast amortized update time of the algorithm is obtained by the random choice of the hooks. Roughly speaking, the algorithm only has to perform significant work when the oblivious adversary hits a hook upon deleting some edge  $(u, v)$  from  $E_i$ ; this happens with probability  $\Omega(\frac{1}{|\mathcal{E}_i(u)|})$  and incurs a cost of  $O(|\mathcal{E}_i(u)|)$ , yielding constant expected cost per update to  $E_i$ . More formally, the filtering performed by the algorithm together with invariant (HI) guarantees the following property.

**Lemma 3.1** ([BKS12]). *For every  $0 \leq i \leq k-1$  and every edge  $(u, v) \in E$ ,  $\Pr[(u, v) = \text{hook}(u, i)] \leq \frac{1+2\epsilon}{|\mathcal{E}_i(u)|}$  for any constant  $0 < \epsilon \leq \frac{1}{4}$ .*

The main probabilistic tool for obtaining this guarantee is a Chernoff bound for positive random variables. Compared to the well-known Chernoff bound for binary random variables, the more general tail bound needs a longer sequence of random variables to guarantee a small deviation from the expectation with high probability: the overhead is a factor of  $b \log(b)$ , where  $b$  is the ratio between the largest and the smallest value of the random variables.

**Theorem 3.2** ([BKS12]). *Let  $o_1, \dots, o_\ell$  be  $\ell$  positive numbers such that the ratio of the largest to the smallest number is at most  $b$ , and let  $X_1, \dots, X_\ell$  be  $\ell$  independent random variables such that  $X_i$  takes value  $o_i$  with probability  $p$  and 0 otherwise. Let  $\mathcal{X} = \sum_{1 \leq i \leq \ell} X_i$  and  $\mu = \mathbf{E}[\mathcal{X}] = \sum_{1 \leq i \leq \ell} o_i p$ . There exists a constant  $\gamma \leq 80$  such that if  $\ell \geq \gamma a b \frac{1}{\epsilon^3 p} \ln(n) \log(b)$  for any  $0 < \epsilon \leq \frac{1}{4}$ ,  $a > 1$ , and a positive integer  $n$ , then the following inequality holds:*

$$\Pr[\mathcal{X} < (1 - \epsilon)\mu] < \frac{1}{n^a}$$

The running-time argument sketched above only bounds the running time of each level “in isolation”. For every  $0 \leq i \leq k-1$ , one update to  $G_i$  could lead to more than one *induced* update to  $G_{i+1}$ . Thus, the hierarchical nature of the algorithm leads to an exponential blow-up in the number of induced updates and thus in the running time. Baswana et al. further argue that the

hierarchy only has to be maintained up to level  $\lfloor \frac{k}{2} \rfloor$  by using a slightly more sophisticated rule for edges to enter the spanner from the top level. Together with a careful choice of data structures that allows constant expected time per atomic change, this analysis gives the following guarantee.

**Theorem 3.3** (Implicit in [BKS12]). *Assume there is a fully dynamic edge filtering algorithm  $\mathfrak{F}$  that, in expectation, generates at most  $U(n)$  changes to  $\mathcal{F}_i(u)$  per update to  $E_i(u)$  and, in expectation, has an update time of  $U(n) \cdot T(n)$ . Then, for every  $k \geq 2$ , there is a fully dynamic algorithm  $\mathfrak{S}$  for maintaining a  $(2k - 1)$ -spanner of expected size  $O(kn^{1+1/k} + kn \max_{i,u} |\mathcal{I}_i(u)|)$  with expected update time  $O((3 + 4\epsilon + U(n))^{k/2} \cdot T(n))$ . If the bounds on  $\mathfrak{F}$  are amortized (worst-case), then so is the update time of  $\mathfrak{S}$ .*

### 3.2.3 Summary of Dynamic Filtering Problem

As we focus on the dynamic filtering in the rest of this section, we summarize the most important aspects of this problem in the following. In a dynamic filtering algorithm we focus on a specific vertex  $u \in V_i$  at a specific level  $i$  of the hierarchy<sup>4</sup>, i.e., there will be a separate instance of the filtering algorithm for each vertex in  $V_i$ . The algorithm takes parameters  $\lambda \geq g > 1$ ,  $0 < \epsilon < 1$  and  $a > 1$ , and operates on the subset of edges of  $E_i$  incident on  $u$  and any vertex  $v$  in a non-sampled cluster  $c \in C \setminus R_i$ . These edges are given to the filtering algorithm as a partition  $\bigcup_{c \in C_i \setminus R_i} E_i(u, c)$ , where  $C_i \setminus R_i$ , the set of non-sampled clusters at level  $i$ , will never change over the course of the algorithm.<sup>5</sup> The dynamic updates to be processed by the algorithm are of two types: insertion of some edge  $(u, v)$  to some  $E_i(u, c)$ , and deletion of some edge  $(u, v)$  from some  $E_i(u, c)$ . The goal of the algorithm is to maintain a partition of the clusters into  $\lceil \log_g(n) \rceil$  buckets  $0, \dots, \lfloor \log_g(n) \rfloor$ , a set of clusters  $\mathcal{I}_i(u)$  and a set of edges  $\mathcal{F}_i(u)$  such that conditions (F1), (F2) and (F3) are satisfied.

Condition (F1) states that clusters in the same bucket need to have approximately the same number of edges incident on  $u$ . The “normal” size of  $|E_i(u, c)|$  for a cluster in bucket  $j$  would be  $g^j$  and the algorithm makes sure that  $\frac{g^j}{\lambda} \leq |E_i(u, c)| \leq \lambda g^j$ . Thus, ratio between the largest and the smallest value of  $|E_i(u, c)|$  among clusters  $c$  in the same bucket is at most  $\lambda^2$ . This value corresponds to the parameter  $b$  in Theorem 3.2. The edges in  $\mathcal{F}_i(u)$  serve as a filter for the dynamic spanner algorithm in the sense that only edges in this set are passed on to level  $i + 1$  in the hierarchy. Condition (F2) states that an edge  $(u, v)$  may only be contained in  $\mathcal{F}_i(u)$  if the bucket containing the cluster of  $v$  contains at least  $\ell := 4\gamma a \lambda^2 \frac{1}{\epsilon^3} n^{1/k} \ln(n) \ln(\lambda)$  clusters. Here the choice of  $\ell$  comes from Theorem 3.2;  $a$  is a constant that controls the error probability,  $\epsilon$  controls the amount of deviation from the mean in the Chernoff bound, and  $\gamma$  is a constant from the theorem. Condition (F3) states that clusters  $c$  for which some edge incident on  $u$  and  $c$  is not contained in  $\mathcal{F}_i(u)$  need to be contained in  $\mathcal{I}_i(u)$  (called *inactive* clusters in [BKS12]). Intuitively this is the case because for such clusters the spanner algorithm cannot rely on all relevant edges being present at the next level and thus has to deal with these clusters in a special way.

The goal is to design a filtering algorithm with a small value of  $\lambda$  that has small update time. An additional goal in the algorithm is to keep the number of changes performed to  $\mathcal{F}_i(u)$  small. A change to  $\mathcal{F}_i(u)$  after processing an update to  $E_i(u, c)$  is also called an *induced update* as, in the overall dynamic spanner algorithm, such changes might appear as updates to level  $i + 1$  in the

<sup>4</sup>As explained above, the spanner algorithm only applies the filtering to vertices  $u \in \mathcal{N}_i$ , but we actually run a dynamic filtering algorithm for each vertex in  $V_i$ . The arguments of Baswana et al. for Theorem 3.3 take into account the induced updates occurring whenever some vertex  $u$  joins or leaves the set  $\mathcal{N}_i$ .

<sup>5</sup>Note if vertices join or leave clusters the dynamic filtering algorithm only sees updates for the corresponding edges.

hierarchy, i.e., the insertion (deletion) of an edge  $(u, v)$  to (from)  $\mathcal{F}_i(u)$  might show up as an insertion (deletion) at level  $i + 1$ . As this update propagation takes place in all levels of the hierarchy, we would like have a dynamic filtering algorithm that only performs  $O(1)$  changes to  $\mathcal{F}_i(u)$  per update to its input.

### 3.2.4 Filtering Algorithm with Amortized Update Time

The bound of Baswana et al. follows by providing a dynamic filtering algorithm with the following guarantees.

**Lemma 3.4** (Implicit in [BKS12]). *For any  $a > 1$  and any  $0 < \epsilon \leq \frac{1}{4}$ , there is a dynamic filtering algorithm with amortized update time  $O(1/\epsilon)$  for which the amortized number of changes performed to  $\mathcal{E}_i(u)$  per update to  $E_i(u)$  is at most  $4 + 10\epsilon$  such that  $\mathcal{I}_i(u) \leq O(\frac{a}{\epsilon^7} n^{1/k} \log^2(n))$ , i.e.,  $U(n) = 4 + 10\epsilon = O(1)$  and  $T(n) = O(1/\epsilon)$ .*

Note that the dynamic filtering algorithm is the only part of the algorithm by Baswana et al. that requires amortization. Thus, if one could remove the amortization argument from the dynamic filtering algorithm, one would obtain a dynamic spanner algorithm with worst-case expected guarantee on the update time, which in turn could be strengthened to a worst-case high-probability guarantee. This is exactly how we proceed in the following.

To facilitate the comparison with our new filtering algorithm, we shortly review the amortized algorithm of Baswana et al. Their algorithm uses  $g = \lambda = \frac{1}{\epsilon}$  where  $\epsilon$  is a constant that is optimized to give the fastest update time for the overall spanner algorithm. This leads to  $O(\log_g(n))$  overlapping buckets such that all clusters in bucket  $j$  have between  $g^{j-1}$  and  $g^j$  edges incident on  $u$ .

The algorithm does the following: Every time the number of edges incident on  $u$  of some cluster  $c$  in bucket  $j$  grows to  $g^{j+1}$ ,  $c$  is moved to bucket  $j + 1$ , and every time this number falls to  $g^{j-1}$ ,  $c$  is moved to bucket  $j - 1$ . The algorithm further distinguishes *active* and *inactive* buckets such that active buckets contain at least  $\ell$  clusters and all inactive buckets contain at most  $\kappa\ell$  clusters for some constant  $\kappa$ . An active bucket will be inactivated if its size falls to  $\ell$  and an inactive bucket will be activated if its size grows to  $\kappa\ell$ . Additionally, the algorithm makes sure that  $\mathcal{F}_i(u)$  consists of all edges incident on clusters from active buckets and that  $\mathcal{I}_i$  consists of all clusters in inactive buckets.

By employing soft thresholds for maintaining the buckets and their activation status, Baswana et al. make sure that for each update to  $E_i(u)$  the running time and the number of changes made to  $\mathcal{F}_i(u)$  is constant. For example, every time a cluster  $c$  is moved from bucket  $j$  to bucket  $j + 1$  with a different activation status, the algorithm incurs a cost of at most  $O(g^{j+1})$  – i.e., proportional to  $|E_i(u, c)|$  – for adding or removing the edges of  $E_i(u, c)$  to  $\mathcal{F}_i(u)$ . This cost can be amortized over at least  $g^{j+1} - g^j = \Theta(g^{j+1})$  insertions to  $E_i(u, c)$ , which results in an amortized cost of  $O(g) = O(\frac{1}{\epsilon})$ , i.e., constant when  $\frac{1}{\epsilon}$  is constant. Similarly, the work connected to activation and de-activation is  $O(g)$  when amortized over  $\Theta(\ell)$  clusters joining or leaving the bucket, respectively.

## 3.3 Modified Filtering Algorithm

In the following, we provide our new filtering algorithm with worst-case expected update time, i.e., we prove the following theorem.

**Theorem 3.5.** *For every  $0 \leq i \leq k - 1$  and every  $u \in \mathcal{N}_i$ , there is a filtering algorithm that has worst-case expected update time  $O(\log(n))$  and per update performs at most 16 changes to  $\mathcal{F}_i(u)$  in expectation, i.e.,  $U(n) = 16$  and  $T(n) = O(\log(n))$ . The maximum size of  $\mathcal{I}_i(u)$  is  $O(n^{1/k} \log^6(n) \log \log(n))$ .*

Together with Theorem 3.3 the promised result follows.

**Corollary 3.6** (Restatement of Theorem 1.3). *For every  $k \geq 2$ , there is a fully dynamic algorithm for maintaining a  $(2k - 1)$ -spanner of expected size  $O(kn^{1+1/k} \log^6(n) \log \log(n))$  that has expected worst-case expected update time  $O(20^{k/2} \log(n))$ .*

We now apply the reduction of Theorem 1.1 to maintain  $O(\log(n))$  instances of the dynamic spanner algorithm and use the union of the maintained subgraphs as the resulting spanner. The reduction guarantees that, at any time, one of the maintained subgraphs, and thus also their union, will indeed be a spanner and that the update-time bound holds with high probability.

**Corollary 3.7** (Restatement of Theorem 1.4). *For every  $k \geq 2$ , there is a fully dynamic algorithm for maintaining a  $(2k - 1)$ -spanner of expected size  $O(kn^{1+1/k} \log^7(n) \log \log(n))$  that has worst-case update time  $O(20^{k/2} \log^3(n))$  with high probability.*

### 3.3.1 Design Principles

Our new algorithm uses the following two ideas. First, we observe that it is not necessary to keep only the edges incident from clusters of small buckets out of  $\mathcal{F}_i(u)$ . We can also, somewhat more aggressively, keep away the edges incident from the first  $\ell$  clusters of large buckets out of  $\mathcal{F}_i(u)$ . This is a bit similar to the idea in [BK16] of keeping the first edges of each vertex in the spanner. In this way, we avoid that many updates are induced if the size of a bucket changes from small to large or vice versa. Our modified filtering is done in a deterministic way based only on the current partitioning of the clusters into buckets and on an arbitrary, but fixed ordering of vertices, clusters, and edges.

Second, we employ a probabilistic threshold technique where, after exceeding a certain threshold on the size of the set  $E_i(u, c)$ , a cluster  $c$  changes its bucket with probability roughly inverse to this size threshold. Moving a cluster is an expensive operation that generates changes to the set of filtered edges, which the next level in the spanner hierarchy has to process as induced updates. The idea behind the probabilistic threshold approach is that by taking a sampling probability that is roughly inverse to the number of updates induced by the move, there will only be a constant number of changes in expectation. A straightforward analysis of this approach shows that in each bucket the size threshold will not be exceeded by a factor of more than  $O(\log(n))$  with high probability, which immediately bounds the expected number of changes to the set of filtered edges by  $O(\log(n))$ . By a more sophisticated analysis, taking into account the diminishing probability of not having moved up previously, we can show that exceeding the size threshold by a factor of  $2^t$  happens with probability  $O(1/e^t)$ . Thus, the expected number of induced updates is bounded by an exponentially decreasing series converging to a constant. A similar, but slightly more involved algorithm and analysis is employed for clusters changing buckets because of falling below a certain size threshold.

We remark that a deterministic deamortization of the filtering algorithm by Baswana et al. might be possible in principle without resorting to the probabilistic threshold technique, maybe using ideas similar to the deamortization in the dynamic matching algorithm of Bhattacharya et al. [BHN17]. However, such a deamortization needs to solve non-trivial challenges and we believe



that the probabilistic threshold technique leads to a simpler algorithm. Similarly it might be possible to use the probabilistic threshold technique to emulate the less aggressive filtering of Baswana et al. that only filters away edges incident on large buckets. Here, not using the probabilistic threshold technique seems the simpler choice.

### 3.3.2 Setup of the Algorithm

In our algorithm, described below for a fixed vertex  $u$ , we work with an arbitrary, but fixed, order on the vertices of the graph. The order on the vertices induces an order on the edges, by lexicographically comparing the ordered pair of incident vertices of the edges, and an order on the clusters, by comparing the respective cluster centers. For each  $0 \leq j \leq \lfloor \log(n) \rfloor$  and also for  $j = -\infty$ , we maintain a bucket by organizing the clusters in bucket  $j$  in a binary search tree  $B_j$ , employing the aforementioned order on the clusters. Similarly, for  $0 \leq j \leq \lfloor \log(n) \rfloor$  and also for  $j = -\infty$ , we organize the edges incident on  $u$  and each bucket  $j$  in a binary search tree  $T_j$ , i.e., a search tree ordering the set of edges  $\bigcup_{c \in B_j} E_i(u, c)$ , where these edges are compared lexicographically as cluster-edge pairs. The bucket  $-\infty$  has the special role of organizing all clusters  $c$  that currently have no edges incident on  $u$ . In this section we will use the conventions  $\log(0) = -\infty$  and  $\frac{1}{0} = \infty$  to minimize the effort for handling this special case in the description of the algorithm.

We set  $\lambda = 2^{\lceil \log(4 + \ln(n)) \rceil} = O(\log(n))$ ,  $\ell = 4\gamma a \lambda^2 \frac{1}{e^3} n^{1/k} \ln(n) \ln(\lambda) = O(n^{1/k} \log^3(n) \log \log(n))$  and, for every  $0 \leq j \leq \lfloor \log(n) \rfloor$  we set  $\alpha_j = 2^j$  and for  $j = -\infty$  we set  $\alpha_{-\infty} = 0$ . Our algorithm will maintain the following invariants for every  $0 \leq j \leq \lfloor \log(n) \rfloor$ :

- (B1) For each cluster  $c$  in bucket  $j$ ,  $\frac{\alpha_j}{\lambda} \leq |E_i(u, c)| \leq \lambda \alpha_j$ .
- (B2) The edges of the first  $\ell \cdot \lambda \alpha_j$  cluster-edge pairs of  $T_j$  (or all cluster-edge pairs of  $T_j$  if there are less than  $\ell \cdot \lambda \alpha_j$  of them) are not contained in  $\mathcal{F}_i(u)$  and the remaining edges of  $T_j$  are contained in  $\mathcal{F}_i(u)$ .
- (B3) The first  $1 + \lambda^2 \ell$  clusters of  $B_j$  are contained in  $\mathcal{I}_i(u)$  and the remaining clusters of  $B_j$  are not contained in  $\mathcal{I}_i(u)$ .

Additionally it will maintain the following invariant:

- (B4) For each cluster  $c$  in bucket  $-\infty$ ,  $|E_i(u, c)| = 0$ .

Observe that invariant (B1) is equal to condition (F1) and that invariant (B3) immediately implies the claimed bound on  $\mathcal{I}_i(u)$  as there are  $O(\log(n))$  buckets, each contributing  $O(\lambda^2 \ell)$  clusters.

Furthermore, the invariants also imply correctness in terms of conditions (F2) and (F3) because of the following reasoning: For condition (F2), let  $(u, v) \in \mathcal{F}_i(u)$  and let  $c$  denote the cluster of  $v$ . Then, by invariant (B2), there are at least  $\ell \cdot \lambda \alpha_j$  cluster-edge pairs contained in  $T_j$  that are lexicographically smaller than the pair consisting of  $c$  and  $(u, v)$ . As each cluster in bucket  $j$  has at most  $\lambda \alpha_j$  edges incident on  $u$  by invariant (B1), it follows that there are at least  $\ell$  clusters contained in bucket  $j$  as otherwise  $T_j$  could not contain at least  $\ell \cdot \lambda \alpha_j$  cluster-edge pairs.

For condition (F3), let  $(u, v) \in E_i(u) \setminus \mathcal{F}_i(u)$  and let  $c$  denote the cluster of  $v$ . Then the pair consisting of  $c$  and  $(u, v)$  must be among the first  $\ell \cdot \lambda \alpha_j$  entries of  $T_j$  by invariant (B2). As each cluster in bucket  $j$  has at least  $\frac{\alpha_j}{\lambda}$  edges incident on  $u$  by invariant (B1), there are thus at most  $\frac{\lambda \alpha_j}{\alpha_j / \lambda} \ell = \lambda^2 \ell$  clusters in bucket  $j$  that are smaller than  $c$  in terms of the chosen ordering on the clusters. It follows that  $c$  must be among the first  $1 + \lambda^2 \ell$  clusters of  $B_j$  and by invariant (B3) is thus contained in  $\mathcal{I}_i(u)$  as required by condition (F1).

### 3.3.3 Modified Bucketing Algorithm

The algorithm after an update to some edge  $(u, v)$  is as follows:

- If the edge  $(u, v)$  was inserted and if now  $|E_i(u, c)| \geq 2\alpha_j$ , where  $c$  is the cluster of  $v$  and  $j$  is the number  $c$ 's bucket, do the following: Flip a biased coin that is “heads” with probability  $\min(\frac{1}{\alpha_j}, 1)$ . If the coin shows “heads” or if  $|E_i(u, c)| = \lambda \cdot \alpha_j$ , then move cluster  $c$  up to bucket  $j' = \lceil \log(|E_i(u, c)|) \rceil$  by performing the following steps:
  1. Remove  $c$  from  $B_j$  and add it to  $B_{j'}$ .
  2. Remove all edges of  $E_i(u, c)$  from  $T_j$  and add them to  $T_{j'}$ .
- If the edge  $(u, v)$  was deleted and if now  $|E_i(u, c)| \leq \frac{\alpha_j}{2}$ , where  $c$  is the cluster of  $v$  and  $j$  is the number  $c$ 's bucket, do the following: Flip a biased coin that is “heads” with probability  $\min(\frac{2^{2t+1}}{\alpha_j}, 1)$  for the maximum  $t \geq 1$  such that  $|E_i(u, c)| \leq \frac{\alpha_j}{2^t}$ . If the coin shows “heads” or if  $|E_i(u, c)| = \frac{\alpha_j}{\lambda}$ , then move cluster  $c$  down to bucket  $j' = \lfloor \log(|E_i(u, c)|) \rfloor$  by performing the following steps:
  1. Remove  $c$  from  $B_j$  and add it to  $B_{j'}$ .
  2. Remove all edges of  $E_i(u, c)$  from  $T_j$  and add them to  $T_{j'}$ .

Additionally, invariants (B2) and (B3) are maintained in the trivial way by making the necessary changes to  $\mathcal{F}_i(u)$  after a change to  $T_j$  and to  $\mathcal{I}_i$  after a change to  $B_j$ , respectively. Observe that the algorithm handles the corner cases for invariant (B4) correctly. Furthermore, invariant (B1) is satisfied because the following invariant (B1') holds as well for every  $0 \leq j \leq \lfloor \log(n) \rfloor$  by the design of the algorithm:

(B1') Whenever a cluster  $c$  moves to bucket  $j$ ,  $\frac{\alpha_j}{2} < |E_i(u, c)| < 2\alpha_j$ .

### 3.3.4 Analysis of Induced Updates and Running Time

We now analyze the number of changes to  $\mathcal{F}_i(u)$  per update to  $E_i(u)$ . These changes are also called induced updates. When multiplying this number by a factor of  $O(\log(n))$  – the worst-case time per operation for self-balancing binary search trees – this also bounds the update time of our algorithm. Observe that each update to  $E_i(u)$  causes at most one move of a cluster  $c$  from one bucket to another bucket. For each such move we incur at most  $3|E_i(u, c)|$  changes to  $\mathcal{F}_i(u)$ . For technical reasons, we go on by giving slightly different analyses for the cases of moving up and moving down.

**Moving Up.** For every integer  $1 \leq t \leq \log(\lambda) - 1$ , let  $p_t$  be the probability that  $2^t \alpha_j \leq |E_i(u, c)| < 2^{t+1} \alpha_j$  when  $c$  is moved up and let  $q$  be the probability that  $|E_i(u, c)| = \lambda \cdot \alpha_j$  when  $c$  is moved up. Note that this covers all events for  $c$  being moved up. As observed above, each move induces at most  $3|E_i(u, c)|$  updates, where  $|E_i(u, c)| < 2^{t+1}$  with probability  $p_t$  and  $|E_i(u, c)| \leq n$  in any case. Thus, by the law of total expectation, the expected number of induced updates per insertion to  $E_i(u, c)$  is at most

$$\sum_{1 \leq t \leq \log(\lambda) - 1} p_t \cdot 3 \cdot 2^{t+1} \alpha_j + q \cdot 3n.$$

We now bound  $p_t$ , the probability that  $2^t \alpha_j \leq |E_i(u, c)| < 2^{t+1} \alpha_j$  when  $c$  is moved up. As soon as  $|E_i(u, c)|$  exceeds the threshold  $2\alpha_j$ , each insertion makes  $c$  move up with probability  $\frac{1}{\alpha_j}$  (when

the biased coin shows “heads”). For  $t = 1$ , we clearly have  $p_t \leq \frac{1}{\alpha_j}$ . For  $2 \leq t \leq \log(\lambda) - 1$ ,  $p_t$  is determined by one “heads” preceded by at least  $2^t \alpha_j - 2\alpha_j$  “tails” in the coin flips of previous insertions to  $E_i(u, c)$ , i.e.,  $p_t$  is bounded by

$$p_t \leq \frac{1}{\alpha_j} \cdot \left(1 - \frac{1}{\alpha_j}\right)^{(2^t - 2) \cdot \alpha_j} \leq \frac{1}{\alpha_j} \cdot \frac{1}{e^{2^t - 2}}.$$

Here we use the inequality  $(1 - \frac{1}{x})^x \leq \frac{1}{e}$ , where  $e$  is Euler’s constant. Similarly,  $q$ , the probability that  $|E_i(u, c)| = \lambda \alpha_j$  with  $\lambda = 2^{\lceil \log(4 + \ln(n)) \rceil}$  when  $c$  is moved up, is determined by  $\alpha_j(\lceil a \ln(n) \rceil + 2) - 2\alpha_j$  “tails”. Thus,  $q$  is bounded by

$$q \leq \frac{1}{e^{2^{\lceil \log(4 + \ln(n)) \rceil} - 2}} \leq \frac{1}{e^{2 + \ln(n)}} = \frac{1}{e^2 n}.$$

We can now bound the expected number of induced updates by

$$\begin{aligned} \sum_{1 \leq t \leq \log(\lambda) - 1} p_t \cdot 3 \cdot 2^{t+1} \alpha_j + q \cdot 3n &= \frac{1}{\alpha_j} \cdot 3 \cdot 2^2 \alpha_j + \sum_{1 \leq t \leq \log(\lambda) - 1} \frac{1}{\alpha_j} \cdot \frac{1}{e^{2^t - 2}} \cdot 3 \cdot 2^{t+1} \alpha_j + \frac{1}{e^2 n} \cdot 3n \\ &\leq 12 + 6 \cdot \sum_{2 \leq t \leq \infty} \frac{2^t}{e^{2^t - 2}} + 0.41 \\ &\leq 12 + 6 \cdot 0.57 + 0.41 \\ &\leq 16. \end{aligned}$$

**Moving Down.** For every  $1 \leq t \leq \log(\lambda) - 1$ , let  $p_t$  be the probability that  $\frac{\alpha_j}{2^{t+1}} < |E_i(u, c)| \leq \frac{\alpha_j}{2^t}$  when  $c$  is moved down and let  $q$  be the probability that  $|E_i(u, c)| = \frac{\alpha_j}{\lambda}$  when  $c$  is moved down. As observed above, each move induces at most  $3|E_i(u, c)|$  updates and thus, by the law of total expectation, the expected number of induced updates per deletion from  $E_i(u, c)$  is at most

$$\sum_{1 \leq t \leq \log(\lambda) - 1} p_t \cdot 3 \frac{\alpha_j}{2^t} + q \cdot 3n.$$

We now bound  $p_t$ , the probability that  $\frac{\alpha_j}{2^{t+1}} < |E_i(u, c)| \leq \frac{\alpha_j}{2^t}$ . For  $t = 1$ , we clearly have  $p_1 \leq \frac{2^3}{\alpha_j} = \frac{8}{\alpha_j}$  as this is the probability that just a single coin flip made the cluster move down. For  $2 \leq t \leq \log(\lambda) - 1$ , observe that for  $|E_i(u, c)| \leq \frac{\alpha_j}{2^t}$  to hold, there must have been at least  $t - 1$  subsequences of deletions such that after every deletion in subsequence  $s$  we had  $\frac{\alpha_j}{2^{s+1}} < |E_i(u, c)| \leq \frac{\alpha_j}{2^s}$  (where  $1 \leq s \leq t - 1$ ). Observe that the  $s$ -th subsequence consists of  $m_s := \frac{\alpha_j}{2^s} - \frac{\alpha_j}{2^{s+1}} = \frac{\alpha_j}{2^{s+1}}$  many deletions. Remember that during the  $s$ -th subsequence the probability of  $c$  moving down is  $\min(\frac{2^{2s+1}}{\alpha_j}, 1)$ . If  $\alpha_j \leq 2^{2s+1}$  for any subsequence  $s$ , then  $c$  moves down certainly in subsequence  $s$  and thus  $p_t = 0$ . Otherwise,  $p_t$  is determined by one “heads” preceded by at least  $m_s$  “tails” for each

subsequence  $s$ , it is bounded by

$$\begin{aligned}
p_t &\leq \frac{2^{2t+1}}{\alpha_j} \cdot \prod_{1 \leq s \leq t-1} \left(1 - \frac{2^{2s+1}}{\alpha_j}\right)^{m_s} \\
&= \frac{2^{2t+1}}{\alpha_j} \cdot \prod_{1 \leq s \leq t-1} \left(1 - \frac{2^{2s+1}}{\alpha_j}\right)^{\frac{\alpha_j}{2^{s+1}}} \\
&\leq \frac{2^{2t+1}}{\alpha_j} \cdot \prod_{1 \leq s \leq t-1} \frac{1}{e^{2^s}} \\
&= \frac{2^{2t+1}}{\alpha_j} \cdot \frac{1}{e^{\sum_{1 \leq s \leq t-1} 2^s}} \\
&= \frac{2^{2t+1}}{e^{2^t-2} \alpha_j}.
\end{aligned}$$

Similarly,  $q$ , the probability that  $|E_i(u, c)| = \frac{\alpha_j}{\lambda}$  with  $\lambda = 2^{\lceil \log(4+\ln(n)) \rceil}$  when  $c$  is moved down, is bounded by

$$q \leq \frac{1}{2^{\lceil \log(4+\ln(n)) \rceil - 2}} \leq \frac{1}{e^{2+\ln(n)}} = \frac{1}{e^2 n}.$$

We can now bound the expected number of induced updates by

$$\begin{aligned}
\sum_{1 \leq t \leq \log(\lambda)-1} p_t \cdot 3 \cdot \frac{\alpha_j}{2^t} + q \cdot 3n &= p_1 \cdot 3 \cdot \frac{\alpha_j}{2} + \sum_{2 \leq t \leq \log(\lambda)-1} p_t \cdot 3 \cdot \frac{\alpha_j}{2^t} + q \cdot 3n \\
&\leq 12 + \sum_{2 \leq t \leq \log(\lambda)-1} \frac{2^{2t+1}}{e^{2^t-2} \alpha_j} \cdot 3 \cdot \frac{\alpha_j}{2^t} + \frac{1}{e^2 n} \cdot 3n \\
&\leq 12 + 6 \cdot \sum_{1 \leq t \leq \infty} \frac{2^t}{e^{2^t-2}} + 0.41 \\
&\leq 12 + 6 \cdot 0.57 + 0.41 \\
&\leq 16.
\end{aligned}$$

This concludes the proof that the expected number of induced updates is at most 16.

## 4 Dynamic Maximal Matching with Worst-Case Expected Update Time

In this section we turn to proving Theorem 1.5. We achieve our result by modifying the algorithm of Baswana et al. [BGS18], which achieves *amortized* expected time  $O(\log(n))$ . We start by describing the original algorithm of Baswana et al., and then discuss why their algorithm does not provide a worst-case expected guarantee, and the modifications we make to achieve this guarantee. Throughout this section, we define a vertex to be *free* if it is not matched, and we define  $\text{MATE } v$ , for matched  $v$ , to the vertex that  $v$  is matched to.

## 4.1 The Original Matching Algorithm of Baswana et al.

**High-Level Overview.** Let us consider the trivial algorithm for maintaining a maximal matching. Insertion of an edge  $(u, v)$  is easy to handle in  $O(1)$  time: if  $u$  and  $v$  are both free then we add the edge to the matching; otherwise, we do nothing. Now consider deletion of an edge  $(u, v)$ . If  $(u, v)$  was not in the matching then the current matching remains maximal, so there is nothing to be done and the update time is only  $O(1)$ . If  $(u, v)$  was in the matching, then both  $u$  and  $v$  are now free and must scan all of their neighbors looking for a new neighbor to match to. The update time is thus  $\max\{\text{DEGREE}(u), \text{DEGREE}(v)\}$ . This is the only expensive operation.

At a very high level, the idea of the Baswana et al. is to create a hierarchy of the vertices (loosely) according to their degrees. High degree vertices are more expensive to handle. To counterbalance this, the algorithm ensures that when a high degree vertex  $v$  picks a new mate, it chooses that mate *at random* from a large number of neighbors of  $v$ . Thus, although the deletion of the matching edge  $(v, \text{MATE}(v))$  will be expensive, there is a high probability that the adversary will first have to delete many non-matching  $(v, w)$  (which are easy to process) before it finds  $(v, \text{MATE}(v))$ . (Recall that the algorithm of Baswana et al. and our modification both assume an oblivious adversary).

### Setup of the Algorithm.

- Each edge  $(u, v)$  will be *owned* by exactly one of its endpoints. Let  $O_v$  contain all edges owned by  $v$ . Loosely speaking, if  $(u, v) \in O_v$  then  $v$  is responsible for telling  $u$  about any changes in its status (e.g.  $v$  becomes unmatched or changes levels in the hierarchy), but not vice versa.
- The algorithm maintains a partition of the vertices into  $\lfloor \log_4(n) \rfloor + 2$  levels. The levels are numbered from  $-1$  to  $L_0 = \lfloor \log_4(n) \rfloor$ . During the algorithm, when a vertex moves to level  $i$ , it owns at least  $4^i$  edges. Level  $-1$  then contains the vertices that own no edges. The algorithm always maintains the invariant that if  $\text{LEVEL}(u) < \text{LEVEL}(v)$  then edge  $(u, v) \in O_v$ .
- For every vertex  $u$ , the algorithm stores a dynamic hash table of the edges in  $O_u$ . The algorithm also maintains the following list of edges for  $u$ : for each  $i \geq \text{LEVEL}(u)$ , let  $\mathcal{E}_u^i$  be the set of all those edges incident on  $u$  from vertices at level  $i$  that are not owned by  $u$ . The set  $\mathcal{E}_u^i$  will be maintained in a dynamic hash table. However, the onus of maintaining  $\mathcal{E}_u^i$  will not be on  $u$ , because these edges are by definition not owned by  $u$ . For example, if a neighbor  $v$  of  $u$  moves from level  $i > \text{LEVEL}(u)$  to level  $j > i$ , then  $v$  will remove  $(u, v)$  from  $\mathcal{E}_u^i$  and insert it to  $\mathcal{E}_u^j$ .

**Invariants and Subroutines.** Define  $N_{<j}(v)$  to contain all neighbors of  $v$  strictly below level  $j$  and  $N_{=j}(v)$  to contain all neighbors of  $v$  at level exactly  $j$ . The key invariant of the hierarchy is that a vertex moves up to a higher level whenever it can guarantee that by doing so it will have sufficiently many neighbors below it. For  $j > \text{LEVEL}(v)$ , define  $\phi_v(j) = |N_{<j}(v)|$ , and  $\phi_v(j) = 0$  otherwise. We can equivalently define  $\phi_v(j)$  in terms of the  $O_v$  and  $\mathcal{E}_v^i$  structures: For a vertex  $v$  with  $\text{LEVEL}(v) = i$ ,

$$\phi_v(j) = \begin{cases} |O_v| + \sum_{i \leq k < j} |\mathcal{E}_v^k| & \text{if } j > i \\ 0 & \text{otherwise.} \end{cases}$$

We now describe some guarantees of the Baswana et al. algorithm. Note that the hierarchy only imposes an upper bound on  $N_{<j}(v)$  (Invariant 3); a lower bound on  $N_{<j}(v)$  only comes into play when  $v$  picks a new matching edge (Matching Property).

- **Invariant 1:** Each edge is owned by exactly one endpoint, and if the endpoints of the edge are at different levels, the edge is owned by the endpoint at higher level. (If the two endpoints are at the same level, then the tie is broken appropriately by the algorithm.)
- **Invariant 2:** Every vertex at level  $\geq 0$  is matched and every vertex at level  $-1$  is free.
- **Invariant 3:** For each vertex  $v$  and for all  $j > \text{LEVEL}(v)$ ,  $\phi_v(j) < 4^j$  holds true. Note that  $\phi_v(j) = |N_{<j}(v)|$  by definition. Combined with Invariant 1 this implies that  $|\mathcal{O}_v| \leq 4^{\text{LEVEL}(v)+1} = O(4^{\text{LEVEL}(v)})$  and  $N_{=\text{LEVEL}(v)}(v) \leq 4^{\text{LEVEL}(v)+1} = O(4^{\text{LEVEL}(v)})$  if  $\text{LEVEL}(v) < L_0$ . For  $\text{LEVEL}(v) = L_0$ ,  $4^{L_0+1} \geq n$ , so trivially  $|\mathcal{O}_v| = O(4^{\text{LEVEL}(v)})$  and  $N_{=\text{LEVEL}(v)}(v) = O(4^{\text{LEVEL}(v)})$ .
- **Invariant 4:** Both the endpoints of a matched edge are at the same level.
- **Matching Property:** If a vertex  $v$  at level  $j > -1$  is (temporarily) unmatched, the algorithm proceeds as follows: if  $|N_{<j}(v)| \geq 4^j$ ,  $v$  picks a new mate *uniformly at random* from  $N_{<j}(v)$ ; If  $|N_{<j}(v)| < 4^j$ , then  $v$  falls to level  $j - 1$  and is recursively processed there (i.e. depending on the size of  $N_{<j-1}(v)$ ,  $v$  either picks a random mate from  $N_{<j-1}(v)$  or continues to fall.)

*Remark 4.1.* Observe that if we maintain these invariants then we always have a maximal matching: By Invariant 1, each edge  $e$  is owned by exactly one endpoint  $v$ . As by Invariant 3 a vertex at level  $-1$  owns no edges,  $v$  is at level  $\geq 0$ , and by Invariant 2,  $v$  must be matched. Thus, every edge has an endpoint that is matched.

We now consider the procedures used by the algorithm of Baswana et al. to maintain the hierarchy and the maximal matching. The bulk of the work is in maintaining  $\mathcal{O}_v$ ,  $\mathcal{E}_v^j$ , and  $\phi_v(j)$ , which change due to external additions and deletions of edges, and also due to the algorithm internally moving vertices in the hierarchy to satisfy the invariants above. We largely stick to the notation of the original paper, but we omit details that remain entirely unchanged in our approach. See Section 4 in [BGS18] for the original algorithm description (and its analysis).

- **INCREMENT- $\phi(v, i)$**  increases  $\phi_v(i)$  by one, whereas **DECREMENT- $\phi(v, i)$**  decreases it.
- **RISE( $v, i, j$ )** (new notation) moves a vertex from level  $i$  to  $j$ . This results in changes to many of the  $\mathcal{O}$  and  $\mathcal{E}$  lists. In particular,  $v$  takes ownership of all edges  $(v, w)$  with  $w \in N_{<j}(v)$ . Moreover, for any vertex  $w \in N_{<i}(v)$ , edge  $(v, w)$  is removed from  $\mathcal{E}_w^i$ , and for every  $w \in N_{<j}(v)$ , edge  $(v, w)$  is added to  $\mathcal{E}_w^j$ . As a result, the algorithm runs **DECREMENT- $\phi(w, k)$**  for every  $w \in N_{<j}(u)$ , and every  $i < k \leq j$ . A careful analysis bounds the total amount of bookkeeping work at  $O(4^j)$  (see Lemma 4.3).
- **FALL( $v, i$ )** (new notation) moves  $v$  from level  $i$  to level  $i - 1$ . As above this leads to bookkeeping work:  $\mathcal{O}_w$ ,  $\mathcal{E}_w^j$ , and  $\mathcal{E}_w^{j-1}$  change for many neighbors  $w$  of  $v$ . Note that only vertices  $w$  previously owned by  $v$  are affected, so by Invariant 3, the total amount of bookkeeping work is at most  $|\mathcal{O}_v| = O(4^i)$ .

The algorithm must also do **INCREMENT- $\phi(w, i)$**  for every  $w$  that was previously in  $N_{<i}(v)$ . Such an increment might result in  $w$  violating Invariant 3 (if  $\phi_w(i)$  goes from  $4^i - 1$  to  $4^i$ ), in which case the algorithm executes **RISE( $w, \text{LEVEL}(w), i$ )**. Moreover, if  $w'$  was the previous mate of  $w$ , then edge  $(w, w')$  is removed from the matching to preserve invariant 4, so the algorithm must also execute **FIXFREEVERTEX( $w$ )** and **FIXFREEVERTEX( $w'$ )** (see below), which can in turn lead to more calls to **FALL** and **RISE**. One of the main tasks of the analysis will be to bound this cascade.

- $\text{FIXFREEVERTEX}(v)$  handles the case when a vertex  $v$  is unmatched; this can happen because the matching edge was deleted, or because  $v$  newly rose/fell to level  $i$ , where  $i = \text{LEVEL}(v)$ . Following the Matching Property, if  $|N_{<i}(v)| < 4^i$ , then the algorithm executes  $\text{FALL}(v, i)$ , followed by  $\text{FIXFREEVERTEX}(v)$ . On the other hand, if  $|N_{<i}(v)| \geq 4^i$ , then  $v$  remains at level  $i$  and picks a new mate by executing  $\text{GENERICRANDOMSETTLE}(v, i)$ .
- $\text{GENERICRANDOMSETTLE}(v, i)$  Finds a new mate  $w$  for a vertex  $v$  at level  $i$  assuming that  $|N_{<i}(v)| \geq 4^i$ . The algorithm picks  $w$  uniformly at random from  $N_{<i}(v)$ . Let  $\ell = \text{LEVEL}(w) < i$ . The algorithm first does  $\text{RISE}(w, \ell, i)$  (to satisfy Invariant 4), and then matches  $v$  to  $w$ . Note that if  $\ell \neq -1$ , then  $w$  had a previous mate  $w'$  which is now unmatched, so the algorithm now does  $\text{FIXFREEVERTEX}(w')$ .

**Handling Edge Updates.** We now show how the algorithm maintains the invariants under edge updates. First consider the insertion of edge  $(u, v)$ . Say w.l.o.g that  $\text{LEVEL}(v) \geq \text{LEVEL}(u)$ . Then  $(u, v)$  is added to  $O_v$  and to  $\mathcal{E}_u^{\text{LEVEL}(v)}$ . The algorithm must then execute  $\text{INCREMENT-}\phi(u, j)$  and  $\text{INCREMENT-}\phi(v, j)$  for every  $j > \text{LEVEL}(v)$ . This takes time  $O(\log(n))$  and might additionally result in some level  $\ell$  for which  $\phi_v(\ell) \geq 4^\ell$  (or  $\phi_u(\ell) \geq 4^\ell$ ), in which case Invariant 3 is violated so the algorithm performs  $\text{RISE}(v, \text{LEVEL}(v), \ell)$  (or  $\text{RISE}(u, \text{LEVEL}(u), \ell)$ ). (If  $\phi_v(\ell) \geq 4^\ell$  for multiple levels  $\ell$ , then  $v$  rises to the highest such  $\ell$ .)

Now consider the deletion of an edge  $(u, v)$  with  $\text{LEVEL}(v) \geq \text{LEVEL}(u)$ . The algorithm first does  $O(\log(n))$  work of simple bookkeeping: it removes  $(u, v)$  from  $O_v$  and  $\mathcal{E}_u^{\text{LEVEL}(v)}$ , and executes the corresponding calls to  $\text{DECREMENT-}\phi$ . If  $(u, v)$  was *not* a matching edge, the work ends there: unlike with  $\text{INCREMENT-}\phi$ , the procedure  $\text{DECREMENT-}\phi$  cannot lead to the violation of any invariants. By contrast, the most expensive operation is the deletion of a *matched* edge  $(u, v)$ , because the algorithm must execute  $\text{FIXFREEVERTEX}(u)$ , and  $\text{FIXFREEVERTEX}(v)$ .

**Analysis Sketch.** Whereas our final algorithm is very similar to the original algorithm of Baswana et al., our analysis is quite different, so we only provide a brief sketch of their original analysis. The basic idea is that because a vertex  $v$  is only responsible for edges in  $O_v$ , processing a vertex at level  $i$  takes time  $O(4^{i+1})$  (Invariant 3). The crux of the analysis is in arguing that vertices at high level are processed less often. There are two primary ways a vertex  $v$  can be processed at level  $i$ . **1)**  $v$  rises to level  $i$  because  $\phi_v(i)$  goes from  $4^i - 1$  to  $4^i$ . This does not happen often because many  $\text{INCREMENT-}\phi(v, i)$  are required to reach such a high  $\phi_v(i)$ . **2)** the matching edge  $(v, \text{MATE}(v))$  is deleted from the graph. This does not happen often because by Matching Property,  $v$  originally picks its mate at random from at least  $4^i$  options, so since the adversary is oblivious, it will in expectation delete many non-matching edges  $(v, w)$  (which are easy to process) before it hits upon  $(v, \text{MATE}(v))$ .

## 4.2 Our Modification of the Baswana et al. Algorithm

There are two reasons why the original algorithm of Baswana et al. does not guarantee a worst-case expected update time.

**1:** The algorithm uses a hard threshold for  $\phi_v(i)$ : the update which increases  $\phi_v(i)$  from  $4^i - 1$  to  $4^i$  is guaranteed to lead to the expensive execution of  $\text{RISE}(v, \text{LEVEL}(v), i)$ . Thus, while their algorithm guaranteed that overall few updates lead to this expensive event, it is not hard to construct an update sequence which forces one particular update to be an expensive one. To overcome this,

we use a randomized threshold, where every time  $\phi_v(i)$  increases,  $v$  rises to level  $i$  with probability  $\Theta(\log(n)/4^i)$ .

**2:** Consider the deletion of an edge  $(u, v)$  where  $i = \text{LEVEL}(v) \geq \text{LEVEL}(u)$ . Baswana et al. showed that this deletion takes time  $O(\log(n))$  if  $u \neq \text{MATE}(v)$ , and time  $O(4^i)$  if  $u = \text{MATE}(v)$ . At first glance this seems to lead to an expected-worst-case guarantee: we know by the Matching Property that  $v$  picked its mate at random from a set of at least  $4^i$  vertices, so if we could argue that for any edge  $(u, v)$  we always have  $\Pr[\text{MATE}(v) = u] \leq 1/4^i$ , then the expected time to process *any* deletion would be just  $O(\log(n))$ .

Unfortunately, in the original algorithm it is *not* the case that  $\Pr[\text{MATE}(v) = u] \leq 1/4^i$ . To see this, consider the following sequence of updates to a vertex  $v$ , in which  $v$  will be always at level  $i$ , every updated edge  $(u, v)$  will have  $\text{LEVEL}(u) < \text{LEVEL}(v)$ , and  $|N_{<i}(v)|$  will always be between  $4^i$  and  $2 \cdot 4^i$ . The other vertices in the sequence are  $v', x_1, x_2, \dots, x_{4^i-1}$  and  $y_1, y_2, \dots, y_{4^i-1}$ . At the beginning,  $v$  has an edge to  $v'$  and to all the  $x_i$ . The update sequence repeats the following cyclical process for very many rounds: insert an edge to every  $y_i$ , delete the edge to every  $x_i$ , insert an edge to every  $x_i$ , delete the edge to every  $y_i$ , insert the edge to every  $y_i$ , and so on. Note that the edge from  $v$  to  $v'$  is never deleted. We claim that as we continue this process for a long time,  $\Pr[\text{MATE}(v) = v'] \rightarrow 1$ . The reason is that the algorithm of Baswana et al. only picks a new mate for  $v$  when the previous matching edge was deleted. But the process repeatedly deletes all edges except  $(v, v')$ , so it will continually pick a new matching edge at random until it eventually picks  $(v, v')$ , at which point  $v'$  will remain the mate of  $v$  throughout the process. The original algorithm of Baswana et al. is thus *not* worst-case expected: if the adversary starts with the above (long) sequence and then deletes  $(v, v')$ , this deletion is near-guaranteed to be expensive because  $\Pr[\text{MATE}(v) = v'] \sim 1$ .

One way to overcome this issue is to give  $v$  a small probability of resetting its matching edge every time a new vertex enters or is removed from  $N_{<i}(v)$ ; this would ensure that even if  $(v, v')$  becomes the matching edge at some point during the process, it will not stick forever. In fact we will show that it is enough to have a small reset probability only for edges added to  $N_{<i}(v)$ , not for edges removed.

#### 4.2.1 List of Changes to the Baswana et al. Algorithm

The algorithm maintains the same lists  $\mathcal{O}_v$  and  $\mathcal{E}_v^i$ , as well as the counters  $\phi_v(i)$ .

- Invariants 1-4 are exactly the same as above.
- Define  $C$  to be a sufficiently large constant used by the algorithm.
- Whenever the algorithm executes  $\text{INCREMENT-}\phi(v, i)$  for a vertex  $v$  with  $\text{LEVEL}(v) < i$ , the algorithm: **1)** performs  $\text{RISE}(v, \text{LEVEL}(v), i)$  with probability  $p^{\text{rise}} = C \log(n)/4^i$ . We call this a *probabilistic rise*. **2)** always performs  $\text{RISE}(v, \text{LEVEL}(v), i)$  if  $\phi_v(i)$  increases from  $4^i - 1$  to  $4^i$ ; we call this a *threshold rise*. (The original algorithm of Baswana et al. only performed threshold rises. Our new version modifies line 13 in the pseudocode of Procedure  $\text{PROCESS-FREE-VERTICES}$  of [BGS18], as well as the paragraph “Handling insertion of an edge” in Section 4.2.)
- **Matching Property\*** If a vertex  $v$  at level  $i > -1$  is (temporarily) unmatched and  $|N_{<i}(v)| \geq 4^i/(32C \log(n))$ , then  $v$  will pick a new mate *uniformly at random* from  $N_{<i}(v)$ . If  $|N_{<i}(v)| < 4^i/(32C \log(n))$ , then  $v$  will pick a new mate *uniformly at random* from  $N_{<i}(v)$ .



$4^i/(32C \log(n))$ , then  $v$  falls to level  $i - 1$  and is recursively processed from there. Note that Matching Property\* is identical to Matching Property above, but with  $4^i/(32C \log(n))$  instead of  $4^i$ . This leads to the following change in procedure  $\text{FIXFREEVERTEX}(v)$ . Let  $i = \text{LEVEL}(v)$ : if  $|N_{<i}(v)| \geq 4^i/(32C \log(n))$ , then the algorithm executes  $\text{GENERICRANDOMSETTLE}(v, i)$ , and if  $|N_{<i}(v)| < 4^i/(32C \log(n))$ , then it executes  $\text{FALL}(v, i)$ . (Our version modifies line 5 of Procedure  $\text{FALLING}$  of [BGS18]).

- We make the following change to procedure  $\text{FALL}(v, i)$ . Recall that as a result of  $v$  falling to level  $i - 1$ ,  $v$  now belongs to  $N_{<i}(u)$  for every neighbor  $u$  of  $v$  at level  $i$ . Each such neighbor  $u$  then executes  $\text{RESETMATCHING}(u, i)$  with probability  $p^{\text{reset}} = 1/4^{i+3}$ .  $\text{RESETMATCHING}(u, i)$  simply picks a new matching edge for  $u$  by removing edge  $(u, \text{MATE}(u))$  from the matching and then calling  $\text{FIXFREEVERTEX}(u)$  and  $\text{FIXFREEVERTEX}(\text{MATE}(u))$ . (Our version modifies lines 3 and 4 in Procedure  $\text{FALLING}$  of [BGS18]).

**Pseudocode.** We give the pseudocode for the whole modified algorithm in Algorithms 1 and 2. The pseudocode shows how the basic procedures of the algorithm (e.g.  $\text{RISE}$ ,  $\text{FALL}$ ,  $\text{FIXFREEVERTEX}$ ) call each other. We note that in addition to the work shown in the pseudocode, each change of the level of a vertex in the hierarchy is also accompanied by straightforward “bookkeeping” work that changes the corresponding sets  $\mathcal{O}_v$  and  $\mathcal{E}_v^i$ ; the bookkeeping work maintains the invariant that every edge is owned by the endpoint at higher level (ties can be broken arbitrarily), and that  $\mathcal{E}_v^i$  contains all edges  $(v, w)$  such that  $\text{LEVEL}(w) = i$  and  $(v, w) \notin \mathcal{O}_v$ . For example, if a vertex falls from level  $i$  to level  $i - 1$ , then for every edge  $(v, w) \in \mathcal{O}_v$  we do the following: if  $\text{LEVEL}(w) < i$  then we transfer  $(v, w)$  from  $\mathcal{E}_w^i$  to  $\mathcal{E}_w^{i-1}$ ; if  $\text{LEVEL}(w) = i$ , then we transfer  $(v, w)$  from  $\mathcal{O}_v$  to  $\mathcal{O}_w$ , we remove  $(v, w)$  from  $\mathcal{E}_v^i$ , and we add  $(v, w)$  to  $\mathcal{E}_v^i$ . To avoid clutter, we omit this bookkeeping work from the pseudocode. Note that the bookkeeping only requires to process the edges owned by  $v$  and can thus be done in time  $O(4^{\text{LEVEL}(v)})$ . Similarly, we sometimes explicitly need to compute the sets  $N_{<\text{LEVEL}(v)}(v)$  and  $N_{=\text{LEVEL}(v)}(v)$ , which can also be done in time  $O(4^{\text{LEVEL}(v)})$ .

The matching maintained by the algorithm in the pseudocode is denoted by  $\mathcal{M}$ . Note that for technical reasons calls of  $\text{FIXFREEVERTEX}(v)$  for vertices  $v$  in our algorithm are not executed immediately. Instead, we maintain a global queue  $Q$  of vertices  $v$  for which we still need to perform  $\text{FIXFREEVERTEX}(v)$ . We avoid adding the same call to the queue twice by additionally managing the elements in the queue in a dynamic hash table.

#### 4.2.2 Correctness of the Modified Algorithm

To show the correctness of the modified algorithm we need to show that it fulfills Invariants 1–4 and that Matching Property\* holds. We will do so in this subsection. Termination is guaranteed in the next section, which shows that the expected time to process an adversarial edge insertion/deletion is finite.

**Lemma 4.2.** *Invariants 1–4, and the MatchingProperty\* hold before and after each edge insertion.*

*Proof. Invariant 1:* Invariant 1 is automatically ensured by the “bookkeeping work” described above, since whenever a vertex  $v$  changes level, the bookkeeping work modifies  $\mathcal{O}_v$  accordingly.

*Invariant 2:* To show invariant 2 we need to show that (a) every vertex on a level larger than  $-1$  is matched and (b) every vertex on level  $-1$  is free. We show the claim by induction on the number of updates. Initially the graph is empty and every vertex is unmatched and on level  $-1$ . Thus, the

---

**Algorithm 1:** Fully Dynamic Maximal Matching Algorithm

---

```
1 Procedure DELETE( $u, v$ ) // Process deletion of edge  $(u, v)$ 
2   if  $(u, v) \in \mathcal{M}$  then
3     Initialize empty queue  $Q$ 
4     Perform bookkeeping work for deletion of  $(u, v)$ 
5     Add  $u$  to  $Q$  (if it is not already contained in  $Q$ ) // Execute FIXFREEVERTEX( $u$ ) later
6     Add  $v$  to  $Q$  (if it is not already contained in  $Q$ ) // Execute FIXFREEVERTEX( $v$ )
7     later
8     PROCESSQUEUE()
9   else
10    Perform bookkeeping work for deletion of  $(u, v)$ 
11 Procedure INSERT( $u, v$ ) // Process insertion of edge  $(u, v)$ 
12   Initialize empty queue  $Q$ 
13   Perform bookkeeping work for insertion of  $(u, v)$ 
14   foreach  $j > \text{LEVEL}(v)$  do INCREMENT- $\phi(u, j)$ 
15   foreach  $j > \text{LEVEL}(u)$  do INCREMENT- $\phi(u, j)$ 
16   if  $\text{LEVEL}(u) < \text{LEVEL}(v)$  then
17     With probability  $p^{\text{reset}}$  do RESETMATCHING( $v$ )
18   if  $\text{LEVEL}(v) < \text{LEVEL}(u)$  then
19     With probability  $p^{\text{reset}}$  do RESETMATCHING( $u$ )
20   PROCESSQUEUE()
21 Procedure PROCESSQUEUE()
22   while  $Q$  is not empty do
23     Pick arbitrary vertex  $v$  from  $Q$ 
24     FIXFREEVERTEX( $v$ )
```

---

---

**Algorithm 2:** Fully Dynamic Maximal Matching Algorithm

---

```
24 Procedure RESETMATCHING( $v$ ) // Called only if LEVEL( $v$ ) > -1
25    $w = \text{MATE}(v)$ 
26    $\mathcal{M} \leftarrow \mathcal{M} \setminus \{(v, w)\}$  // Unmatch  $v$  and  $w$ 
27   Add  $v$  to  $Q$  (if it is not already contained in  $Q$ ) // Execute FIXFREEVERTEX( $v$ ) later
28   Add  $w$  to  $Q$  (if it is not already contained in  $Q$ ) // Execute FIXFREEVERTEX( $w$ ) later
29 Procedure FIXFREEVERTEX( $v$ )
30    $i \leftarrow \text{LEVEL}(v)$ 
31   if  $i > -1$  and  $v$  is unmatched then
32     if  $|N_{<i}(v)| \geq 4^i / (32C \log(n))$  then
33       Compute  $N_{<i}(v)$ 
34       GENERICRANDOMSETTLE( $v, i$ )
35     else
36       FALL( $v, i$ )
37 Procedure GENERICRANDOMSETTLE( $v, i$ ) // Called only if  $|N_{<i}(v)| \geq 4^i / (32C \log(n))$ 
38   Pick  $w \in N_{<i}(v)$  uniformly at random
39   Perform bookkeeping work to move  $w$  to level  $i$ 
40   if  $\exists(w, x) \in \mathcal{M}$  then // Check if  $w$  is matched with some neighbor  $x$ 
41      $\mathcal{M} \leftarrow \mathcal{M} \setminus \{(w, x)\}$  // Unmatch  $w$  and  $x$ 
42     Add  $x$  to  $Q$  (if it is not already contained in  $Q$ ) // Execute FIXFREEVERTEX( $x$ ) later
43    $\mathcal{M} \leftarrow \mathcal{M} \cup (v, w)$  // Match  $v$  and  $w$ 
44 Procedure FALL( $v, i$ )
45   Compute  $N_{<i}(v)$  and  $N_{=i}(v)$ 
46   Perform bookkeeping work to to move  $v$  from level  $i$  to level  $i - 1$ 
47   foreach  $w \in N_{<i}(v)$  do
48     INCREMENT- $\phi(w, i)$ 
49   foreach  $w \in N_{=i}(v)$  do
50     With probability  $p^{\text{reset}}$  do RESETMATCHING( $w$ )
51   Add  $v$  to  $Q$  (if it is not already contained in  $Q$ ) // Execute FIXFREEVERTEX( $v$ ) later
52 Procedure INCREMENT- $\phi(v, i)$ 
53   Perform bookkeeping work
54   With probability  $p^{\text{rise}}$  do RISE( $v, \text{LEVEL}(v), i$ ) // Probabilistic rise
55   if  $\phi_v(i) \geq 4^i$  then RISE( $v, \text{LEVEL}(v), i$ ) // Threshold rise
56 Procedure RISE( $v, i, j$ )
57   Perform bookkeeping work to move  $v$  from level  $i$  to level  $j$ 
58   if  $\exists(v, w) \in \mathcal{M}$  then // Check if  $v$  is matched with some neighbor  $w$ 
59      $\mathcal{M} \leftarrow \mathcal{M} \setminus \{(v, w)\}$  // Unmatch  $v$  and  $w$ 
60     Add  $w$  to  $Q$  (if it is not already contained in  $Q$ ) // Execute FIXFREEVERTEX( $w$ )
        later
61   Add  $v$  to  $Q$  (if it is not already contained in  $Q$ ) // Execute FIXFREEVERTEX( $v$ ) later
```

---

claim holds. Assume now that the claim holds before an edge insertion or deletion. We will show that it holds also after the edge insertion or deletion was processed.

We first show (a). A vertex  $v$  on level larger than  $-1$  can violate Invariant 2 if (1) its matched edge was deleted or (2) it became unmatched in procedure `GENERICRANDOMSETTLE` or `RISE`. In both cases  $v$  is placed on the queue (if it is not already there). Then the current procedure completes and then other calls to `FIXFREEVERTEX` might be executed before the call `FIXFREEVERTEX(v)` is started. Thus, it is possible that the hierarchy has changed between the time when  $v$  was placed on the queue and the time when its execution starts. This is the reason why `FIXFREEVERTEX(v)` first checks whether  $v$  is still on a level larger than  $-1$  and whether it is still unmatched. If this is not the case,  $v$  fulfills Invariant 2. If this is still the case, then the main body of `FIXFREEVERTEX(v)` is executed, which either matches  $v$  with `GENERICRANDOMSETTLE(v, LEVEL(v))` or it decreases the level of  $v$  (if  $v$  does not have “enough” neighbors on levels below `LEVEL(v)`) and then places  $v$  on the queue. As the update algorithm does not terminate until the queue is empty, it is guaranteed that all vertices fulfill (a) at termination of the update.

To show (b) note that a vertex  $u$  is only matched in procedure `GENERICRANDOMSETTLE` and in this case it needs to be on a level  $i$  such that either the vertex  $u$  itself or its newly matched partner  $v$  fulfill the property that there is at least one neighbor in a level *below* level  $i$ . As  $-1$  is the lowest level, it follows that  $i > -1$ , which shows (b).

*Invariant 3:* For Invariant 3 we need to show for all  $j > \text{LEVEL}(v)$  that  $|N_{<j}(v)| < 4^j$ . We show the claim by induction on the number of updates. The property certainly holds at the beginning of the algorithm when there are no edges. Assume it was true before the current edge update. We will show that then it also holds after the current edge update. Let  $v$  be a vertex. The set  $N_{<j}(v)$  only increases if a neighbor  $w$  drops from a level above  $j$  to a level below  $j$ . Since each execution of `FALL` decreases the level of a vertex only by one, the set  $N_{<j}(v)$  can only increase if a neighbor  $w$  drops from  $j$  to  $j - 1$ . As a consequence it follows that the sets  $N_{<k}(v)$  for *all*  $k \neq j$  are unchanged, and, thus,  $|N_{<k}(v)| < 4^k$  for all  $k \neq j$ , i.e., there is only one set  $N_{<\cdot}(v)$  that might reach its threshold value, namely  $N_{<j}(v)$ . The fall of  $w$  from level  $j$  calls `INCREMENT- $\phi(v, j)$` , which in turn immediately calls `RISE(v, LEVEL(v), j)` if  $|N_{<j}(v)| = 4^j$ . After  $v$  has moved up to level  $j$ , it holds that for all  $k > j$  that  $|N_{<k}(v)| < 4^k$  as this was also true before the rise. Thus, Invariant 3 holds again for  $v$ .

*Invariant 4:* For Invariant 4 we have to show that the endpoints of every matched edge are at the same level. Note that two vertices  $v$  and  $w$  only become matched in procedure `GENERICRANDOMSETTLE` and right before that the vertex (out of the two) on the lower level is “pulled up” to the level of the higher vertex. Thus, both are at the same level when they are matched.

*MatchingProperty\*:* Finally `MatchingProperty*` holds for every vertex  $v$  for the following reason: As soon as a vertex becomes unmatched,  $v$  is placed on the queue. Whenever this call is executed, it checks whether  $|N_{<i}(v)| \geq 4^i / (32C \log(n))$ , where  $i = \text{LEVEL}(v)$ , is fulfilled and if so, it calls `GENERICRANDOMSETTLE(v, i)`, which in turn picks a random neighbor of  $N_{<i}(v)$  and matches  $v$  with it. If, however,  $|N_{<i}(v)| < 4^i / (32C \log(n))$ , then `FIXFREEVERTEX(v)` calls the procedure `FALL(v, i)`. The procedure `FALL` checks again whether it still holds that  $|N_{<i}(v)| < 4^i / (32C \log(n))$ , and if so  $v$  is moved one level down. Since in this case the vertex is still unmatched, `FALL(v, i)` also inserts  $v$  into the queue, which later on results in a call to `FIXFREEVERTEX(v)` executed on  $v$ ’s new level  $i - 1$ . Thus,  $v$  continues to fall until it either reaches a level  $i$  where  $|N_{<i}(v)| \geq 4^i / (32C \log(n))$  (in which case it is matched there) or until it reaches level  $-1$ , in which case  $|N_{<i}(v)| = 0 < 1$ . Hence, in either case `MatchingProperty*` holds.  $\square$

### 4.2.3 Analysis of the Modified Algorithm

Note that each procedure used by the algorithm (e.g. FALL or FIXFREEVERTEX) incurs two kinds of costs:

- **Bookkeeping work:** As discussed above, if the procedure changes the level of a vertex, the algorithm must do bookkeeping work to maintain the various  $\mathcal{O}_v$  and  $\mathcal{E}_v^i$  data structures .
- **Recursive work:** a change in the hierarchy could lead other vertices to violate one of the invariants, and so lead to the execution of further procedures.

We start with the easier task of analyzing the bookkeeping work. The FALL, GENERICRANDOMSETTLE, and FIXFREEVERTEX procedures all require  $O(4^i)$  time to process a vertex  $v$  at level  $i$ : this is because the bookkeeping work only requires us to look at  $\mathcal{O}_v$ , which by Invariant 3 contains at most  $O(4^i)$  edges. We now analyze the bookkeeping required for procedure RISE:

**Lemma 4.3.** *RISE( $v, i, j$ ) requires time  $O(4^j)$ .*

*Proof.* Although they do not state it as such, this lemma holds for the original Baswana et al. algorithm as well. When  $v$  rises from level  $i$  to level  $j$ , the algorithm performs bookkeeping of two sorts. Firstly, every vertex  $u$  that is owned by  $v$  after the rise must update  $\mathcal{E}_u^i$  and  $\mathcal{E}_u^j$ ; by Invariant 3 there are at most  $O(4^j)$  neighbors to update. Secondly, every neighbor  $u$  of  $v$  in  $N_{<j}(v)$  must execute DECREMENT- $\phi(u, k)$  for every  $\max\{i, \text{LEVEL}(u)\} < k \leq j$ . The total cost is upper bounded by

$$O(N_{<j}(v) \cdot (j - i) + \sum_{k=i+1}^{j-1} |N_{=k}(v)| \cdot (j - k)), \quad (6)$$

where  $N_{=k}(v)$  is the number of neighbors of  $v$  at level  $k$ . But note that for  $k > i$ ,  $|N_{=k}(v)| < 4^k + 1$ , since otherwise  $v$  would violated Invariant 3 for level  $k$  even before the procedure call that led to RISE( $v, i, j$ ). Similarly,  $|N_{<i}(v)| < 4^{i+1} + 1$ . Plugging these bounds into Equation 6 yields a geometric sum totalling  $O(4^j)$ .  $\square$

Before analyzing the recursive work, we bound the probability that INCREMENT- $\phi(v, i)$  calls RISE. The chance of a probabilistic rise is always the same  $p^{\text{rise}} = \Theta(\log(n)/4^i)$ . We now bound threshold-rises:

**Lemma 4.4.** *The sequence of oblivious updates predefined by the adversary gives some probability distribution on the point in time (in the sequence of updates) to process the first call to INCREMENT- $\phi$ , the second call to INCREMENT- $\phi$ , the third one, and so on. Claim: for any  $k$ , it holds with high probability that the  $k$ th call to INCREMENT- $\phi$  does not lead to a threshold rise.*

*Proof.* Let  $B_{v,i}$  be the bad event that the  $k$ th call to INCREMENT- $\phi$  increments  $\phi_v(i)$  from  $4^i - 1$  to  $4^i$ . It is enough to show that  $\neg B_{v,i}$  occurs with high probability; we can then union bound over all pairs  $(v, i)$ . Let  $t_k$  be the time at which this  $k$ th call to INCREMENT- $\phi$  occurs, and note that at the beginning of time  $t_k$  we have  $\text{LEVEL}(v) < i$ , since otherwise we would have  $\phi_v(i) = 0$  and no threshold rise would occur. Now, let  $t$  be the earliest point in time such that  $\text{LEVEL}(v) < i$  in the entire time interval from  $t$  to  $t_k$ . It is not hard to see that because Matching Property\* only allows a vertex to fall to below level  $i$  when  $|N_{<i}(v)| < 4^j/(32C \log(n))$ , it must be the case that at time  $t$  we have  $\phi_i(v) < 4^i/(32C \log(n)) < 4^i/2$ . Thus, there must have been at least  $4^i/2$  calls to

INCREMENT- $\phi(v, i)$  in time interval  $(t, t_k)$ , and by the assumption that  $\text{LEVEL}(v) < i$  in this entire time interval, none of these  $4^i/2$  calls to INCREMENT- $\phi(v, i)$  led to a probabilistic rise. But each probabilistic rise occurs independently with probability  $C \log(n)/4^i$  (for a sufficiently large  $C$ ), so a simple Chernoff bound shows us that with high probability this event does not occur.  $\square$

We now turn to bounding the recursive work incurred by a procedure. Let us first define this more formally. Bringing attention to the pseudocode, we note that each procedure is either directly called by some previous procedure, or, in the case of `FIXFREEVERTEX`, it is added to the queue by a previous procedure; we say that the called procedure is *caused* by the calling procedure. For example, a procedure `FALL` leads to many calls to INCREMENT- $\phi$ , and so can potentially cause many calls to procedure `RISE`. We can thus construct a causation tree, where the parent procedure causes all the children procedure calls. We then say that the *total work* of a procedure is the bookkeeping work of that procedure, plus the total work of all of its children in the causation tree; equivalently, the total work of a procedure is the total bookkeeping work required to process all of its descendants in the causation tree.

We now show that for any vertex  $v$  at level  $i = \text{LEVEL}(v)$ , the expected total work of `FALL`( $v, i$ ), `GENERICRANDOMSETTLE`( $v, i$ ), or `FIXFREEVERTEX`( $v$ ) is at most  $O(4^i)$ . (We handle the procedure `RISE` later.) To this end, we need the following notation. When we refer to the vertex hierarchy  $H$  at time  $t$ , this includes the level assigned to each vertex at time  $t$ , as well as the set of edges in the matching at time  $t$ . The vertex hierarchy thus fully captures the current state of the algorithm. Note that the time to process any procedure at time  $t$  depends on two things: the vertex hierarchy at time  $t$ , and the random coin flips made after time  $t$ . Thus, we can define  $E_i^{\text{fall}}(v, H)$  to be the expected total work to process `FALL`( $v, i$ ) given that the state of the current hierarchy is  $H$ , where the expectation is taken over all coin flips made after time  $t$ . (We assume that in the hierarchy  $H$  vertex  $v$  has level  $i$ , since otherwise `FALL`( $v, i$ ) is not a valid procedure call.) We define  $E_i^{\text{settle}}(v, H)$  and  $E_i^{\text{free}}(v, H)$  analogously. We say that some hierarchy  $H$  is valid if it satisfies all of the hierarchy invariants above; note that our dynamic algorithm always maintains a valid hierarchy.

We are now ready to introduce our key notation. We let  $E_i^{\text{fall}}$  be the maximum of all  $E_i^{\text{fall}}(v, H)$ , where the maximum is taken over all vertices  $v$ , and all valid hierarchies  $H$  in which  $v$  has level  $i$ . Define  $E_i^{\text{settle}}$  and  $E_i^{\text{free}}$  accordingly. Define  $E_i^{\text{max}} = \max\{E_i^{\text{fall}}, E_i^{\text{settle}}, E_i^{\text{free}}\}$ . Note that because  $E_i^{\text{max}}$  takes the maximum over all valid hierarchies, it is an upper bound on the expected time to process *any* update at level  $i$ . We now prove a recursive formula for bounding  $E_i^{\text{max}}$ .

**Lemma 4.5.**  $E_i^{\text{max}} \leq O(4^i) + 2E_{i-1}^{\text{max}}$

*Proof.* We first show that  $E_i^{\text{settle}} \leq O(4^i) + E_{i-1}^{\text{max}}$ . `GENERICRANDOMSETTLE`( $v, i$ ) picks some random mate  $v'$  for  $v$  with  $\text{LEVEL}(v') < i$ , performs  $O(4^i)$  bookkeeping work to move  $w$  to level  $i$  (Lemma 4.3), and then causes a single other procedure call, namely, `FIXFREEVERTEX`(`OLD-MATE`( $v'$ )); this caused procedure call occurs at some level less than  $i$ , so the expected total work can be upper bounded by  $E_{i-1}^{\text{max}}$ .

Now consider `FIXFREEVERTEX`( $v$ ), where  $i = \text{LEVEL}(v)$ . As discussed above, the bookkeeping work of this procedure is at most  $O(|\mathcal{O}_v|) = O(4^i)$ . The algorithm then causes one other procedure call: either `GENERICRANDOMSETTLE`( $v, i$ ) or `FALL`( $v, i$ ), depending on the size of  $N_{<i}(v)$ . We have already bounded  $E_i^{\text{settle}}$ , so all that remains is to bound  $E_i^{\text{fall}}$ .

Recall that the algorithm only executes `FALL`( $v, i$ ) when  $N_{<i}(v) < 4^i/(32C \log(n))$ . The procedure `FALL` requires the standard  $O(|\mathcal{O}_v|) = O(4^i)$  bookkeeping work, and it also causes a call to `FIXFREEVERTEX`( $v$ ) at level  $i-1$ , which has  $E_{i-1}^{\text{max}}$  expected total work. But unlike the other procedure, `FALL`( $v, i$ )

can also cause additional procedure calls at level  $i$ . This can happen in two ways: **1)** Each neighbor  $u$  of  $v$  at level  $i$  executes  $\text{RESETMATCHING}(u, i)$  with probability  $1/4^{i+3}$ . **2)** Each neighbor  $u$  of  $v$  at level  $i - 1$  or less executes  $\text{INCREMENT-}\phi(u, i)$ . This has a small chance of resulting in  $\text{RISE}(u, \text{LEVEL}(u), i)$ , followed by  $\text{FIXFREEVERTEX}(u)$ , where  $\text{LEVEL}(u) = i$ , and  $\text{FIXFREEVERTEX}(\text{OLD-MATE}(u))$ , where  $\text{LEVEL}(\text{OLD-MATE}(u)) \leq i - 1$ .

Let  $X^{\text{reset}}$  be the number of  $\text{RESETMATCHING}(u, i)$  triggered by the fall of  $v$ , and let  $X^{\text{rise}}$  be the number of  $\text{RISE}(u, \text{LEVEL}(u), i)$  triggered by the fall. Note that  $\mathbb{E}[X^{\text{reset}}] = 1/16$  because by Invariant 3,  $v$  has at most  $4^{i+1}$  neighbors at level  $i$ , and each of them has a  $p^{\text{reset}} = 1/4^{i+3}$  probability of being reset. We now argue that  $\mathbb{E}[X^{\text{rise}}] = 1/16$ . By Matching Property\*,  $v$  has at most  $4^i/(32C \log(n))$  neighbors  $u$  at lower level before the fall, each of which executes  $\text{INCREMENT-}\phi(u, i)$ . Our modification to the original algorithm ensures that this increment has a  $p^{\text{rise}} = C \log(n)/4^i$  chance of inducing a probability-rise, and by Lemma 4.4 the probability of a threshold-rise is negligible, so for simplicity we upper bound it by  $C \log(n)/4^i$ . Thus:  $\mathbb{E}[X^{\text{rise}}] = [4^i/(16C \log(n))][2C \log(n)/4^i] = 1/16$ .

We now consider the total work to process a fall. Firstly, the fall automatically triggers  $O(4^i)$  bookkeeping work plus it causes a procedure call at level  $i - 1$ ; by definition, the expected total work to process this additional procedure call can be upper bounded with  $\mathbf{E}_{i-1}^{\max}$ . We also have to do additional work for each matching call to  $\text{RESETMATCHING}$  or  $\text{RISE}$ . Each reset triggers two additional procedure calls at level  $i$ ; the expected time to process each of these procedure calls can be upper bounded with  $\mathbf{E}_i^{\max}$ . Note that this upper bound allows to achieve a crucial probabilistic independence: although the value of  $X^{\text{rise}}$  might be correlated with the time to process these calls to  $\text{RISE}$  (both depend on the current hierarchy), the value of  $X^{\text{rise}}$  is *completely independent* from  $\mathbf{E}_i^{\max}$ , since the latter takes the maximum over all possible hierarchies, and so does not depend on the current hierarchy. Similarly, each procedure requires  $O(4^i)$  bookkeeping work, plus it causes a call to a procedure at level  $i$  and another at level less than  $i$ , both of which we upper bound with  $\mathbf{E}_i^{\max}$ . Putting it all together, we can write a recursive formula for  $\mathbf{E}_i^{\max}$ .

$$\begin{aligned} \mathbf{E}_i^{\max} &\leq O(4^i) + \mathbf{E}_{i-1}^{\max} + (2\mathbf{E}_i^{\max} + O(4^i)) \sum_{k=1}^{\infty} k \Pr[X^{\text{reset}} + X^{\text{rise}} = k] \\ &\leq O(4^i) + \mathbf{E}_{i-1}^{\max} + (2\mathbf{E}_i^{\max} + O(4^i))(\mathbb{E}[X^{\text{rise}} + X^{\text{reset}}]) \\ &= O(4^i) + \mathbf{E}_{i-1}^{\max} + (2\mathbf{E}_i^{\max} + O(4^i))(1/8) < O(4^i) + \mathbf{E}_{i-1}^{\max} + \mathbf{E}_i^{\max}/2. \quad \square \end{aligned}$$

**Corollary 4.6.** *The expected total work for a call to  $\text{FIXFREEVERTEX}$ ,  $\text{FALL}$ ,  $\text{GENERICRANDOMSETTLE}$ ,  $\text{RISE}$ , or  $\text{RESETMATCHING}$  at level  $i$  is  $O(4^i)$ , where  $\text{RISE}(v, i, j)$  is said to be a procedure call at level  $j$ .*

*Proof.* Solving the recurrence relation in Lemma 4.5 yields  $\mathbf{E}_i^{\max} = O(4^i)$ , which gives us the desired bound for  $\text{FIXFREEVERTEX}$ ,  $\text{FALL}$ , and  $\text{GENERICRANDOMSETTLE}$ . Procedure  $\text{RESETMATCHING}$  simply makes two calls to  $\text{FIXFREEVERTEX}$ , so the same  $O(4^i)$  bound applies. Procedure  $\text{RISE}$  requires  $O(4^i)$  bookkeeping work (Lemma 4.3), and then causes at most two other calls to procedures other than  $\text{RISE}$ , each of which we know has expected total work  $O(4^i)$ .  $\square$

Now that we have analyzed the time to process the individual procedure calls, we turn our attention to the time work required to process an adversarial edge insertion/deletion. Note that the most direct reason the algorithm might have to perform a procedure call at level  $i$  is the deletion of matching edge  $(v, \text{MATE}(v))$  at level  $i$ . Our modifications to the algorithm allow us to do without the charging argument of Baswana et al., and instead directly bound the probability that a deleted

edge  $(u, v)$  is a matching edge. Note that for  $(u, v)$  to be a matching edge, it must have been chosen by a  $\text{GENERICRANDOMSETTLE}(u, i)$  or  $\text{GENERICRANDOMSETTLE}(v, i)$  for some level  $i$ . There are thus  $O(2 \log(n))$  possible procedure calls that could have created this matching edge: we bound the probability of each separately.

**Lemma 4.7.** *Let  $(u, v)$  be any edge at any time during the update sequence, and let  $0 \leq \ell \leq \lfloor \log_4(n) \rfloor$  be any level in the hierarchy. Then:  $\Pr[(u, v)$  is a matching edge that was chosen by  $\text{GENERICRANDOMSETTLE}(v, \ell)] = O(\log^2(n)/4^\ell)$ , where the probability is over all random choices made by the algorithm.*

*Proof.* Let  $t^*$  be the current time when edge  $(u, v)$  is deleted. Let  $E^{\text{matching}}$  be the event that  $(u, v)$  is a matching edge that was chosen by  $\text{GENERICRANDOMSETTLE}(v, \ell)$ . We assume that  $\text{LEVEL}(v) = \ell$  at time  $t^*$ , since otherwise  $\Pr[E^{\text{matching}}] = 0$ . Recall that  $\text{GENERICRANDOMSETTLE}(v, \ell)$  can be called for three reasons: **1:**  $v$  moves to level  $\ell$  from another level **OR** **2:** The algorithm executes  $\text{RESETMATCHING}$ : this can either be a  $\text{RESETMATCHING}(v, \ell)$  due to an addition to  $N_{<\ell}(v)$ , or a  $\text{RESETMATCHING}(\text{MATE}(v), \ell)$  due to an addition to  $N_{<\ell}(\text{MATE}(v))$ . **3:** the matching edge  $(v, \text{MATE}(v))$  is deleted by the adversary. Let  $t_{\text{level}}^{\text{crit}}$  be the last update before  $t^*$  where  $v$  changed levels, and note that at time  $t_{\text{level}}^{\text{crit}}$   $v$  was assigned to level  $\ell$ . Let  $t_{\text{rematch}}^{\text{crit}}$  be the last update before  $t^*$  at which  $\text{GENERICRANDOMSETTLE}(v, \ell)$  was called as a result of some  $\text{RESETMATCHING}$ . Let  $t^{\text{crit}} = \max\{t_{\text{level}}^{\text{crit}}, t_{\text{rematch}}^{\text{crit}}\}$  be the later of these two times. Note that at all times  $t$  after  $t^{\text{crit}}$  and before  $t^*$ :  $\text{LEVEL}(v) = \ell$ , and the only possible cause for  $\text{GENERICRANDOMSETTLE}(v, \ell)$  is that the current matching edge  $(v, \text{MATE}(v))$  is deleted at time  $t$ .

Before continuing with the proof, we briefly discuss the naive approach to the proof, and why it fails to work. Note that if we focus on any single call to  $\text{GENERICRANDOMSETTLE}(v, \ell)$  in the time interval  $(t^{\text{crit}}, t^*)$ , Matching Property\* guarantees that the probability of the particular edge  $(u, v)$  being picked as the matching edge is very small:  $O(\log(n)/4^\ell)$ . Now, let  $t$  be the *last* time before  $t^*$  that  $\text{GENERICRANDOMSETTLE}(v, \ell)$  is called, and note that the matching edge at time  $t^*$  is precisely the matching edge picked at time  $t$ . It is tempting to (falsely) argue that at time  $t$  the probability that  $\text{GENERICRANDOMSETTLE}(v, \ell)$  picked edge  $(u, v)$  is at most  $O(\log(n)/4^\ell)$ . But this might not be true, because although  $\text{GENERICRANDOMSETTLE}(v, \ell)$  picks an edge uniformly at random from many options, the fact that we condition on  $t$  being the *last* random settle before  $t^*$  can greatly skew the distribution. Consider, for illustration, the update sequence at the beginning of Section 4.2: the sequence repeatedly deletes all edges other than  $(u, v)$ , so any edge other than  $(u, v)$  is unlikely to be the *last* matching edge, since it will soon be deleted. To overcome this issue, we now present a more complex analysis that (loosely speaking) bounds the total number of times  $\text{GENERICRANDOMSETTLE}(v, \ell)$  is called in the time interval  $(t^{\text{crit}}, t^*)$ .

Whenever a neighbor  $u$  is added to  $N_{<\ell}(v)$ , we call this a rematch opportunity for  $v$ . We say that  $t^{\text{crit}}$  has gap  $\gamma$ , for a non-negative integer  $\gamma$ , if the number of rematch opportunities after  $t^{\text{crit}}$  and before  $t^*$  is in the interval  $[\gamma 4^\ell, (\gamma + 1)4^\ell)$ . Let  $E_\gamma^{\text{gap}}$  be the event that  $t^{\text{crit}}$  has gap  $\gamma$ . Clearly

$$\Pr[E^{\text{matching}}] = \sum_{\gamma=0}^{\infty} \Pr[E_\gamma^{\text{gap}}] \cdot \Pr[E^{\text{matching}} | E_\gamma^{\text{gap}}]. \quad (7)$$

We first upper bound  $\Pr[E_\gamma^{\text{gap}}]$ . For this event to occur, there must be at least  $\gamma 4^\ell$  rematch opportunities in the time interval  $(t^{\text{crit}}, t^*)$ , but by definition of  $t^{\text{crit}}$  none of these rematch opportunities can actually result in a  $\text{RESETMATCHING}(v, \ell)$ . Recall that each rematch opportunity independently executes a rematch with probability  $1/4^{\ell+3}$ . Thus:



$$\Pr[E_Y^{\text{gap}}] \leq \left(1 - \frac{1}{4^{\ell+3}}\right)^{\gamma 4^\ell} < e^{-\gamma/64}. \quad (8)$$

We now bound  $\Pr[E^{\text{matching}} \mid E_Y^{\text{gap}}]$ . Note that for  $E^{\text{matching}}$  to occur,  $(u, v)$  must be chosen as a matching edge sometime in time interval  $[t^{\text{crit}}, t^*]$ ; any matching choices made before time  $t^{\text{crit}}$  are irrelevant because the matching is changed at time  $t^{\text{crit}}$ . Let  $X^{\text{settle}}$  be the variable for the number of times  $\text{GENERICRANDOMSETTLE}(v, \ell)$  is called in time interval  $[t^{\text{crit}}, t^*]$ . There is one  $\text{GENERICRANDOMSETTLE}(v, \ell)$  possibly executed at time  $t^{\text{crit}}$ . (Technical note: there might also not be one if  $v$  is pulled to level  $\ell$  by the  $\text{GENERICRANDOMSETTLE}$  of another vertex). All other  $\text{GENERICRANDOMSETTLE}(v, \ell)$  are caused by the deletion of the matching edge  $(v, \text{MATE}(v))$ . Now, by Matching Property\*, each  $\text{GENERICRANDOMSETTLE}(v, \ell)$  picks the particular target edge  $(u, v)$  with probability  $1/|N_{<\ell}(v)| \leq 32C \log(n)/4^\ell$ . We thus have:

$$\Pr[E^{\text{matching}} \mid E_Y^{\text{gap}}] \leq \frac{32C \log(n)}{4^\ell} \sum_{k=1}^{\infty} k \cdot \Pr[X^{\text{settle}} = k \mid E_Y^{\text{gap}}]. \quad (9)$$

It now remains to upper bound  $\mathbf{E}[X^{\text{settle}} \mid E_Y^{\text{gap}}]$ . We say that  $v$  undergoes a deletion whenever we delete an edge  $(u, v)$  with  $u \in N_{<\ell}(v)$  (recall:  $\ell = \text{LEVEL}(v)$ ). We claim that  $v$  undergoes a total of at most  $(\gamma + 5)4^\ell$  deletions in time interval  $[t^{\text{crit}}, t^*]$ . This is because at time  $t^{\text{crit}}$  we have  $\text{LEVEL}(v) = \ell$ , so by Invariant 3  $|N_{<\ell}(v)| \leq 4^{\ell+1}$ , and because we are conditioning on  $t^{\text{crit}}$  having gap  $\gamma$ , we know that there are at most  $(\gamma + 1)4^\ell$  insertions into  $|N_{<\ell}(v)|$  during time interval  $[t^{\text{crit}}, t^*]$ . Let  $D = (\gamma + 5)4^\ell$  be the upper bound on this total number of deletions. Now, whenever we execute  $\text{GENERICRANDOMSETTLE}(v, \ell)$  during time interval  $[t^{\text{crit}}, t^*]$ , let the span of this procedure call be the number of edges in  $N_{<\ell}(v)$  that are deleted before the matching edge chosen by the procedure is deleted. Note that the sum of spans of all the  $\text{GENERICRANDOMSETTLE}(v, \ell)$  is at most  $D$ . We say that a  $\text{GENERICRANDOMSETTLE}(v, \ell)$  has a big span if it has span at least  $4^\ell/(64C \log(n))$ . By Matching Property\* each  $\text{GENERICRANDOMSETTLE}(v, \ell)$  chooses a matching edge at random from at least  $4^\ell/(32C \log(n))$  choices, so it has a big span with probability at least  $1/2$ . Let  $B_j$  ( $B$  for big) be the binary random variable that is 1 if the  $j$ th instance of  $\text{GENERICRANDOMSETTLE}(v, \ell)$  in time interval  $[t^{\text{crit}}, t^*]$  has a big span and 0 otherwise. Note that summing over all instances of  $\text{GENERICRANDOMSETTLE}(v, \ell)$  in time interval  $[t^{\text{crit}}, t^*]$  we have  $\sum_j B_j \leq D/\frac{4^\ell}{64C \log(n)} = (\gamma + 5)64C \log(n)$ . We thus have:

$$\sum_{k=1}^{\infty} k \cdot \Pr[X^{\text{settle}} = k \mid E_Y^{\text{gap}}] \leq \sum_{k=1}^{\infty} k \cdot \Pr[B_1 + B_2 + \dots + B_k \leq (\gamma + 5)64C \log(n)] \quad (10)$$

Recall that for each  $B_i$  we have  $\mathbf{E}[B_i] \geq 1/2$ , independently of all the other  $B_i$ . Thus,  $\mathbf{E}[B_1 + \dots + B_k] \geq k/2$ , so a simple application of the Chernoff bound tells us that the tail-end of the right-most summation in Equation 10 converges to a constant after  $k > (\gamma + 5)256C \log(n)$ , since the probability in the summation decreases exponentially. We thus have an upper bound of  $\mathbf{E}[X^{\text{settle}} \mid E_Y^{\text{gap}}] = O((\gamma + 1) \log(n))$ . Plugging this into Equation 9 yields  $\Pr[E^{\text{matching}} \mid E_Y^{\text{gap}}] = O((\gamma + 1) \frac{\log^2(n)}{4^\ell})$ . Combining this with Equation 8 and plugging into Equation 7 yields the desired  $\Pr[E^{\text{matching}}] = O(\frac{\log^2(n)}{4^\ell} \sum_{\gamma=0}^{\infty} \frac{1+\gamma}{e^{\gamma/64}}) = O(\frac{\log^2(n)}{4^\ell})$ .  $\square$

*Proof of Theorem 1.5.* Let us first consider the insertion of a new edge  $(u, v)$ . The algorithm has to update some subset of  $\mathcal{O}_u, \mathcal{O}_v, \mathcal{E}_u^{\text{LEVEL}(v)}, \mathcal{E}_v^{\text{LEVEL}(u)}$ , and then it has to perform  $O(\log(n))$  calls

to INCREMENT- $\phi(u, i)$  and INCREMENT- $\phi(v, i)$ . Each INCREMENT- $\phi(v, i)$  has a  $p^{\text{rise}} = \Theta(\log(n)/4^i)$  chance of leading to a probability-rise, and a negligible probability of leading to a threshold-rise (Lemma 4.4), so plugging in the cost of a call to RISE from Lemma 4.3, the expected time to process the insertion of  $(u, v)$  is  $O(\sum_{i=0}^{\lfloor \log_4(n) \rfloor} (\log(n)/4^i) \cdot 4^i) = O(\log^2(n))$ .

Let us now consider the deletion of an edge  $(u, v)$ . If  $(u, v)$  is a non-matching edge, then as in the case of insertion, the algorithm only needs to perform  $O(1)$  bookkeeping work and  $O(\log(n))$  calls to DECREMENT- $\phi(u, i)$ ; calls to DECREMENT- $\phi$  do not lead to changes in the hierarchy, so the algorithm stops there. Thus the only case left to consider is the deletion of a matching edge  $(u, v)$ . By Invariant 4  $\text{LEVEL}(u) = \text{LEVEL}(v)$ , so let us say they are both equal to  $\ell$ . The deletion of  $(u, v)$  requires the algorithm to execute  $\text{FIXFREEVERTEX}(u)$  and  $\text{FIXFREEVERTEX}(v)$ , which by Corollary 4.6 requires time  $O(4^\ell)$ . By Lemma 4.7, the total expected update time is thus  $O(\sum_{\ell=0}^{\lfloor \log_4(n) \rfloor} 4^\ell \cdot (\log^2(n)/4^\ell)) = O(\log^3(n))$ .  $\square$

**Explicitly Maintaining a List of the Edges in the Matching.** Both the amortized expected algorithm of Baswana et al. [BGS18], as well as our worst-case expected modification in Theorem 1.5, store the matching in the simplest possible data structure  $D$ : they are both able to maintain a single list containing all the edges of a maximal matching. By Remark 1.2, the high-probability worst-case result in 1.6 stores the matching in a slightly different data structure: it stores  $O(\log(n))$  lists  $D_i$ , along with a pointer to some  $D_j$  such that  $D_j$  is guaranteed to contain the edges of a maximal matching. Dynamic algorithms are typically judged by update and query time, and from this perspective our data structure is equivalently powerful, since we can use the correct  $D_j$  to answer queries about the matching.

However, in some applications, it is desirable to maintain the matching as a single list. The reason is that this way one ensures “continuity” between the updates: for example, the  $O(\log(n))$  update time of Baswana et al. guarantees that every update only changes the underlying maximal matching by  $O(\log(n))$  edges (amortized). This is no longer true of our high-probability worst-case algorithm in Theorem 1.6, because a single update might cause the algorithm to switch the pointer from some  $D_i$  to some  $D_j$ : this still results in a fast update time, but the underlying maximal matching can change by  $\Theta(n)$  edges.

As discussed in Remark 1.2, if we insist on maintaining a single list of edges in the matching, we can do so with almost the same high-probability worst-case update time as stated in Theorem 1.6, but the resulting matching is only  $(2 + \epsilon)$ -approximate, and no longer maximal. This  $(2 + \epsilon)$ -approximation is achieved as follows. The algorithm of Theorem 1.6 stores  $O(\log(n))$  lists  $D_i$ , one of which is guaranteed to be a maximal matching. In particular, this algorithm maintains a fully dynamic data structure with query access to a 2-approximate matching that can output  $\ell$  arbitrary edges of the matching in time  $O(\ell)$ . The very recent black-box reduction in [SS18] takes such a “discontinuous” algorithm for dynamic maximum matching and turns it into a “continuous” one at the cost of an extra  $(1 + \epsilon)$  factor in the approximation. By applying this reduction with  $\epsilon' = \epsilon/2$  we obtain a fully dynamic algorithm for maintaining a matching with an approximation factor of  $2(1 + \epsilon/2) = (2 + \epsilon)$  and a high-probability worst-case update time of  $O(\log^5(n) + 1/\epsilon)$ . The reduction of [SS18] also applies to the dynamic  $(2 + \epsilon)$ -approximate matching algorithms of Arar et al. [Ara<sup>+</sup>18] and Charikar and Solomon [CS18], whose update times we can beat for certain regimes of  $\epsilon$ .

## Acknowledgements

The first author would like to thank Mikkel Thorup and Alan Roytman for a very helpful discussion of the proof of Theorem 1.1. We thank the anonymous reviewers of SODA 2019 for their helpful comments.

## References

- [Abr<sup>+</sup>16] Ittai Abraham, David Durfee, Ioannis Koutis, Sebastian Krinninger, and Richard Peng. “On Fully Dynamic Graph Sparsifiers”. In: *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*. 2016, pp. 335–344 (cit. on pp. 2, 5).
- [ACK17] Ittai Abraham, Shiri Chechik, and Sebastian Krinninger. “Fully dynamic all-pairs shortest paths with worst-case update-time revisited”. In: *Proceedings of the Symposium on Discrete Algorithms (SODA)*. 2017, pp. 440–452 (cit. on p. 2).
- [AFI06] Giorgio Ausiello, Paolo Giulio Franciosa, and Giuseppe F. Italiano. “Small Stretch Spanners on Dynamic Graphs”. In: *Journal of Graph Algorithms and Applications* 10.2 (2006). Announced at ESA’05, pp. 365–385 (cit. on p. 4).
- [Ara<sup>+</sup>18] Moab Arar, Shiri Chechik, Sarel Cohen, Cliff Stein, and David Wajc. “Dynamic Matching: Reducing Integral Algorithms to Approximately-Maximal Fractional Algorithms”. In: *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*. 2018, 7:1–7:16 (cit. on pp. 2, 4, 5, 34).
- [AW14] Amir Abboud and Virginia Vassilevska Williams. “Popular Conjectures Imply Strong Lower Bounds for Dynamic Problems”. In: *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*. 2014, pp. 434–443 (cit. on p. 4).
- [Awe85] Baruch Awerbuch. “Complexity of Network Synchronization”. In: *Journal of the ACM* 32.4 (1985). Announced at STOC’84, pp. 804–823 (cit. on p. 4).
- [BCH17] Sayan Bhattacharya, Deeparnab Chakrabarty, and Monika Henzinger. “Deterministic Fully Dynamic Approximate Vertex Cover and Fractional Matching in  $O(1)$  Amortized Update Time”. In: *Proceedings of the International Conference on Integer Programming and Combinatorial Optimization (IPCO)*. 2017, pp. 86–98 (cit. on p. 4).
- [BGS18] Surender Baswana, Manoj Gupta, and Sandeep Sen. “Fully Dynamic Maximal Matching in  $O(\log n)$  Update Time (Corrected Version)”. In: *SIAM Journal on Computing* 47.3 (2018). Announced at FOCS’11, pp. 617–650 (cit. on pp. 4, 5, 20, 22, 24, 25, 34).
- [Bha<sup>+</sup>18] Sayan Bhattacharya, Deeparnab Chakrabarty, Monika Henzinger, and Danupon Nanongkai. “Dynamic Algorithms for Graph Coloring”. In: *Proceedings of the Symposium on Discrete Algorithms (SODA)*. 2018, pp. 1–20 (cit. on p. 5).
- [BHI18] Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. “Deterministic Fully Dynamic Data Structures for Vertex Cover and Matching”. In: *SIAM Journal on Computing* 47.3 (2018). Announced at SODA’15, pp. 859–887 (cit. on p. 4).
- [BHN16] Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. “New Deterministic Approximation Algorithms for Fully Dynamic Matching”. In: *Proceedings of the Symposium on Theory of Computing (STOC)*. 2016, pp. 398–411 (cit. on p. 4).

- [BHN17] 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: *Proceedings of the Symposium on Discrete Algorithms (SODA)*. 2017, pp. 470–489 (cit. on pp. 4, 16).
- [BK16] Greg Bodwin and Sebastian Krinninger. “Fully Dynamic Spanners with Worst-Case Update Time”. In: *Proceedings of the European Symposium on Algorithms (ESA)*. 2016, 17:1–17:18 (cit. on pp. 2, 4, 16).
- [BKS12] Surender Baswana, Sumeet Khurana, and Soumojit Sarkar. “Fully Dynamic Randomized Algorithms for Graph Spanners”. In: *ACM Transactions on Algorithms* 8.4 (2012). Announced at SODA’08, 35:1–35:51 (cit. on pp. 4, 5, 9, 11–15).
- [CHK16] Keren Censor-Hillel, Elad Haramaty, and Zohar S. Karnin. “Optimal Dynamic Distributed MIS”. In: *Proceedings of the Symposium on Principles of Distributed Computing (PODC)*. 2016, pp. 217–226 (cit. on p. 5).
- [CS18] Moses Charikar and Shay Solomon. “Fully Dynamic Almost-Maximal Matching: Breaking the Polynomial Worst-Case Time Barrier”. In: *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*. 2018, 33:1–33:14 (cit. on pp. 2, 4, 34).
- [Elk11] Michael Elkin. “Streaming and Fully Dynamic Centralized Algorithms for Constructing and Maintaining Sparse Spanners”. In: *ACM Transactions on Algorithms* 7.2 (2011). Announced at ICALP’07, 20:1–20:17 (cit. on p. 4).
- [Fis17] Manuela Fischer. “Improved Deterministic Distributed Matching via Rounding”. In: *Proceedings of the International Symposium on Distributed Computing (DISC)*. 2017, 17:1–17:15 (cit. on p. 4).
- [Gib<sup>+</sup>15] David Gibb, Bruce M. Kapron, Valerie King, and Nolan Thorn. “Dynamic graph connectivity with improved worst case update time and sublinear space”. In: *CoRR abs/1509.06464* (2015). arXiv: 1509.06464 (cit. on p. 2).
- [GK18] Gramoz Goranci and Sebastian Krinninger. “Dynamic Low-Stretch Trees via Dynamic Low-Diameter Decompositions”. In: *CoRR abs/1804.04928* (2018). arXiv: 1804.04928 (cit. on p. 4).
- [GKP08] Fabrizio Grandoni, Jochen Könemann, and Alessandro Panconesi. “Distributed Weighted Vertex Cover via Maximal Matchings”. In: *ACM Transactions on Algorithms* 5.1 (2008). Announced at COCOON’05, 6:1–6:12 (cit. on p. 4).
- [GP13] Manoj Gupta and Richard Peng. “Fully Dynamic  $(1 + \epsilon)$ -Approximate Matchings”. In: *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*. 2013, pp. 548–557 (cit. on p. 4).
- [Gup<sup>+</sup>17] Anupam Gupta, Ravishankar Krishnaswamy, Amit Kumar, and Debmalya Panigrahi. “Online and Dynamic Algorithms for Set Cover”. In: *Proceedings of the Symposium on Theory of Computing (STOC)*. 2017, pp. 537–550 (cit. on p. 4).
- [Hen<sup>+</sup>15] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. “Unifying and Strengthening Hardness for Dynamic Problems via the Online Matrix-Vector Multiplication Conjecture”. In: *Proceedings of the Symposium on Theory of Computing (STOC)*. 2015, pp. 21–30 (cit. on p. 4).

- [HKP01] Michal Hanckowiak, Michal Karonski, and Alessandro Panconesi. “On the Distributed Complexity of Computing Maximal Matchings”. In: *SIAM Journal on Discrete Mathematics* 15.1 (2001). Announced at SODA’98, pp. 41–57 (cit. on p. 4).
- [KKM13] Bruce M. Kapron, Valerie King, and Ben Mountjoy. “Dynamic graph connectivity in polylogarithmic worst case time”. In: *Proceedings of the Symposium on Discrete Algorithms (SODA)*. 2013, pp. 1131–1142 (cit. on p. 2).
- [Lat<sup>+</sup>11] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. “Filtering: A Method for Solving Graph Problems in MapReduce”. In: *Proceedings of the Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2011, pp. 85–94 (cit. on p. 4).
- [NS16] Ofer Neiman and Shay Solomon. “Simple Deterministic Algorithms for Fully Dynamic Maximal Matching”. In: *ACM Transactions on Algorithms* 12.1 (2016). Announced at STOC’13, 7:1–7:15 (cit. on p. 4).
- [NSW17] Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. “Dynamic Minimum Spanning Forest with Subpolynomial Worst-Case Update Time”. In: *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*. 2017, pp. 950–961 (cit. on p. 2).
- [OR10] Krzysztof Onak and Ronitt Rubinfeld. “Maintaining a Large Matching and a Small Vertex Cover”. In: *Proceedings of the Symposium on Theory of Computing (STOC)*. 2010, pp. 457–464 (cit. on p. 4).
- [San04] Piotr Sankowski. “Dynamic Transitive Closure via Dynamic Matrix Inverse”. In: *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*. 2004, pp. 509–517 (cit. on p. 2).
- [San07] Piotr Sankowski. “Faster Dynamic Matchings and Vertex Connectivity”. In: *Proceedings of the Symposium on Discrete Algorithms (SODA)*. 2007, pp. 118–126 (cit. on p. 4).
- [Sol16] Shay Solomon. “Fully Dynamic Maximal Matching in Constant Update Time”. In: *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*. 2016, pp. 325–334 (cit. on p. 4).
- [SS18] Noam Solomon and Shay Solomon. “Reoptimization via Gradual Transformations”. In: *CoRR abs/1803.05825* (2018). arXiv: 1803.05825 (cit. on p. 34).